CMSC 131: Chapter 30 (Supplement)
Sorting

Sorting and Searching

Sorting: Given a **list** of items from any ordered domain (integers, doubles, Strings, Dates, …) **permute** (rearrange) the elements so they are in **increasing order**.

Searching: Given a **list** of items and given a designated **query item** q, determine **whether** q appears in the list, and if so, then **where**.

Sorting and Searching: are two of the most fundamental tasks in all of computer science.

- Although these may seem pretty simple, there are many different ways to solve these problems.
- These approaches are quite different in terms of **complexity** and **efficiency**.
- There is a 722 page book (by D. E. Knuth) devoted to just these two topics alone!

Note: We will talk only about Sorting (one type of Sort: Selection Sort). You will see Search in CMSC 132.

Selection Sort

Selection Sort: one of the simplest sorting algorithms known. (**Recall**: an **algorithm** is a method of solving a problem.)

Input Argument: Let **a** denote the array to be sorted. We assume only that the elements of a are from any class that implements the **Comparable interface**, that is, it defines a public method:

```
int compareTo( Object obj )
```

This compares the current object to object obj. It returns:

< 0	**if**	this < obj
== 0	**if**	this == obj
> 0	**if**	this > obj

Examples of Objects that implement Comparable:

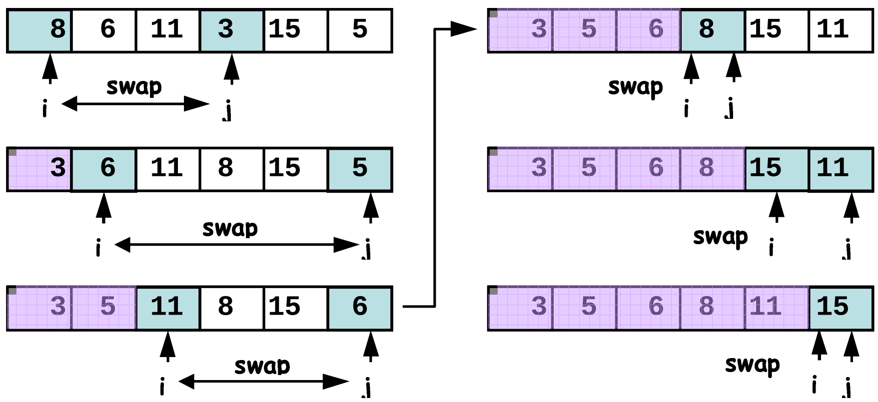
- All the **numeric wrappers** (Integer, Double, Float, etc.)
- **String**
- Any class of yours, for which you define **compareTo**( ).

Selection Sort Algorithm

Selection Sort: Works as follows. Let a[0 … n-1] be the array to be sorted.

```
for ( i running from 0 up to n-1 ) {
    Let j = index of the smallest of a[i], a[i+1], … a[n-1];
    swap a[i] with a[j];
}
```

Example:



Selection Sort

```
public static void selectionSort( Comparable[ ] a ) {
    for ( int i = 0; i < a.length; i++ ) {
        int j = indexOfMin( i, a );
        swap( i, j, a );
    }
}

private static int indexOfMin( int start, Comparable[ ] a ) {
    Comparable min = a[start];
    int minIndex = start;
    for ( int i = start + 1; i < a.length; i++ )
        if ( a[i].compareTo( min ) < 0 ) {
            min = a[i];
            minIndex = i;
        }
    return minIndex;
}

private static void swap( int i, int j, Comparable[ ] a ) {
    Comparable temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Using Selection Sort

Polymorphism: Selection sort is polymorphic in the sense that it can be applied to any array whose elements implement the **Comparable** interface.

Example:

```
Integer[ ] list1 = { new Integer( 8 ), new Integer( 6 ),
       /* blah, blah, blah… */   new Integer( 5 ) };
String[ ] list2 = { "Carol", "Bob", "Ted", "Alice", "Schultzie" };
selectionSort( list1 );
selectionSort( list2 );
```

Running time of Selection Sort

Efficiency: How long does Selection Sort take to run?

What should we count?

- **Milliseconds of execution time**?
  - Depends on the speed of your particular **computer**.
  - We prefer a **platform-independent measure.**

- **Statements of Java code that are executed**?
  - This depends on the **programmer**.
  - We would like a quantity that is a function of the algorithm, not the specific way it was coded.

- **Number of times we call compareTo( )**?
  - This is an acceptable machine/programmer-independent statistic.
  - This method is called every time through the innermost loop and **depends only on the algorithm**.

Running time of Selection Sort

Running time depends on the contents of the array:

- **Length**: The principal determinant of running time is the number of elements, **n**, in the array.
- **Contents**: For some sorting algorithms, the running time may depend on the contents of the array. E.g., some algorithms run faster if the initial array is nearly sorted.
- **Worst-case running time**: Among all arrays of length n, consider the one that has the **highest** running time.
- **Average-case running time**: **Average** the running time over all arrays of length n. (This is very **messy**, so we won't do it.)

Worst-case running time:

- Let **T(n)** denote the time (measured as the number of calls to compareTo( )) required in the worst-case to sort an array of n items using Selection Sort.

Running time of Selection Sort

How many times is compareTo( ) called? Call this **T(n)**.

- Let **n** denote the length of array a.
- We go through the for-loop in selectionSort( ) exactly **n times**,
  for i = 0, 1, 2, …, n-1.
- Each time we call indexOfMin(i, a), we call compareTo( ) to compare the min to each element of the **subarray a[i+1, …, n-1]**.
- In general, any subarray a[j, …, k] contains **k – j + 1 elements**.
- Thus, each call to indexOfMin(i, a) makes (n-1) - (i+1) + 1 = **n–i–1** calls to compareTo( ).
- To compute T(n), we simple add up **(n–i–1)**, for i = 0, 1, …, n-1:

i=0	(n-0-1)	= n-1
i=1	(n-1-1)	= n-2
i=2	(n-2-1)	= n-3
…		
i=n-2	(n-(n-2)-1)	= 1
i=n-1	(n-(n-1)-1)	= 0

$$\boxed{\begin{aligned} T(n) &= 0 + 1 + \cdots + (n-3) + (n-2) + (n-1) \\ &= \sum_{i=0}^{n-1} i \end{aligned}}$$

Running time of Selection Sort

What is the value of this sum?

$T(n) = 0 + 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$

An old addition trick: Group terms to get a common sum of values:

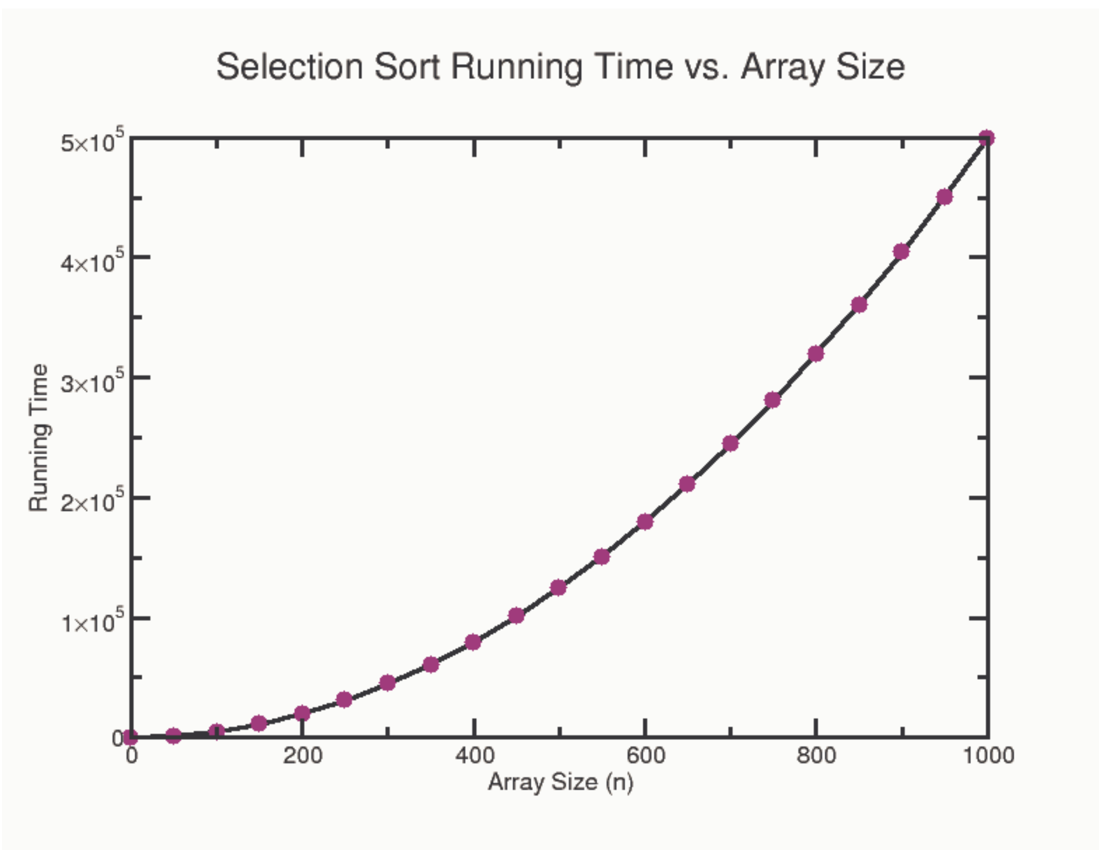
$T(n) = 0 + 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$
$= (1 + (n-1)) + (2 + (n-2)) + (3 + (n-3)) + \ldots$
$= n + n + n + \ldots$

There are roughly **(n-1)/2** pairs, each of which sums to **n**. Thus, the total value is roughly n(n-1)/2. (In fact, this is exactly correct.)

Final running time:

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

Plot of Running Time vs. Array Size



Big-"Oh" Notation

Observation: Efficiency is most critical for large n.

- As n becomes large, the quadratic $n^2/2$ term grows much more **rapidly** than the linear n/2 term.

- Small constant factors in running time like (1/2) depend on the **programmer's implementation**, and are best ignored.

- By ignoring the effects of

  - **lower order terms** (like n/2)
  - constant factors (like 1/2)

  we say that the running time grows **quadratically with n**, that is, "**on the order of n^2**", or succinctly **$O(n^2)$**.

Big-"Oh" Notation

"Big-Oh" Notation: a concise way of expressing the running time of an algorithm by ignoring less critical issues such as

- **lower order (slower growing) terms** and
- **constant multiplicative factors**.

Thus, the running time: T(n) is simply $O(n^2)$.

Formal Mathematical Definition of "Big-Oh":

- We will leave this for later courses (CMSC 250 and 351).