Bubble sort

Outline

The second sorting algorithm is the $\mathbf{O}(n^2)$ bubble sort algorithm

- Uses an opposite strategy from insertion sort

We will examine:

- The algorithm and an example
- Run times
  - best case
  - worst case
  - average case (introducing *inversions*)
- Summary and discussion

Description

Suppose we have an array of data which is unsorted:

- Starting at the front, traverse the array, find the largest item, and move (or *bubble*) it to the top
- With each subsequent iteration, find the next largest item and *bubble* it up towards the top of the array

Description

As well as looking at good algorithms, it is often useful too look at sub-optimal algorithms

- Bubble sort is a simple algorithm with:
  - a memorable name, and
  - a simple idea

- It is also significantly worse than insertion sort

Observations

Some thoughts about bubble sort:

- the Jargon file states that bubble sort is

  “the generic bad algorithm”

- Donald Knuth comments that

  “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”

Obama on bubble sort

When asked the most efficient way to sort a million 32-bit integers, Senator Obama had an answer:

http://www.youtube.com/watch?v=k4RRi_ntQc8

Implementation

Starting with the first item, assume that it is the largest

Compare it with the second item:

- If the first is larger, swap the two,
- Otherwise, assume that the second item is the largest

Continue up the array, either swapping or redefining the largest item

Implementation

After one pass, the largest item must be the last in the list

Start at the front again:

- the second pass will bring the second largest element into the second last position

Repeat $n - 1$ times, after which, all entries will be in place

Implementation

The default algorithm:

```
template <typename Type>
void bubble( 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 Basic Algorithm

Here we have two nested loops, and therefore calculating the run time is straight-forward:

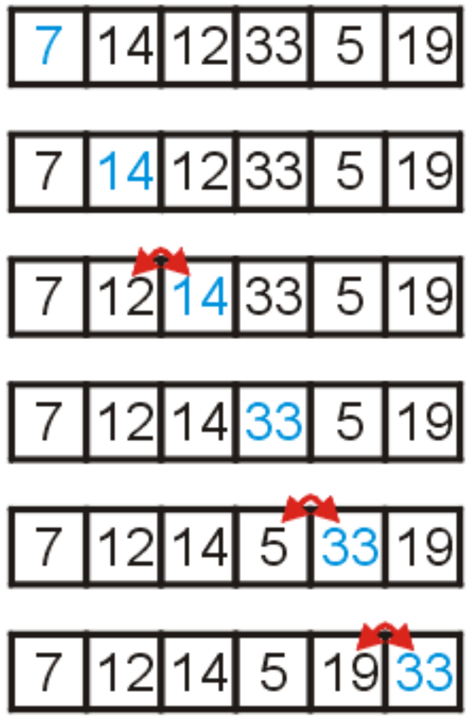
$$\overset{n\quad 1}{\underset{k=1}{}}(n\quad k) = n(n\quad 1)\quad \frac{n(n\quad 1)}{2} = \frac{n(n\quad 1)}{2} = \quad (n^2)$$

Example

Consider the unsorted array to the right

We start with the element in the first location, and move forward:

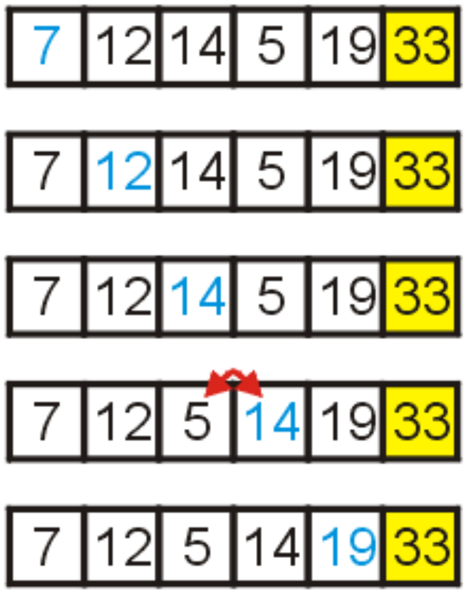
- if the current and next items are in order, continue with the next item, otherwise
- swap the two entries



Example

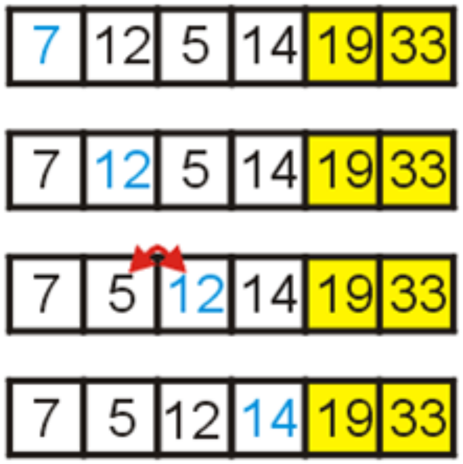
After one loop, the largest element is in the last location

- Repeat the procedure



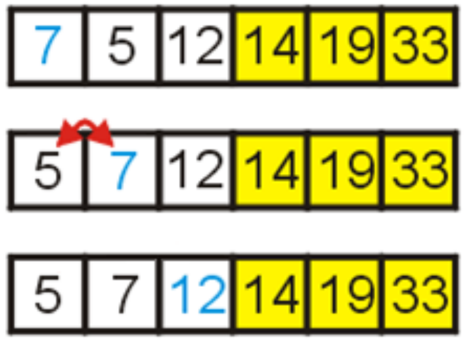
Example

- Now the two largest elements are at the end
  - Repeat again



Example

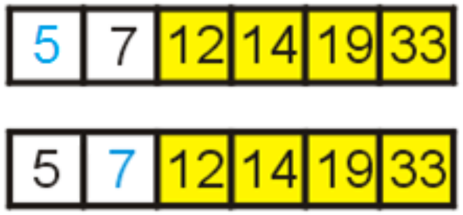
With this loop, 5 and 7 are swapped



Example

Finally, we swap the last two entries to order them

- At this point, we have a sorted array



Implementations and Improvements

The next few slides show some implementations of bubble sort together with a few improvements:

- reduce the number of swaps,
- halting if the list is sorted,
- limiting the range on which we must bubble, and
- alternating between bubbling up and sinking down

First Improvement

We could avoid so many swaps...

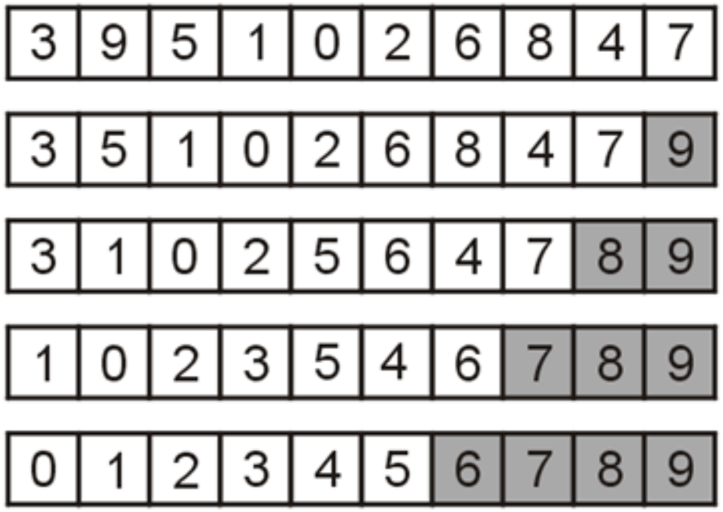
```
template <typename Type>
void bubble( Type *const array, int const n ) {
        for ( int i = n - 1; i > 0; --i ) {
                Type max = array[0];                     // assume a[0] is the max

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

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

Flagged Bubble Sort

One useful modification would be to check if no swaps occur:

- If no swaps occur, the list is sorted
- In this example, no swaps occurred during the 5th pass

Use a Boolean flag to check if no swaps occurred



Flagged Bubble Sort

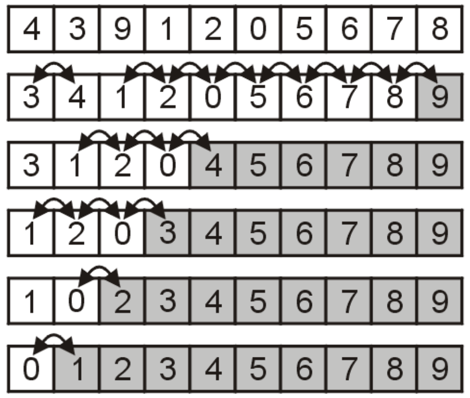
Check if the list is sorted (no swaps)

```
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;

                for ( int j = 1; j < i; ++j ) {
                        if ( array[j] < max ) {
                                array[j - 1] = array[j];
                                sorted = false;
                        } else {
                                array[j - 1] = max;
                                max = array[j];
                        }
                }

                array[i] = max;

                if ( sorted ) {
                        break;
                }
        }
}
```

Range-limiting Bubble Sort

Intuitively, one may believe that limiting the loops based on the location of the last swap may significantly speed up the algorithm

- For example, after the second pass, we are certain all entries after 4 are sorted



The implementation is easier than that for using a Boolean flag

Range-limiting Bubble Sort

Update **`i`** to at the place of the last swap

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

                    for ( int j = 1; j < i; ++j ) {
                              if ( array[j] < max ) {
                                        array[j - 1] = array[j];
                                        i = j - 1;
                              } else {
                                        array[j – 1] = max;
                                        max = array[j];
                              }
                    }

                    array[i] = max;
          }
}
```

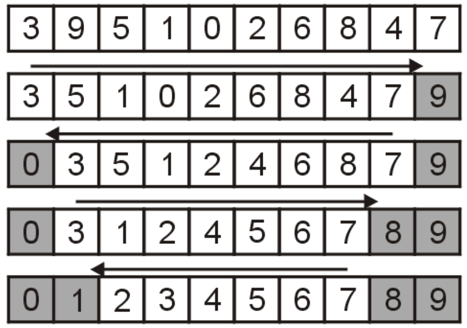
Range-limiting Bubble Sort

Unfortunately, in practice, this does little to affect the number of comparisons

Alternating Bubble Sort

One operation which does significantly improve the run time is to alternate between

- bubbling the largest entry to the top, and
- sinking the smallest entry to the bottom



This does, we will see, make a significant improvement

Alternating Bubble Sort

Alternating between bubbling and sinking:

```
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;
                }
        }
}
```

Bubble up to the back

Sink down to the front

Run-time Analysis

Because the bubble sort simply swaps adjacent entries, it cannot be any better than insertion sort which does $n + d$ comparisons where d is the number of inversions

Unfortunately, it isn't that easy:

- There are numerous unnecessary comparisons

Empirical Analysis

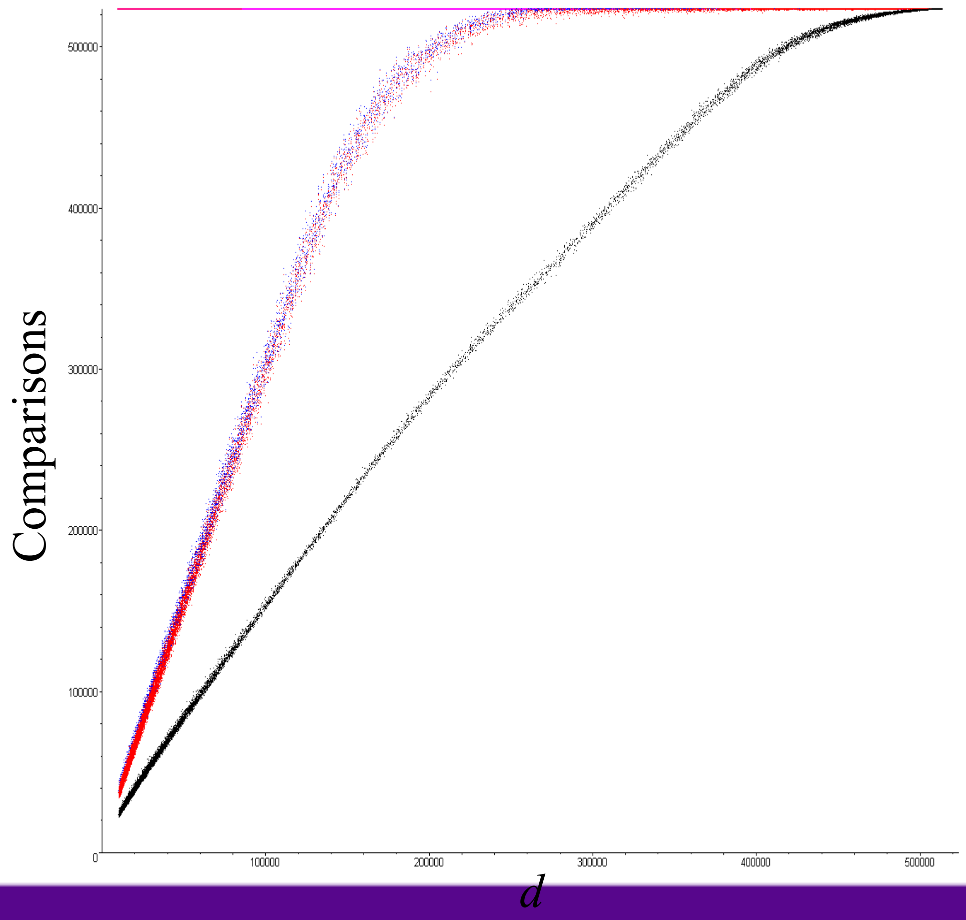
The next slide map the number of required comparisons necessary to sort 32768 arrays of size 1024 where the number of inversions range from 10000 to 523776

- Each point (d, c) is the number of inversions in an unsorted list d and the number of required comparisons c

Empirical Analysis

The following for plots show the required number of comparisons required to sort an array of size 1024

Basic implementation —
Flagged —
Range limiting —
Alternating —

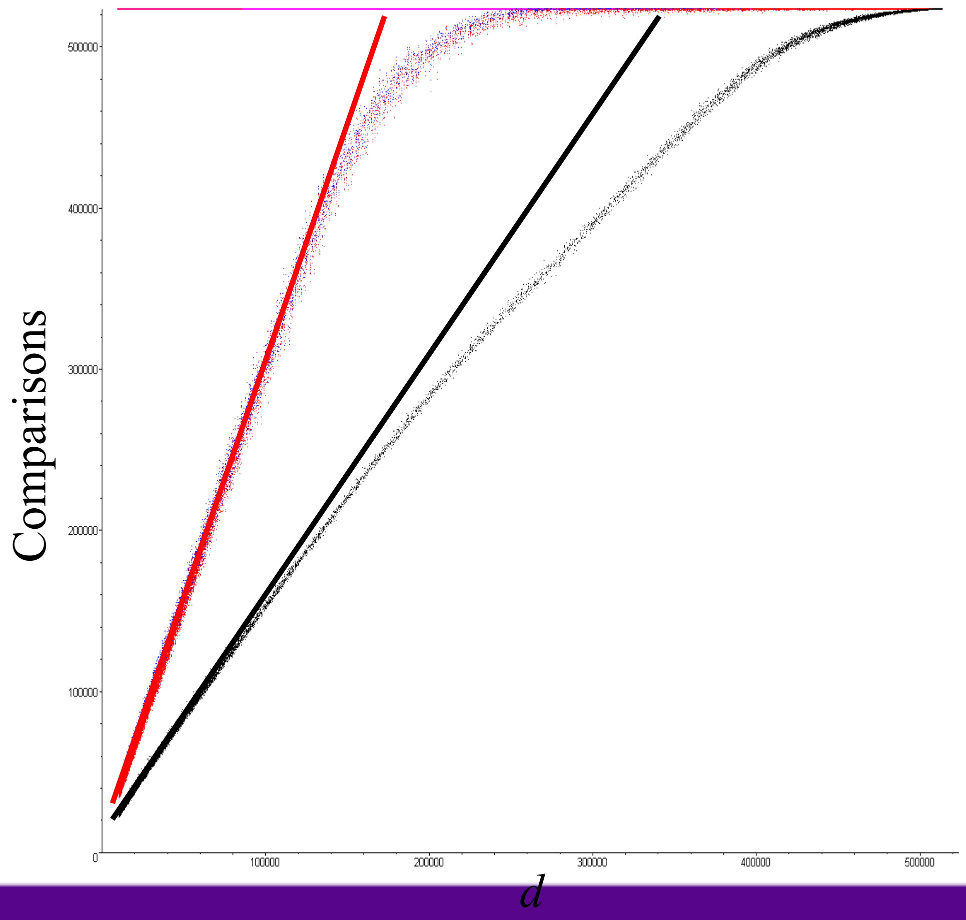


Empirical Analysis

The number of comparisons with the flagged/limiting sort is initially $n + 3d$

For the alternating variation, it is initially $n + 1.5d$

Basic implementation	🟪
Flagged	🟫
Range limiting	🟥
Alternating	⬛

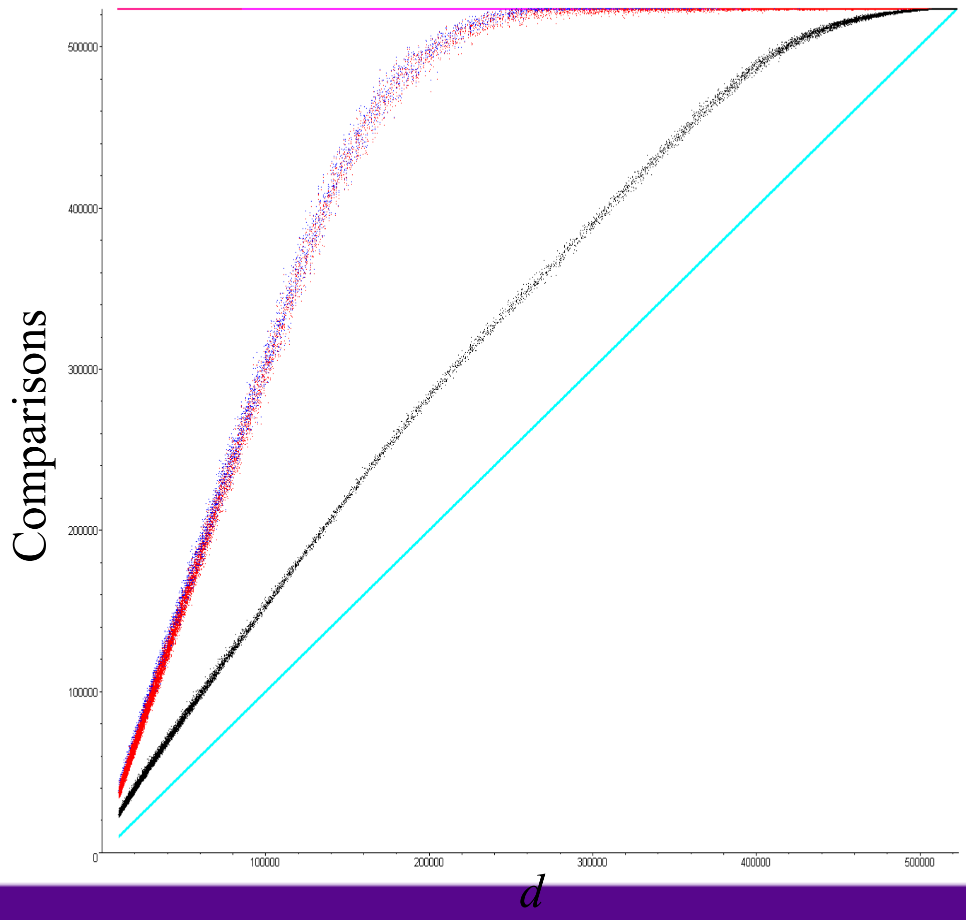


Empirical Analysis

Unfortunately, the comparisons for insertion sort is $n + d$ which is better in all cases except when the list is

- Sorted, or
- Reverse sorted

Basic implementation	▬
Flagged	▬
Range limiting	▬
Alternating	▬
Insertion Sort	▬



Run-Time

The following table summarizes the run-times of our modified bubble sorting algorithms; however, they are all worse than insertion sort in practice

Case	**Run Time**	**Comments**
Worst	$\Theta(n^2)$	$\Theta(n^2)$ inversions
Average	$\Theta(n + d)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	$d = O(n)$ inversions

Summary

In this topic, we have looked at bubble sort

From the description, it sounds as if it is as good as insertion sort

- it has the same asymptotic behaviour
- in practice, however, it is significantly worse
- it is also much more difficult to code...

References

Wikipedia, http://en.wikipedia.org/wiki/Bubble_sort

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.2, p.106-9.