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

Bubble Sort

Insertion Sort

Comparison

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 Refinement

n = number of elements in the array `a`

0	1	2	3	4
35	65	30	60	20

Trace of Selection Sort Refinement

First round: find the smallest item among 0...4, swap with 0-th element

35	65	30	60	20
20	65	30	60	35

Second round: find the smallest item among 1...4, swap with 1-th element

20	65	30	60	35
20	30	65	60	35

Third round: find the smallest item among 2...4, swap with 2-th element

20	30	65	60	35
20	30	35	60	65

Analysis of Selection Sort

- What is the complexity?

Analysis of Selection Sort

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

Analysis of Selection Sort

- 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

- 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

```
/**
 * Sort the table using selection sort algorithm.
 * swap & compare:
       swap count: best case O(n), worst case O(n^2)
       compare count: best case O(n^2), worst case O(n^2)
   Selection sort does more comparison, fewer swaps
 * @param a
 */
```

Code for Selection Sort

```
public void selection_sort(E[] a) {
      int n = a.length;
      for (int round = 0; round < n-1; round++) {
        int next_smallest_idx = round;

       /** find the smallest element starting from current_idx
        *  swap the current element with the smallest elemnt
        *  so after the round-th round, the current_idx-th
        *  mallest item is in place
        */
       for (int next_idx = round + 1; next_idx < n; next_idx++) {
           if (a[next_idx].compareTo(a[round]) < 0) {
                next_smallest_idx = next_idx;
           }
           compare_count += 1;
        }
        E temp = a[round];
        a[round]      = a[next_smallest_idx];
        a[next_smallest_idx]   = temp;
        swap_count += 1;
      }
}
```

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 largest item (i.e., 83 in this example) 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 largest item (i.e., 83 in this example) is guaranteed to be in its correct position.
- There is no need to test it again in the next pass
- At the end of pass i, the i-th largest item is guaranteed to be in its correct position, i.e., n - i

Trace of Bubble Sort

First round: from position 0...4, 83 is in place afterwards

60	42	75	83	27
42	60	75	83	27
42	60	75	27	83

Second round: from position 0...3, 75 is in place afterwards

42	60	75	27	83
42	60	27	75	83

Third round: from position 0...2, 60 is in place afterwards

42	60	27	75	83
42	27	60	75	83

Fourth round: from position 0...1, 42 is in place afterwards

42	27	60	75	83
27	42	60	75	83

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

```
/**
 * Sort the table using bubble sort algorithm.
 * swap & compare:
       swap count: best case O(1), worst case O(n^2)
       compare count: best case O(n), worst case O(n^2)
   Bubble sort always swap locally, so in general, more swap
   is needed to get every elements in place
 * @param a
 */
```

Code for Bubble Sort

```
public void bubble_sort(E[] a) {
      int round = 1;
      boolean exchanges = false;
      do {
         exchanges = false;
         /**
          * The pass-th round: swap adjacent out-of-order
          * elements from 0...a.length - round. At the end
          * of the round-th round, the round-th largest
          * element is in place */
         for (int i = 0; i < a.length - round; i++) {
           if (a[i].compareTo(a[i + 1]) > 0) {
              E temp = a[i];
              a[i] = a[i + 1];
              a[i + 1] = temp;
              swap_count += 1;
              exchanges = true;
           }
          compare_count += 1;
        }
         round++;
      } 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

First round: insert element at position 1 to 0..0

30	25	15	20	28
25	30	15	20	28

Second round: insert element at position 2 to 0..1

25	30	15	20	28
25	30	30	20	28
25	25	30	20	28
15	25	30	20	28

Trace of Insertion Sort

Third round: insert element at position 3 to 0..2

15	25	30	20	28
15	25	30	30	28
15	25	25	30	28
15	20	25	30	28

Fourth round: insert element at position 4 to 0..3

15	20	25	30	28
15	20	25	30	30
15	20	25	28	30

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 the next element 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

```
/** Sort the table using insertion sort algorithm.
 pre:  table contains Comparable objects.
 post: table is sorted.
 swap & compare:
    swap count: best case O(n), worst case O(n^2)
    compare count: best case O(n), worst case O(n^2)
    swap count is similar to compare count
 @param table The array to be sorted
*/
public void insertion_sort(E[] table) {
    for (int nextPos = 1; nextPos < table.length; nextPos++) {
      insert(table, nextPos);
    }
 }
```

Code for Insertion Sort

```
/** Insert the element at next_idx where it belongs
in the array.
pre:  table[0...next_idx-1] is sorted.
post: table[0...next_idx] is sorted.
@param table The array being sorted
@param nextPos The position of the element to insert
*/
public void insert(E[] a, int next_idx) {
        E next_val = a[next_idx];

        while (next_idx > 0 &&
        next_val.compareTo(a[next_idx - 1]) < 0) {
             a[next_idx] = a[next_idx - 1];
             swap_count += 1;
             compare_count += 1;
             next_idx--;
        }
        a[next_idx] = next_val;
        swap_count += 1;
 }
```

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 $\mathcal{O}(1)$