# Data Structures

## Sorting

CS284

# Objectives

- Learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - shell sort
  - merge sort
  - heapsort
  - quicksort
- Understand differences in performance of these algorithms

# Introduction

- Sorting entails arranging data in order
- Familiarity with sorting algorithms is an important programming skill
- The study of sorting algorithms provides insight into
  - problem solving techniques such as divide and conquer
  - the analysis and comparison of algorithms which perform the same task
- While the sort algorithms are not limited to arrays, throughout our lectures we will sort arrays for simplicity

# Using Java Sorting Methods

- The Java API provides a class `Arrays` with several overloaded sort methods for different array types
  - Items to be sorted must be `Comparable` objects, so, for example, **`int`** values must be wrapped in `Integer` objects
- The `Collections` class provides similar sorting methods for Lists
- Sorting methods for arrays of primitive types are based on the quicksort algorithm
- Sorting methods for arrays of objects and Lists are based on the merge sort algorithm
- Both algorithms are $\mathcal{O}(n \log n)$

# Selection Sort

- Make several passes through the array
- Select next smallest item in the array each time
- Place it where it belongs in the array

# Trace of Selection Sort

n = number of elements in the array `a`

```
for fill = 0 to n - 2  {
    posMin = index of the smallest item in
                        subarray a[fill..n-1]
    swap(a,posMin,fill);
}
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 35 | 65 | 30 | 60 | 20 |

- Let's follow the execution on the board

# Trace of Selection Sort Refinement

n = number of elements in the array `a`

```
for fill = 0 to n - 2 {
   posMin = fill
   for next = fill + 1 to n - 1 {
       if (a[next]<a[posMin])
           posMin = next
   }
   swap(a,posMin,fill);
 }
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 35 | 65 | 30 | 60 | 20 |

# Analysis of Selection Sort

```
for fill = 0 to n - 2 {
   posMin = fill
   for next = fill + 1 to n - 1 {
       if (a[next]<a[posMin])
           posMin = next
   }
   swap(a,posMin,fill);
}
```

- What is the complexity?

# Analysis of Selection Sort

```
for fill = 0 to n - 2 {
   posMin = fill
   for next = fill + 1 to n - 1 {
       if (a[next]<a[posMin])
           posMin = next
   }
   swap(a,posMin,fill);
}
```

- What is the complexity? $\mathcal{O}(n^2)$
- How many comparisons are performed?

# Analysis of Selection Sort

```
for fill = 0 to n - 2 {
   posMin = fill
   for next = fill + 1 to n - 1 {
       if (a[next]<a[posMin])
           posMin = next
   }
   swap(a,posMin,fill);
}
```

- What is the complexity? $\mathcal{O}(n^2)$
- How many comparisons are performed? $\mathcal{O}(n^2)$
- How many exchanges are performed

# Analysis of Selection Sort

```
for fill = 0 to n - 2 {
   posMin = fill
   for next = fill + 1 to n - 1 {
       if (a[next]<a[posMin])
           posMin = next
   }
   swap(a,posMin,fill);
}
```

- What is the complexity? $\mathcal{O}(n^2)$
- How many comparisons are performed? $\mathcal{O}(n^2)$
- How many exchanges are performed $\mathcal{O}(n)$

# Code for Selection Sort

```java
public class SelectionSort {
  public static <E extends Comparable<E>> void sort(E[] table) {
    int n = table.length;
    for (int fill = 0; fill < n-1; fill++) {
      // Invariant: table[0...fill-1] is sorted.
      int posMin = fill;

     for (int next = fill + 1; next < n; next++) {
     // Invariant: table[posMin] is the smallest item in
     // table[fill...next-1].
        if (table[next].compareTo(table[posMin]) < 0) {
             posMin = next;
        }
      }
      // Exchange table[fill] and table[posMin].
      E temp = table[fill];
      table[fill]     = table[posMin];
      table[posMin]   = temp;
    }
  }
}
```

Selection Sort

Bubble Sort

Insertion Sort

Comparison

# 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; hence the name

# Trace of Bubble Sort

```
do
  for each pair of adjacent array elements
    if the values in a pair are out of order
      Exchange the values
while the array in not sorted
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 60 | 42 | 75 | 83 | 27 |

# Trace of Bubble Sort

```
do
  for each pair of adjacent array elements
    if the values in a pair are out of order
      Exchange the values
while the array in not sorted
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 60 | 42 | 75 | 83 | 27 |

- At the end of pass 1, the last item (i.e. the one at index 4) is guaranteed to be in its correct position.
- There is no need to test it again in the next pass

# Trace of Bubble Sort

```
do
  for each pair of adjacent array elements
    if the values in a pair are out of order
      Exchange the values
while the array in not sorted
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 60 | 42 | 75 | 83 | 27 |

- At the end of pass 1, the last item (i.e. the one at index 4) is guaranteed to be in its correct position.
- There is no need to test it again in the next pass
- Where $n$ is the length of the array, after the completion of $n-1$ passes (4, in this example) the array is sorted

# Trace of Bubble Sort

- Sometimes an array will be sorted before $n-1$ passes.
- This can be detected if there are no exchanges made during a pass through the array

```
do
  exchanges=false;
  for each pair of adjacent array elements
    if the values in a pair are out of order {
      Exchange the values
      exchanges=true;
    }
while exchanges==true
```

# Analysis of Bubble Sort

- The number of comparisons and exchanges is represented by $(n-1)+(n-2)+...+3+2+1$
- Worst case:
  - number of comparisons is $\mathcal{O}(n^2)$
  - number of exchanges is $\mathcal{O}(n^2)$
- Compared to selection sort with its $\mathcal{O}(n^2)$ comparisons and $\mathcal{O}(n)$ exchanges, bubble sort usually performs worse
- If the array is sorted early, the later comparisons and exchanges are not performed and performance is improved
- Bubble sort works best on arrays nearly sorted and worst on inverted arrays (elements are in reverse sorted order)

# Code for Bubble Sort

```
public class BubbleSort {
  public static <E extends Comparable<E>> void sort(E[] table) {
    int pass = 1;
    boolean exchanges = false;
    do {
       // Invariant: Elements after table.length-pass+1
       // are in place.
       exchanges = false;
       // Compare each pair of adjacent elements.
       for (int i = 0; i < table.length - pass; i++) {
         if (table[i].compareTo(table[i + 1]) > 0) {
             // Exchange pair.
           E temp = table[i];
           table[i] = table[i + 1];
           table[i + 1] = temp;
           exchanges = true;
         }
      }
      pass++;
    } while (exchanges);
```

# Insertion Sort

- Based on the technique used by card players to arrange a hand of cards
- The player keeps the cards that have been picked up so far in sorted order
- When the player picks up a new card, the player makes room for the new card and then inserts it in its proper place

# Trace of Insertion Sort (for an Array `a`)

```
for each array element from the second (nextPos = 1) to the last {
   Insert a[nextPos] where it belongs in a, increasing
    the length of the sorted subarray by 1 element
}
```

- To adapt the insertion algorithm to an array that is filled with data, we start with a sorted subarray consisting of only the first element

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 30 | 25 | 15 | 20 | 28 |

- Let's follow the execution on the board

# Trace of Insertion Sort

```
for nextPos = 1 to n-1  {
   Insert a[nextPos] where it belongs in a, increasing
    the length of the sorted subarray by 1 element
}
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 30 | 25 | 15 | 20 | 28 |

nextPos

# Trace of Insertion Sort Refinement

```
for nextPos = 1 to n-1  {
  nextPos is the position of the element to insert;
  nextVal = a[nextPos];
  while (nextPos>0 and a[nextPos-1] > nextVal) {
    Shift the element at nextPos-1 to position nextPos;
    nextPos--;
  }
  Insert nextVal at nextPos;
}
```

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 30 | 25 | 15 | 20 | 28 |

- Let's follow the execution on the board

# Analysis of Insertion Sort

- The insertion step is performed $n - 1$ times
- In the worst case, all elements in the sorted subarray are compared to `nextVal` for each insertion
- The maximum number of comparisons will then be:

$$1 + 2 + 3 + ... + (n - 2) + (n - 1)$$

- which is $\mathcal{O}(n^2)$

# Analysis of Insertion Sort

- In the best case (when the array is sorted already):
  - only one comparison is required for each insertion
  - the number of comparisons is $\mathcal{O}(n)$
- 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, it is the same as the number of comparisons

# Code for Insertion Sort

```java
public class InsertionSort {
  /** Sort the table using insertion sort algorithm.
      pre:  table contains Comparable objects.
      post: table is sorted.
      @param table The array to be sorted
   */
  public static <E extends Comparable<E>>
     void sort(E[] table) {
     for (int nextPos = 1; nextPos < table.length; nextPos++) {
       // Invariant: table[0...nextPos-1] is sorted.
       // Insert element at position nextPos
       // in the sorted subarray.
       insert(table, nextPos);
     }
  }
```

# Code for Insertion Sort

```java
/** Insert the element at nextPos where it belongs
      in the array.
      pre:  table[0...nextPos-1] is sorted.
      post: table[0...nextPos] is sorted.
      @param table The array being sorted
      @param nextPos The position of the element to insert
   */
private static <E extends Comparable<E>>
   void insert(E[] table, int nextPos) {
       E nextVal = table[nextPos]; // Element to insert.
       while (nextPos > 0 &&
       nextVal.compareTo(table[nextPos - 1]) < 0) {
            table[nextPos] = table[nextPos - 1]; // Shift down.
            nextPos--; // Check next smaller element.
       }
       // Insert nextVal at nextPos.
       table[nextPos] = nextVal;
   }
}
```

# Comparison of Quadratic Sorts

| | Number of comparisons | | Number of exchanges | |
|---|---|---|---|---|
| | Best | Worst | Best | Worst |
| Selection sort | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Bubble sort | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n^2)$ |
| Insertion sort | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ |

# Comparison of Quadratic Sorts

- Insertion sort
  - gives the best performance for most arrays
  - takes advantage of any partial sorting in the array and uses less costly shifts
- Bubble sort generally gives the worst performance—unless the array is nearly sorted
  - big-O analysis ignores constants and overhead
- None of the quadratic search algorithms are particularly good for large arrays ($n > 1000$)
- The best sorting algorithms provide $n \log n$ average case performance

# Comparison of Quadratic Sorts

- All quadratic sorts require storage for the array being sorted
- However, the array is sorted in place
- While there are also storage requirements for variables, for large $n$, the size of the array dominates and extra space usage is $\mathcal{O}(1)$