SORTING

Chapter 8

Sorting

Why sort?
To make searching faster!

How?
Binary Search gives log(n) performance.

There are many algorithms for sorting: bubble sort, selection sort, insertion sort, quick sort, heap sort, …

Why so many?

First we will learn some of them and perhaps we will be able to answer this question.

[Psst: While performance has a lot to do with it, it isn't always about that!]

Declaring a Generic Method (cont.)

```
public static void sort (Comparable[] table)
{

    // use x.CompareTo(y) to sort
    // returns a value <0, if x < y
    // return a value == 0, if x == y
    // >0 if x > y

} // sort()
```

Selection Sort

Section 8.2

Selection Sort

- Sorts an array by making several passes through the array, selecting a next smallest item in the array each time and placing it where it belongs in the array

```
for (int fill=0; fill < n-1; fill++) {
   // find posMin in A[i=fill]..A[n-1]
   // such that A[posMin] is the smallest
   // Swap/exchange items A[fill] and A[posMin]
}
```

Analysis of Selection Sort

This loop is performed n-1 times

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Analysis of Selection Sort (cont.)

There are n-1 exchanges

```
1. for fill = 0 to n - 2 do
2.     Initialize posMin to fill
3.     for next = fill + 1 to n - 1 do
4.         if the item at next is less than the item at posMin
5.             Reset posMin to next
6.     Exchange the item at posMin with the one at fill
```

Analysis of Selection Sort (cont.)

This comparison is performed $(n - 1 - fill)$ times for each value of *fill* and can be represented by the following series:
$(n-1) + (n-2) + \ldots + 3 + 2 + 1$

```
1. for fill = 0 to n - 2 do
2.     Initialize posMin to fill
3.     for next = fill + 1 to n - 1 do
4.         if the item at next is less than the item at posMin
5.             Reset posMin to next
6.     Exchange the item at posMin with the one at fill
```

Analysis of Selection Sort (cont.)

The series
$(n\text{-}1) + (n\text{-}2) + \ldots + 3 + 2 + 1$
is a well-known series and can be written as

$$\frac{n \times (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Analysis of Selection Sort (cont.)

For very large n we can ignore all but the significant term in the expression, so the number of
- comparisons is $O(n^2)$
- exchanges is $O(n)$

An $O(n^2)$ sort is called a *quadratic sort*

1. **for** `fill = 0` to `n - 2` do
2. Initialize `posMin` to `fill`
3. **for** `next = fill + 1` to `n - 1` do
4. **if** the item at `next` is less than the item at `posMin`
5. Reset `posMin` to next
6. Exchange the item at `posMin` with the one at `fill`

Code for Selection Sort (cont.)

```
public static void sort (Comparable[] table) {
   int n = table.length;
   for (int fill=0; fill < n-1; fill++) {
      //Initialize posMin to fill
      int posMin = fill;
      for (int next = fill + 1; next < n; next++) {
         // if the item at next is less than the item at posMin
        if (table[next].compareTo(table[posMin]) < 0) {
           // Reset posMin to next
           posMin = next;
      // Exchange the item at posMin with the one at fill
      Comparable temp = table[fill];
      table[fill] = table[posMin];
      table[posMin] = temp;
   }
} // sort()
```

Bubble Sort

Section 8.3

Bubble Sort

- Also a quadratic 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 (cont.)

pass	4
exchanges made	1

[0]	27
[1]	42
[2]	60
[3]	75
[4]	83

1. Initialize `exchanges` to **`false`**
2. **`for`** each pair of adjacent array elements
3. **`if`** the values in a pair are out of order
4. Exchange the values
5. Set `exchanges` to **`true`**

The algorithm can be modified to detect exchanges

Analysis of Bubble Sort

- The number of comparisons and exchanges is represented by

$$(n - 1) + (n - 2) + \ldots + 3 + 2 + 1$$

- Worst case:
  - number of comparisons is $O(n^2)$
  - number of exchanges is $O(n^2)$
- Compared to selection sort with its $O(n^2)$ comparisons and $O(n)$ exchanges, bubble sort usually performs worse
- If the array is sorted early, the later comparisons and exchanges are not performed and performance is improved

Analysis of Bubble Sort (cont.)

- The best case occurs when the array is sorted already
  - one pass is required ($O(n)$ comparisons)
  - no exchanges are required ($O(1)$ exchanges)
- Bubble sort works best on arrays nearly sorted and worst on *inverted* arrays (elements are in reverse sorted order)

Code for Bubble Sort

```
public static void sort (Comparable[] table) {
   int n = table.length;
   do {
      exchanged = false;
      for (int i=0; i < n-1; i++) {
         if (table[i].compareTo(table[j]) > 0)
            // Exchange table[i] and table[i+1]
            Comparable temp = table[i];
            table[i] = table[i+1];
            table[i+1] = temp;
            exchanged = true;
      while (swapped);
} // sort()
```

Insertion Sort

Section 8.4

Insertion Sort

- Another quadratic sort, *insertion* sort, is 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

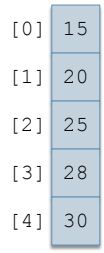
[0]	30
[1]	25
[2]	15
[3]	20
[4]	28

1. **`for`** each array element from the second (**`nextPos`** = 1) to the last
2. Insert the element at **`nextPos`** where it belongs in the array, 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

Trace of Insertion Sort Refinement (cont.)

nextPos	3
nextVal	28



1. **for** each array element from the second (`nextPos = 1`) to the last
2. `nextPos` is the position of the element to insert
3. Save the value of the element to insert in `nextVal`
4. **while** `nextPos > 0` and the element at `nextPos - 1 > nextVal`
5. Shift the element at `nextPos - 1` to position `nextPos`
6. Decrement `nextPos` by `1`
7. Insert `nextVal` at `nextPos`

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 then will be:

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1)$$

- which is $O(n^2)$

Analysis of Insertion Sort (cont.)

- In the best case (when the array is sorted already), only one comparison is required for each insertion
- In the best case, the 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, it is the same as the number of comparisons
- A shift in an insertion sort requires movement of only 1 item, while an exchange in a bubble or selection sort involves a temporary item and the movement of three items
  - The item moved may be a primitive or an object reference
  - The objects themselves do not change their locations

Code for Insertion Sort

```
public static void sort (Comparable[] table) {
    // for each array element from second (nextPos = 1) to the last
    for (int nextPos = 1; nextPos < n; nextPos++) {
        // nextPos  is the position of the element to insert
        // Save the value of the element to insert in nextVal
        Comparable nextVal = table[nextPos];
        //while nextPos>0 and element at nextPos–1> nextVal
        while (nextPos > 0
                && table[nextPos-1].compareTo(nextVal) > 0){
           // Shift the element at nextPos – 1 to position nextPos
           table[nextPos] = table[nextPos-1];
           // Decrement nextPos by 1
           nextPos--;
        }
        // Insert nextVal at nextPos
      table[nextPos] = nextVal;
    }
} // sort()
```

Comparison of Quadratic Sorts

Section 8.5

Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

Comparison of Quadratic Sorts (cont.)

Comparison of growth rates

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Comparison of Quadratic Sorts (cont.)

- 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 (cont.)

- 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 O(1)