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!

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:

```java
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.

```java
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:

```
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:

```java
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 simply add up \( (n-i-1) \), for \( i = 0, 1, ..., n-1 \):

\[
T(n) = 0 + 1 + \cdots + (n-3) + (n-2) + (n-1) = \sum_{i=0}^{n-1} i
\]
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

Selection Sort Running Time vs. Array Size

![Graph showing the running time of Selection Sort vs. array size](image)
Observation: Efficiency is most critical for large \( n \).

- As \( n \) becomes large, the quadratic \( \frac{n^2}{2} \) term grows much more rapidly than the linear \( \frac{n}{2} \) term.

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

- By ignoring the effects of
  
  - lower order terms (like \( \frac{n}{2} \))
  - constant factors (like \( \frac{1}{2} \))

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: 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).
Other Sorting Algorithms

We have seen that Selection Sort is an $O(n^2)$ sorting algorithm.

Q: Are their faster algorithms?
A: Yes. And there are also slower ones!

Survey of common sorting algorithms:

**Bubble Sort**: Probably the most famous of all sorting algorithms, the easiest to implement, and the slowest as well. $O(n^2)$ time.

**Insertion Sort**: Similar to Selection Sort. $O(n^2)$ time.

**Tree Sort**: An efficient sorting algorithm, and one that is easy to visualize, but hard to implement. $O(n \log n)$ time.

**Quick Sort, Merge Sort, Heap Sort**: These are all $O(n \log n)$ time algorithms, and are considered to be the fastest sorting algorithms. They are covered in later courses (CMSC 351).

Bubble Sort: Sorting by Swapping

**Bubble Sort**: is perhaps the simplest of all sorting algorithms, and so it is the one that every programmer remembers. However, it is also the slowest of all the common sorting algorithms.

- Scan each consecutive pair of elements, starting from the left end of the array.

- If a pair is out of order, then swap them.

- If you make it through the entire array without swapping any, then the array is sorted. Otherwise, go back to the start and repeat.

**Minor optimization**: The largest element will "bubble up" to the last position of the array. So, after the $i^{th}$ pass, the largest $i$ elements are in their final places and do not need to be considered.

**Running time**: is $O(n^2)$. In the worst case, you need to make $n$ passes through the array. The $i^{th}$ pass needs perform $n-i-1$ comparisons. (Similar to Selection Sort.)
Example:

Scan 1:

8 6 11 3 15 5

6 8 11 3 15 5

6 8 11 3 15 5

6 8 11 3 15 5

6 8 3 11 15 5

6 8 3 11 15 5

6 8 3 11 15 5

6 8 3 11 15 5

Scan 2:

6 8 3 11 5 15

6 8 3 11 5 15

6 8 3 11 5 15

6 3 8 11 5 15

6 3 8 11 5 15

6 3 8 11 5 15

6 3 8 11 5 15

6 3 8 11 5 15

Scan 3:

6 8 3 11 15 5

6 8 3 11 15 5

6 8 3 11 15 5

6 3 8 11 15 5

3 6 8 5 11 15

3 6 8 5 11 15

3 6 8 5 11 15

3 6 8 5 11 15

Scan 4:

3 6 5 8 11 15

3 6 5 8 11 15

3 6 5 8 11 15

3 5 6 8 11 15

Scan 5 will do no swapping, so the algorithm terminates after this scan.
Insertion Sort

**Insertion Sort**: is another simple sorting algorithm. It is very similar to how someone sorts a hand of cards.

- Let i run from 1 up to n-1.
- At any time the subarray a[0 ... i-1] are already sorted. Our goal is to insert a[i] into its sorted position.
- To do this, we shift elements up to make room for the new item.

**Pseudo-code**:

```plaintext
for i = 1 up to n-1 {
    curr = a[i]
    j = i-1
    while (j ≥ 0 and a[j] > curr) {
        a[j+1] = a[j]; j--;
    }
    a[j+1] = curr;
}
```

**Example**:

```
8 6 11 3 15 5
6 8 11 3 15 5
6 8 11 3 15 5
3 6 8 11 15 5
3 6 8 11 15 5
3 5 6 8 11 15
Final result
```
Tree Sort

The previous algorithms (Selection Sort, Bubble Sort, and Insertion Sort) all run in $O(n^2)$ time. We next consider a more efficient algorithm, which runs in $O(n \log n)$ time.

Tree Sort:

- Insert the elements one by one into a binary tree structure. Each node of the tree holds one element.
- If the new element is less than the value in the current node, it is inserted on the left, otherwise it is inserted on the right.
- When we fall out of the bottom of the tree, we add a new node for the new element.
- Traverse the tree from left to right to obtain the final sorted sequence.
**Tree Sort**

**Implementation:** Tree Sort requires knowledge of linked structures. This will be covered in CMSC 132.

**Efficiency:** In spite of its added complexity, Tree Sort is much more efficient than the previous \( O(n^2) \) algorithms. Why?

- The time needed to add each element to the tree is proportional to the number of levels in the tree.
- If the tree is pretty well balanced, the number of levels is \( O(\log n) \). This is because the number of nodes at each level grows exponentially. Since there are \( n \) nodes, the total number of levels is \( O(\log n) \).
- So, the overall running time of tree sort is \( O(n \log n) \).
How Efficient?

Sorting Running Time vs. Array Size

Searching

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

Linear Search: Visit each element of the array in sequential order, say, by using a for-loop. Here is an example of searching an ArrayList for some query object q. We assume that the elements of the ArrayList have overridden equals().

```java
public static boolean find1( ArrayList a, Object q ) {
    for ( int i = 0; i < a.size(); i++ ) {
        if ( a.get(i).equals( q ) ) return true;
    }
    return false;
}
```
Binary Search

Can we search faster than linear search?

- If the list is not sorted, then no. The query item could be anywhere.
- If the list is sorted (e.g., like a telephone book) we can do much better.

**Binary Search**: repeated bisection. Let $q$ be the query item.

- Access the element in the **middle** of the list, call it $x$.
- If $q == x$, we are **done**.
- If $q < x$, continue the search on the **left half**.
- If $q > x$, continue the search on the **right half**.

**Example**

**Binary search** gets its name because it repeatedly cuts the list in half. So it only requires $O(\log n)$ probes to find the element (or determine it is not there).

**Example**: Find $q = 28$.

```
12 14 16 17 19 20 22 25 26 28 31 33 35 37
```

$22 < 28$

```
12 14 16 17 19 20 22 25 26 28 31 33 35 37
```

$31 > 28$

```
12 14 16 17 19 20 22 25 26 28 31 33 35 37
```

$26 < 28$

```
12 14 16 17 19 20 22 25 26 28 31 33 35 37
```

Found it!
Iterators

ArrayList is inherited from an abstract class called AbstractList.

- Java provides many different data structures that are inherited from AbstractList, e.g. linked lists, binary trees, hash tables.
- Most of these structures do not provide a method like get(i) for accessing elements.
- They all provide a device for enumerating all the elements of the data structure: Iterator. An iterator is an object that allows you to enumerate the elements of a list, one by one.

How Iterators work: Let list be an ArrayList (or any class inherited from AbstractList).

Iterator x = list.Iterator( ): Creates a new iterator object for list. It is positioned at the start of the list.

x.next( ): Returns the next element of the list (an Object), and advances the iterator. (Throws an exception if none left.)

x.hasNext( ): Returns true if more elements remain in the list.

Example

Print all the elements of an ArrayList:

    public static void print( ArrayList a ) {
        Iterator it = a.iterator( );
        while ( it.hasNext( ) )
            System.out.println( it.next( ) );
    }

Perform a linear search in an ArrayList:

    public static boolean find2( ArrayList a, Object q ) {
        Iterator it = a.iterator( );
        while ( it.hasNext( ) ) {
            if ( it.next( ).equals( q ) ) return true;
        }
        return false;
    }