ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.3.
Assistances and comments will be acknowledged.

8.3 Bubble Sort

An alternate approach to sorting is *bubble sort*: Start at the front of the array and check if the first two entries are in order. If not, swap them and proceed to compare the second and third, and so on. After one pass, the largest entry is guaranteed to be in the last location of the array; that is, the largest entry is *bubbled* to the top. Next, starting at the front again, repeat the process but now it is only necessary to check up the second-last entry. Repeat this process for a total of $n - 1$ times and the array is sorted.

This appears to be similar to the insertion sort algorithm; however, it has a few fatal flaws:

> "..the generic bad algorithm..."
> The Jargon File

> "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems"
> Donald Knuth

8.3.1 Implementation

The implementation of the algorithm as described is shown here:

```
template <typename Type>
void bubble_sort( Type *const array, int const n ) {
    for ( int i = n - 1; i > 0; --i ) {
        for ( int j = 0; j < i; ++j ) {
            if ( array[j] > array[j + 1] ) {
                std::swap( array[j], array[j + 1] );
            }
        }
    }
}
```

The run time of this algorithm is easy to calculate:

$$\sum_{k=1}^{n-1}(n-k) = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = \Theta\left(n^2\right).$$

The following table demonstrates how this list of six entries may be sorted after five passes:

7	14	12	33	5	19
7	12	14	5	19	**33**
7	12	5	14	**19**	**33**
7	5	12	**14**	**19**	**33**
5	7	**12**	**14**	**19**	**33**
5	**7**	**12**	**14**	**19**	**33**

8.3.2 Improvements

Clearly, bubble sort running in $Q(n^2)$ time is sub-optimal, so we will try to reduce the run time by:

1. Reducing the number of swaps,
2. Halting if the array is sorted early,
3. Limiting the range on which run the algorithm, and
4. Alternating between bubbling up and sinking down.

8.3.2.1 Reducing the Number of Swaps

With our first attempt, we reduce the number of swaps:

```
template <typename Type>
void bubble( Type *const array, int const n ) {
    for ( int i = n - 1; i > 0; --i ) {
        Type max = array[0];             // assume a[0] is the max

        for ( int j = 1; j <= i; ++j ) {
            if ( array[j] < max ) {
                array[j - 1] = array[j];   // move array[j] to the left
            } else {
                array[j – 1] = max;        // store the old max
                max = array[j];            // get the new max
            }
        }

        array[i] = max;                           // store the max
    }
}
```

While this speeds up the algorithm somewhat, it will not decrease the asymptotic run time.

8.3.2.2 Flagged Bubble Sort

Consider the following sort:

3	9	5	1	0	2	6	8	4	7
3	5	1	0	2	6	8	4	7	**9**
3	1	0	2	5	6	4	7	**8**	**9**
1	0	2	3	5	4	6	**7**	**8**	**9**
0	1	2	3	4	5	**6**	**7**	**8**	**9**

Notice that with the next pass, no swaps will be made: if no swaps are made, the array is sorted.

```
template <typename Type>
void bubble( Type *const array, int const n ) {
    for ( int i = n - 1; i > 0; --i ) {
        Type max = array[0];
        bool sorted = true;             // Assume the array is sorted

        for ( int j = 1; j < i; ++j ) {
            if ( array[j] < max ) {
                array[j - 1] = array[j];
                sorted = false;         // If we move an entry, the array is not sorted
            } else {
                array[j – 1] = max;
                max = array[j];
            }
        }

        array[i] = max;

        if ( sorted ) {                 // Break if we didn't move any entries
            break;
        }
    }
}
```

We will see that this reduces the run time down to $\Theta(n + d)$ where d is the number of inversions; however, it is not as efficient as insertion sort.

8.3.2.3 Range-limited Bubble Sort

Consider the following sort:

4	3	9	1	2	0	5	6	7	8
3	4	1	2	0	5	6	7	8	**9**
3	1	2	0	4	5	6	7	**8**	**9**
1	2	0	3	4	5	6	**7**	**8**	**9**
1	0	2	3	4	5	**6**	**7**	**8**	**9**
0	1	2	3	4	**5**	**6**	**7**	**8**	**9**

Notice that after the third pass, we still had to check that 4, 5, and 6 were in order each time—even though there were no swaps after that. If in a pass we make no swaps after the k^{th} entry, this means that the array must be sorted beyond this point. Thus, we could use a variable to store when the last swap was made and with the next pass, we'd only have to go up to that point.

```
template <typename Type>
void bubble( Type *const array, int const n ) {
    for ( int i = n - 1; i > 0; ) {
        Type max = array[0];
        int ti = 0;                         // Assume everything is sorted

        for ( int j = 1; j < i; ++j ) {
            if ( array[j] < max ) {
                array[j - 1] = array[j];
                ti = j - 1;                 // We made a swap, so assume everything after
            } else {                        // this point is sorted...
                array[j – 1] = max;
                max = array[j];
            }
        }

        array[i] = max;
        i = ti;                              // Set i to the last entry that was moved
    }
}
```

We will see that this does not, however, improve the performance over a flagged bubble sort.

8.3.2.4 Alternating Bubble Sort

As a final solution, consider the idea of alternating between bubbling the largest entry to the top and sinking the smallest entry down to the bottom.

3	9	5	1	0	2	6	8	4	7
3	5	1	0	2	6	8	4	7	**9**
0	3	5	1	2	4	6	8	7	**9**
0	3	1	2	4	5	6	7	**8**	**9**
0	**1**	2	3	4	5	6	7	**8**	**9**

This will make a significant improvement; however, the cost is significant complexity to the code.

```
template <typename Type>
void bubble( Type *const array, int n ) {
        int lower = 0;
        int upper = n - 1;

        while ( true ) {
                int new_upper = lower;

                for ( int i = lower; i < upper; ++i ) {
                        if ( array[i] > array[i + 1] ) {
                                Type tmp = array[i];
                                array[i] = array[i + 1];
                                array[i + 1] = tmp;
                                new_upper = i;
                        }
                }

                upper = new_upper;

                if ( lower == upper ) {
                        break;
                }

                int new_lower = upper;

                for ( int i = upper; i > lower; --i ) {
                        if ( array[i - 1] > array[i] ) {
                                Type tmp = array[i];
                                array[i] = array[i - 1];
                                array[i - 1] = tmp;
                                new_lower = i;
                        }
                }

                lower = new_lower;

                if ( lower == upper ) {
                        break;
                }
        }
}
```

8.3.3 Empirical Run-time Analysis

Because bubble sort swaps adjacent entries, it cannot be any better than insertion sort. Unfortunately, whereas insertion sort performs $n + d$ comparisons and d swaps, bubble sort is not so selective: there are often a significant number of unnecessary comparisons. For example, in bubble sort, if the first n entries are already in order, they will still have to be compared with every pass.

To compare the run times of insertion sort and bubble sort, we created a total of 32768 arrays of size 1024, each of which had somewhere between 10000 and $523776 = \frac{1024 \cdot 1023}{2}$ inversions. For each of these arrays, we place a dot in the location (d, c) where d is the number of inversions and c was the number of required comparisons.

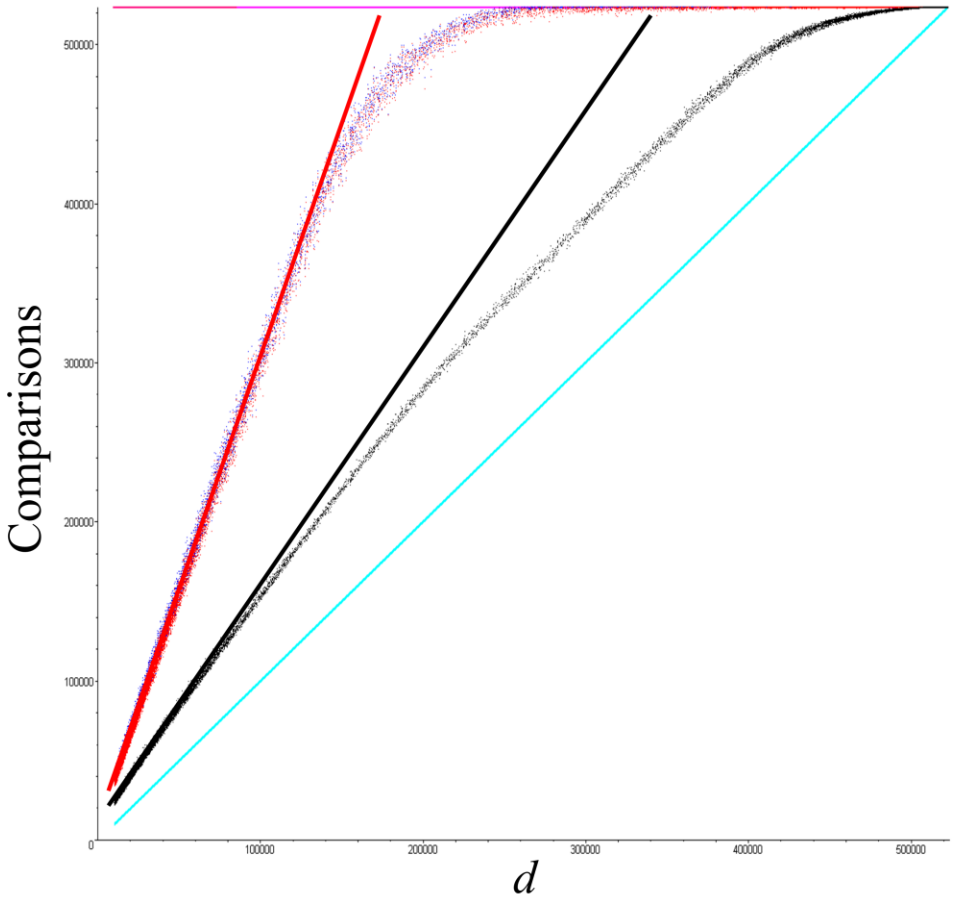

Figure 1. The number of comparisons required for sorting.

The magenta line at the top is the number of comparisons required for the default algorithm—if we never stop early, we must do all 523776 comparisons. The blue line is the number of comparisons required for insertion sort: $n + d$.

Next, we see a fuzzy section of brown and red dots. These are the number of comparisons required for both the flagged and range-limiting bubble sorts. In each case, the number of comparisons can be

approximated by $n + 3d$—that is, we must do $2d$ more comparisons than insertion sort. If you look closely, the range-limiting bubble sort is slightly below the flagged bubble sort, but the difference is negligible.

Finally, there are the points for the alternating bubble sort. Now the initial number of comparisons begins to grow according to $n + 1.5d$. Even this is not as efficient as insertion sort—50 % more comparisons must be made for each inversion that appears in the array.

8.3.4 Run-times

Thus, with a few improvements, bubble sort can be made to have the same asymptotic run time as insertion sort; however, the run time will never be comparable—it will always be significantly slower.

Inversions	Run Time	Comments
Worst	$\Theta(n^2)$	*E.g.*, reverse sorted
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Only if $d = O(n)$