CS101 Algorithms and Data Structures

Insertion and Bubble Sort

Textbook Ch 2

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort
- Bucket sort
- Radix sort

Definition

Sorting is the process of:

- Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, ..., a_{n-1})$$

e.g., numbers, and returning an reordering

$$(a'_0, a'_1, ..., a'_{n-1})$$

such that

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$$

The conversion of an Abstract List into an Abstract Sorted List

Definition

Seldom will we sort isolated values

- Usually we will sort a number of records containing a number of fields based on a *key*:

19991532	Stevenson	Monica	3 Glendridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Kilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19981932	Carol	Ann	81 Oakridge Ave.
20003287	Redpath	David	5 Glendale Ave.

Numerically by ID Number

19981932	Carol	Ann	81 Oakridge Ave.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.
20003287	Redpath	David	5 Glendale Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.

Lexicographically by surname, then given name

19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19985832	Kilji	Islam	37 Masterson Ave.
20003287	Redpath	David	5 Glendale Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.

Definition

In these topics, we will assume that:

- We are sorting integers
  - The algorithms can also be applied to other types of objects as long as we can compare any two objects
- Arrays are to be used for both input and output

In-place Sorting

Sorting algorithms may be performed *in-place*, that is, with the allocation of at most $\Theta(1)$ additional memory (*e.g.*, fixed number of local variables)

- Some definitions of *in place* as using $\mathbf{o}(n)$ memory

Other sorting algorithms require the allocation of second array of equal size

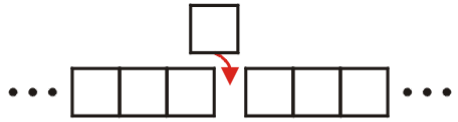
- Requires $\Theta(n)$ additional memory

We will prefer in-place sorting algorithms

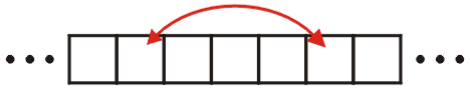
Classifications

The operations of a sorting algorithm are based on the actions performed:

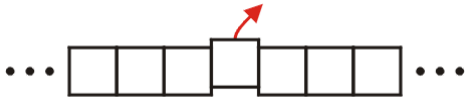
- Insertion



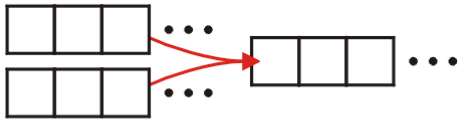
- Exchanging



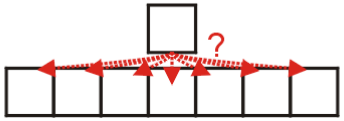
- Selection



- Merging



- Distribution



Run-time

The run time of the sorting algorithms we will look at fall into one of three categories:

$$\Theta(n) \qquad \Theta(n \ln(n)) \qquad \mathbf{O}(n^2)$$

We will examine average- and worst-case scenarios for each algorithm

The run-time may change significantly based on the scenario

Run-time

We will review the more traditional $\mathbf{O}(n^2)$ sorting algorithms:

- Insertion sort, Bubble sort

Some of the faster $\Theta(n \ln(n))$ sorting algorithms:

- Heap sort, Quicksort, and Merge sort

And linear-time sorting algorithms

- Bucket sort and Radix sort
- We must make assumptions about the data

Lower-bound Run-time

Any sorting algorithm must examine each entry in the array at least once

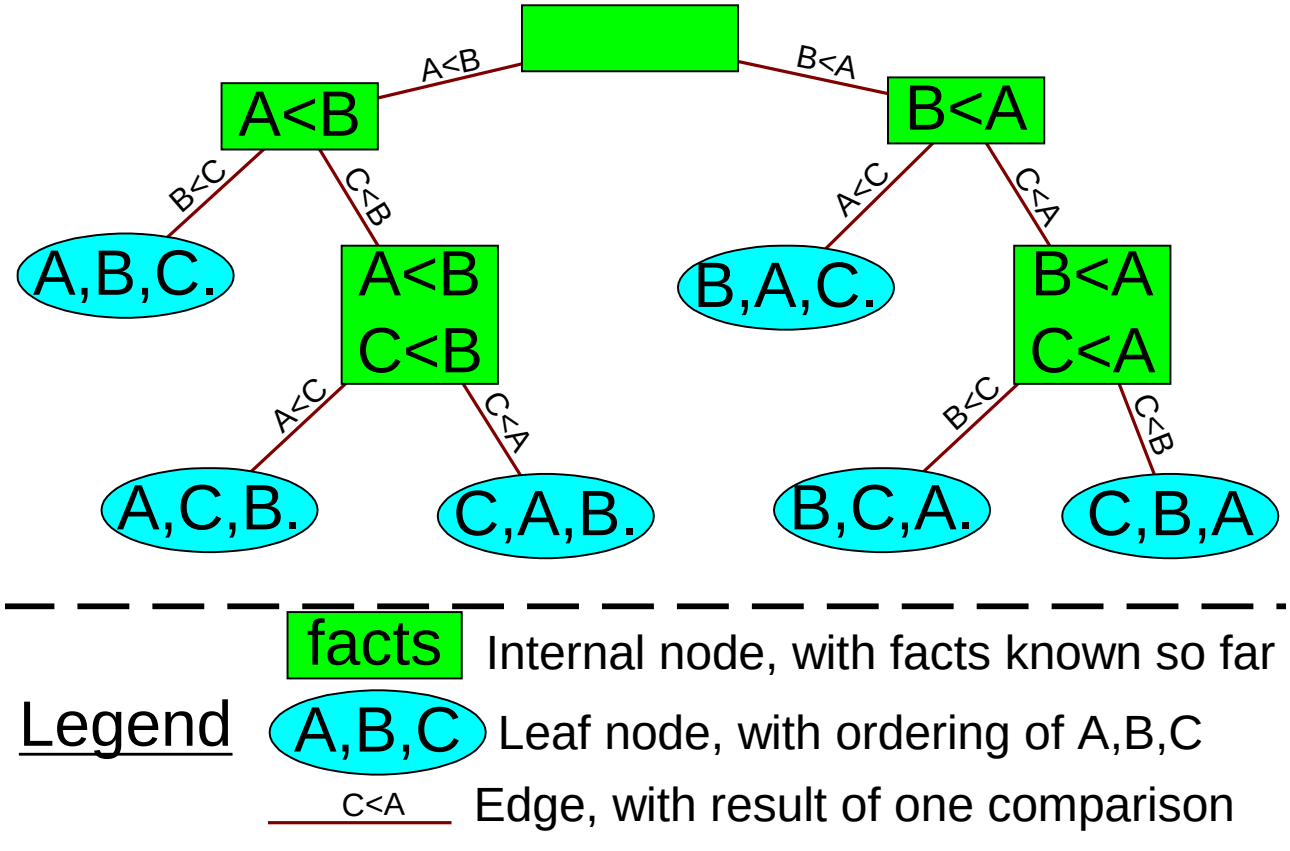
- Consequently, all sorting algorithms must be $\Omega(n)$

We will not be able to achieve $\Theta(n)$ behaviour without additional assumptions

Lower-bound Run-time

The general run time is $\Omega(n \ln(n))$

The proof is based on the idea of a *comparison tree*



Optimal Sorting Algorithms

We will cover some common sorting algorithms

- There is no *optimal* sorting algorithm which can be used in all places
- Under various circumstances, different sorting algorithms will deliver optimal run-time and memory-allocation requirements

Sub-optimal Sorting Algorithms

Before we look at other algorithms, we will consider the Bogosort algorithm:

1. Randomly order the objects, and
2. Check if they're sorted, if not, go back to Step 1.

Run time analysis:

- best case: $\Theta(n)$
- worst: unbounded
- average: $\Theta((n+1)!)$

Summary

Introduction to sorting, including:

- Assumptions
- In-place sorting ($\mathbf{O}(1)$ additional memory)
- Sorting techniques
  - insertion, exchanging, selection, merging, distribution
- Run-time classification: $\mathbf{O}(n)$ $\mathbf{O}(n \ln(n))$ $\mathbf{O}(n^2)$

Overview of proof that a general sorting algorithm must be $\mathbf{\Omega}(n \ln(n))$

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort

Inversions

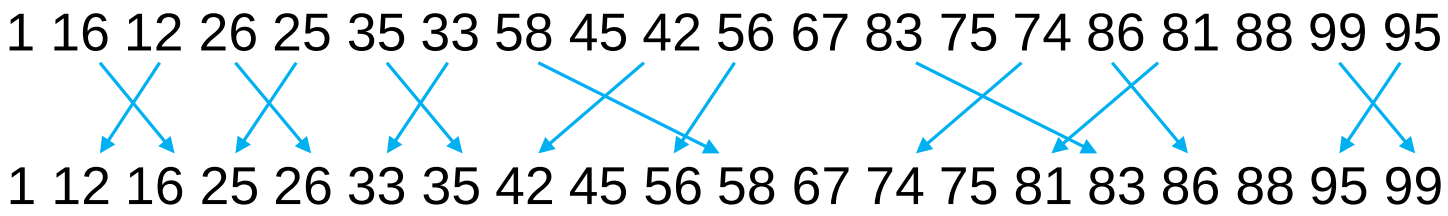
Consider the following three lists:

```
 1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

 1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80  1 31 16 92
```

To what degree are these three lists unsorted?

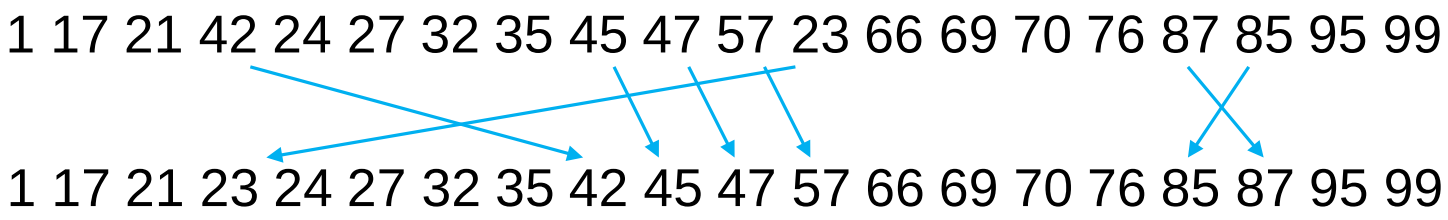
Inversions

The first list requires only a few exchanges to make it sorted



Inversions

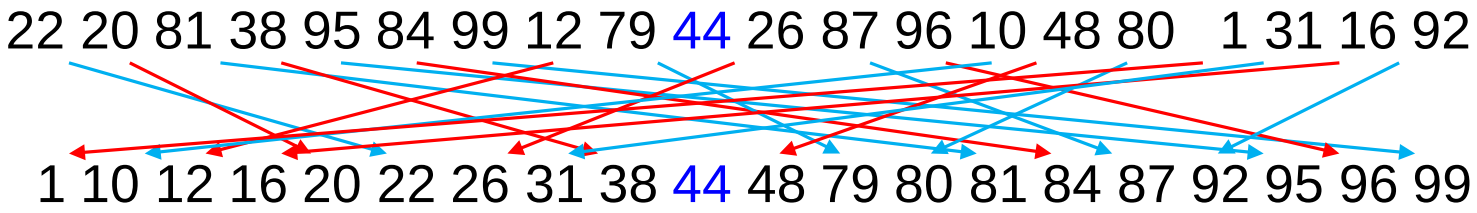
The second list has two entries significantly out of order



however, most entries (13) are in place

Inversions

The third list would, by any reasonable definition, be significantly unsorted



Inversions

Given any list of n numbers, there are

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

pairs of numbers

For example, the list (1, 3, 5, 4, 2, 6) contains the following 15 pairs:

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	(3, 2)	(3, 6)
		(5, 4)	(5, 2)	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

Inversions

You may note that 11 of these pairs of numbers are in order:

(1, 3)	**(1, 5)**	**(1, 4)**	**(1, 2)**	**(1, 6)**
	(3, 5)	**(3, 4)**	(3, 2)	**(3, 6)**
		(5, 4)	(5, 2)	**(5, 6)**
			(4, 2)	**(4, 6)**
				(2, 6)

Inversions

The remaining four pairs are *reversed*, or *inverted*

(1, 3)	(1, 5)	(1, 4)	(1, 2)	(1, 6)
	(3, 5)	(3, 4)	**(3, 2)**	(3, 6)
		(5, 4)	**(5, 2)**	(5, 6)
			(4, 2)	(4, 6)
				(2, 6)

Inversions

Given a permutation of n elements

$$a_0, a_1, ..., a_{n-1}$$

an inversion is defined as a pair of entries which are reversed

That is, (a_j, a_k) forms an inversion if

$$j < k \text{ but } a_j > a_k$$

Inversions

Therefore, the permutation

1, 3, 5, 4, 2, 6

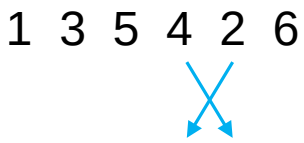
contains four inversions:

(3, 2) (5, 4) (5, 2) (4, 2)

Inversions

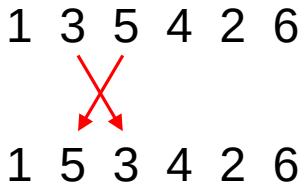
Exchanging (or swapping) two adjacent entries either:

– removes an inversion, *e.g.*,



removes the inversion (4, 2)

– or introduces a new inversion, *e.g.*, (5, 3) with



Number of Inversions

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of numbers in any set of n objects

Consequently, each pair contributes to

- the set of ordered pairs, or
- the set of inversions

For a random ordering, we would expect approximately half of all pairs are inversions:

$$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} = \mathbf{O}(n^2)$$

Number of Inversions

For example, the following unsorted list of 56 entries

61	548	3	923	195	973	289	237	57	299	594	928	515	55
507	351	262	797	788	442	97	798	227	127	474	825	7	182
929	852	504	485	45	98	538	476	175	374	523	800	19	901
349	947	613	265	844	811	636	859	81	270	697	563	976	539

has 655 inversions and 885 ordered pairs

The formula predicts $\frac{1}{2}\binom{56}{2} = \frac{56(56-1)}{4} = 770$ inversions

Number of Inversions

Let us consider the number of inversions in our first three lists:

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

22 20 81 38 95 84 99 12 79 44 26 87 96 10 48 80 1 31 16 92

Each list has 20 entries, and therefore:

- There are $\binom{20}{2} = \frac{20(20-1)}{2} = 190$ pairs
- On average, $190/2 = 95$ pairs would form inversions

Number of Inversions

The first list

1 16 12 26 25 35 33 58 45 42 56 67 83 75 74 86 81 88 99 95

has 13 inversions:

(16, 12) (26, 25) (35, 33) (58, 45) (58, 42) (58, 56) (45, 42)
(83, 75) (83, 74) (83, 81) (75, 74) (86, 81) (99, 95)

This is well below 95, the expected number of inversions

- Therefore, this is likely not to be a *random* list

Number of Inversions

The second list

1 17 21 42 24 27 32 35 45 47 57 23 66 69 70 76 87 85 95 99

also has 13 inversions:

(42, 24) (42, 27) (42, 32) (42, 35) (42, 23) (24, 23) (27, 23)
(32, 23) (35, 23) (45, 23) (47, 23) (57, 23) (87, 85)

This, too, is not a random list

Number of Inversions

The third list

22 20 81 38 95 84 **99** 12 79 44 26 87 96 10 48 80 **1** 31 16 92

has 100 inversions:

(22, 20) (22, 12) (22, 10) (22, **1**) (22, 16) (20, 12) (20, 10) (20, **1**) (20, 16) (81, 38)
(81, 12) (81, 79) (81, 44) (81, 26) (81, 10) (81, 48) (81, 80) (81, **1**) (81, 16) (81, 31)
(38, 12) (38, 26) (38, 10) (38, **1**) (38, 16) (38, 31) (95, 84) (95, 12) (95, 79) (95, 44)
(95, 26) (95, 87) (95, 10) (95, 48) (95, 80) (95, **1**) (95, 16) (95, 31) (95, 92) (84, 12)
(84, 79) (84, 44) (84, 26) (84, 10) (84, 48) (84, 80) (84, **1**) (84, 16) (84, 31) (**99**, 12)
(**99**, 79) (**99**, 44) (**99**, 26) (**99**, 87) (**99**, 96) (**99**, 10) (**99**, 48) (**99**, 80) (**99**, **1**) (**99**, 16)
(**99**, 31) (**99**, 92) (12, 10) (12, **1**) (79, 44) (79, 26) (79, 10) (79, 48) (79, **1**) (79, 16)
(79, 31) (44, 26) (44, 10) (44, **1**) (44, 16) (44, 31) (26, 10) (26, **1**) (26, 16) (87, 10)
(87, 48) (87, 80) (87, **1**) (87, 16) (87, 31) (96, 10) (96, 48) (96, 80) (96, **1**) (96, 16)
(96, 31) (96, 92) (10, **1**) (48, **1**) (48, 16) (48, 31) (80, **1**) (80, 16) (80, 31) (31, 16)

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort

Outline

This topic discusses the insertion sort

We will discuss:

- the algorithm
- an example
- run-time analysis
  - worst case
  - average case
  - best case

Background

Consider the following observations:

- A list with one element is sorted
- In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size $k + 1$

Background

For example, consider this sorted array containing of eight sorted entries

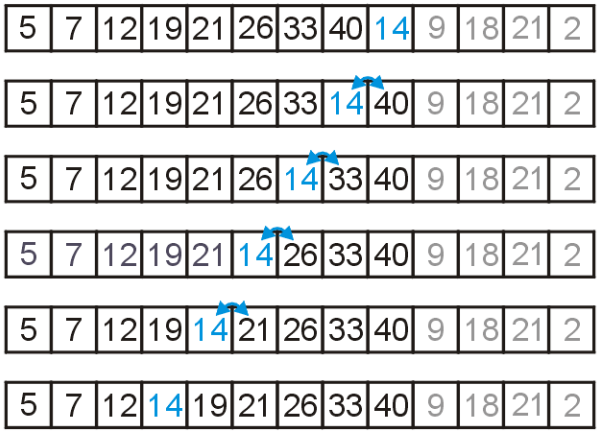
5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	**14**	9	18	21	2

Suppose we want to insert 14 into this array leaving the resulting array sorted

Background

Starting at the back, if the number is greater than 14, copy it to the right

- Once an entry less than 14 is found, insert 14 into the resulting vacancy



The Algorithm

For any unsorted list:

- Treat the first element as a sorted list of size 1

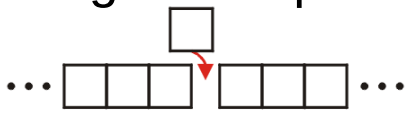
Then, given a sorted list of size $k-1$

- Insert the k^{th} item into the sorted list
- The sorted list is now of size k

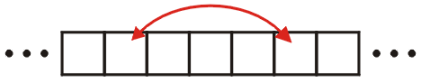
The Algorithm

Recall the five sorting techniques:

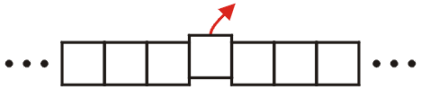
- Insertion



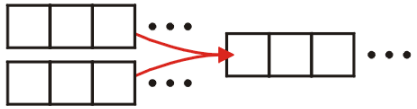
- Exchange



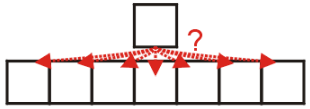
- Selection



- Merging



- Distribution

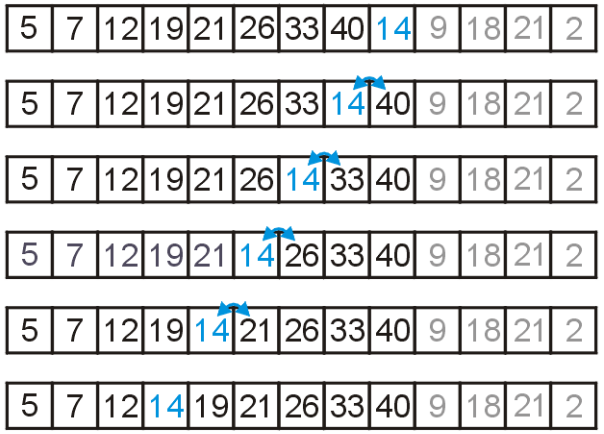


Clearly insertion falls into the first category

The Algorithm

Code for this would be:

```
for ( int j = k; j > 0; --j ) {
    if ( array[j - 1] > array[j] ) {
        std::swap( array[j - 1], array[j] );
    } else {
        // As soon as we don't need to swap, the (k + 1)st
        // is in the correct location
        break;
    }
}
```



Implementation and Analysis

This would be embedded in a function call such as

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
     for ( int k = 1; k < n; ++k ) {
         for ( int j = k; j > 0; --j ) {
                 if ( array[j - 1] > array[j] ) {
                         std::swap( array[j - 1], array[j] );
                 } else {
                         // As soon as we don't need to swap, the (k + 1)st
                         // is in the correct location
                         break;
                 }
         }
     }
}
```

Implementation and Analysis

Let's do a run-time analysis of this code

- the outer for-loop will be executed a total of $n - 1$ times
- In the worst case, the inner for-loop is executed k times
- Thus, the worst-case run time is $O(n^2)$

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
                if ( array[j - 1] > array[j] ) {
                        std::swap( array[j - 1], array[j] );
                } else {
                        // As soon as we don't need to swap, the (k + 1)st
                        // is in the correct location
                        break;
                }
        }
    }
}
```

Implementation and Analysis

Problem: we may break out of the inner loop…

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

Recall: each time we perform a swap, we remove an inversion

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

Thus, the body is run only as often as there are inversions

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

If the number of inversions is d, the run time is $\Theta(n + d)$

Consequences of Our Analysis

The average random list has $d = \Theta(n^2)$ inversions

Insertion sort, however, will run in $\Theta(n)$ time whenever $d = O(n)$

Other benefits:

- The algorithm is easy to implement
- Even in the worst case, the algorithm is fast for small problems

Size	Approximate Time (ns)
8	175
16	750
32	2700
64	8000

Consequences of Our Analysis

Unfortunately, it is not very useful in general:

- Sorting a random list of size $2^{23} \approx 8\,000\,000$ would require approximately one day
- Doubling the size of the list quadruples the required run time
- An optimized quick sort requires less than $4\ \mathrm{s}$ on a list of the above size

Consequences of Our Analysis

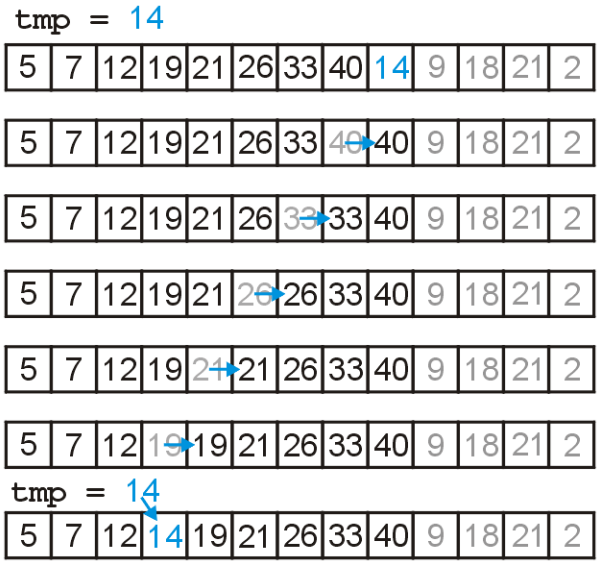
The following table summarizes the run-times of insertion sort

Case	Run Time	Comments
Worst	$\Theta(n^2)$	Reverse sorted
Average	$O(d + n)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Very few inversions: $d = O(n)$

A small improvement

Swapping is expensive, so we could just temporarily assign the new entry

- this reduces assignments by a factor of 3



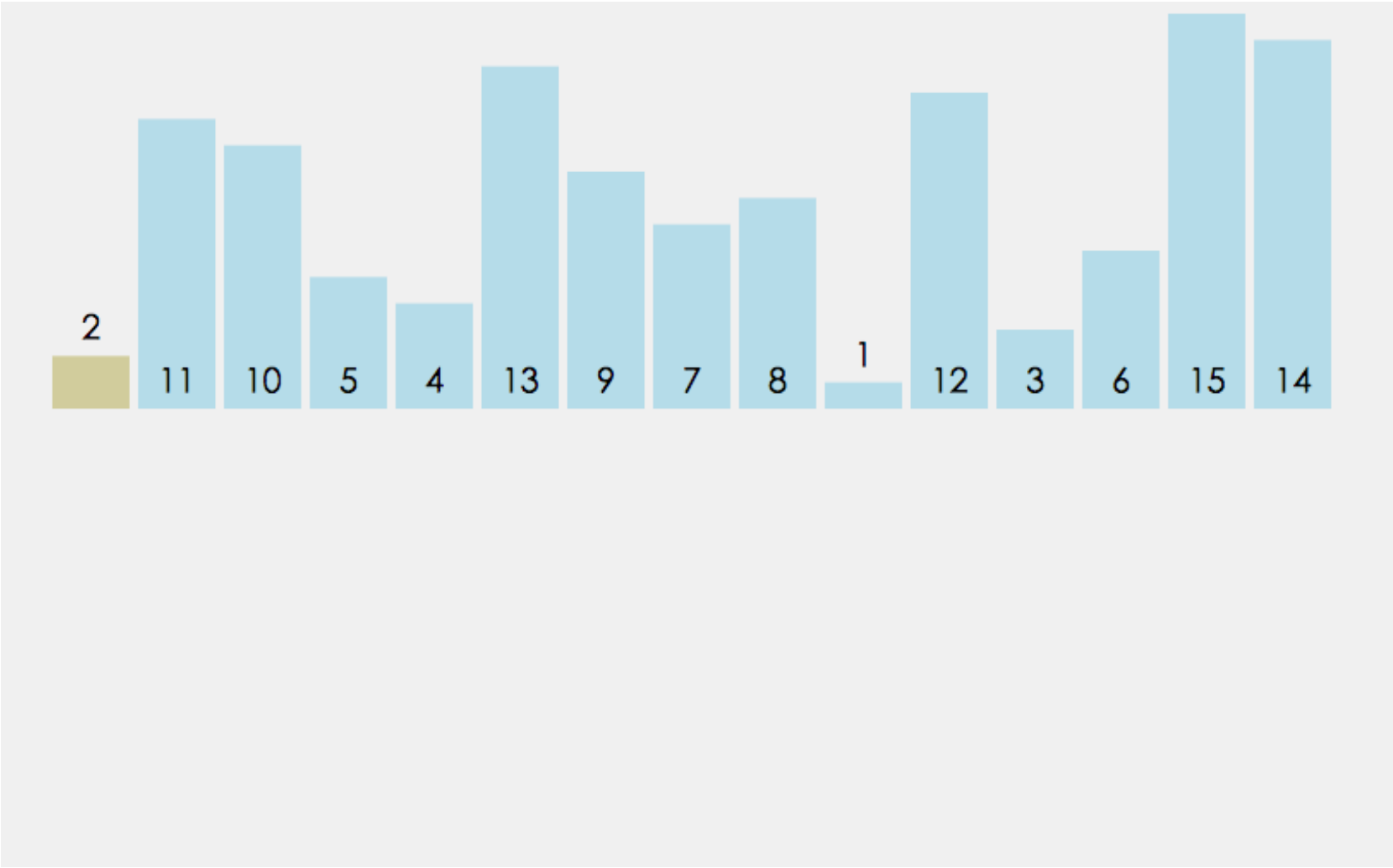
Implementation and Analysis

A reasonably good* implementation

```
template <typename Type>
void insertion( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        Type tmp = array[k];

        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                 array[j] = tmp;
                 goto finished;
            }
        }

        array[0] = tmp;  // only executed if tmp < array[0]

        finished: ;  // empty statement
    }
}
```

* we could do better with pointers

A small improvement



Summary

Insertion Sort:

- Insert new entries into growing sorted lists
- Run-time analysis
  - Actual and average case run time: $\mathbf{O}(n^2)$
  - Detailed analysis: $\Theta(n + d)$
  - Best case ($\mathbf{O}(n)$ inversions): $\Theta(n)$
- Memory requirements: $\Theta(1)$

Outline

- Introduction
- Inversions
- Insertion sort
- Bubble sort

Outline

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

- Uses an opposite strategy from insertion sort

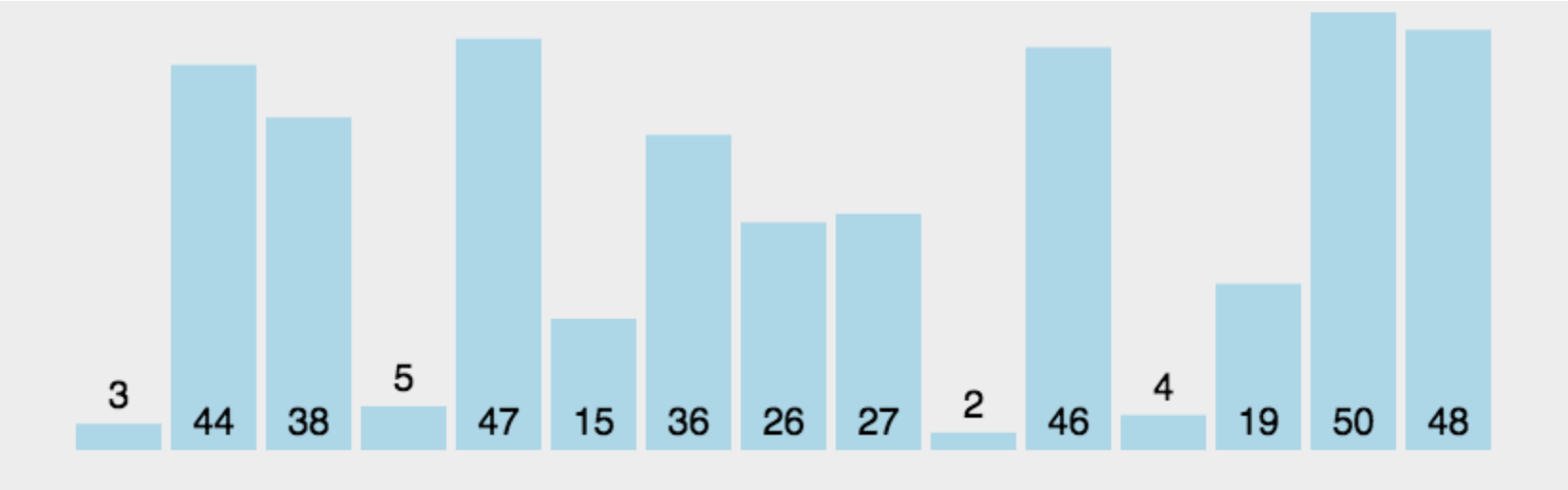
We will examine:

- 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

The basic algorithm

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

The basic algorithm

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

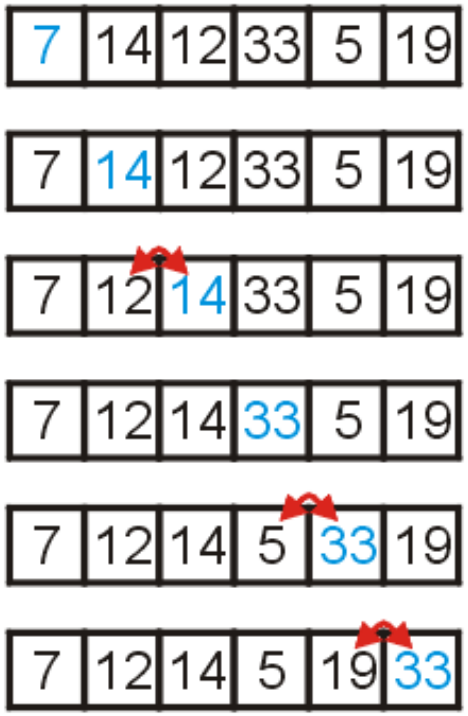
Repeat $n - 1$ times, after which, all entries will be in place

Example

Consider the unsorted array to the right

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

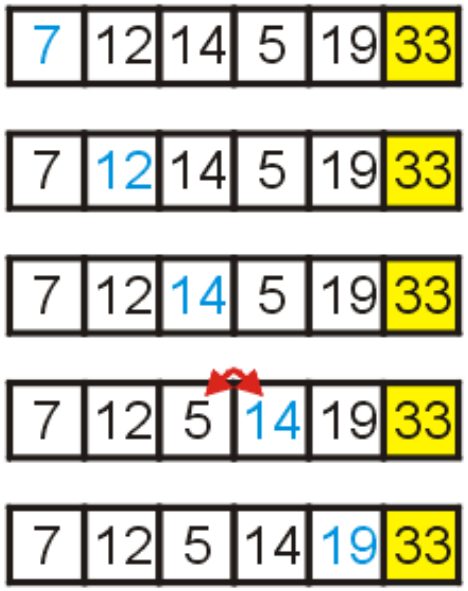
- if the current and next items are in order, continue with the next item, otherwise
- swap the two entries



Example

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

- Repeat the procedure



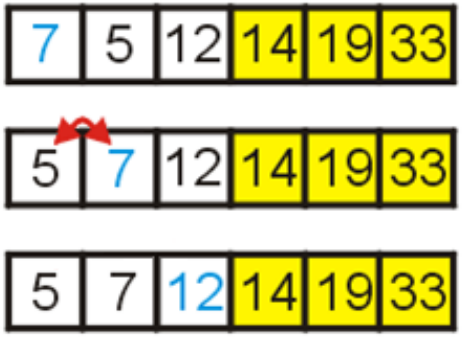
Example

Now the two largest elements are at the end

- Repeat again

Example

With this loop, 5 and 7 are swapped



Example

Finally, we check the last two entries

- At this point, we have a sorted array

The basic algorithm

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] );
                        }
                }
        }
}
```

Analysis

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

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

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
- alternating between bubbling up and sinking down

First Improvement

We could avoid so many swaps...

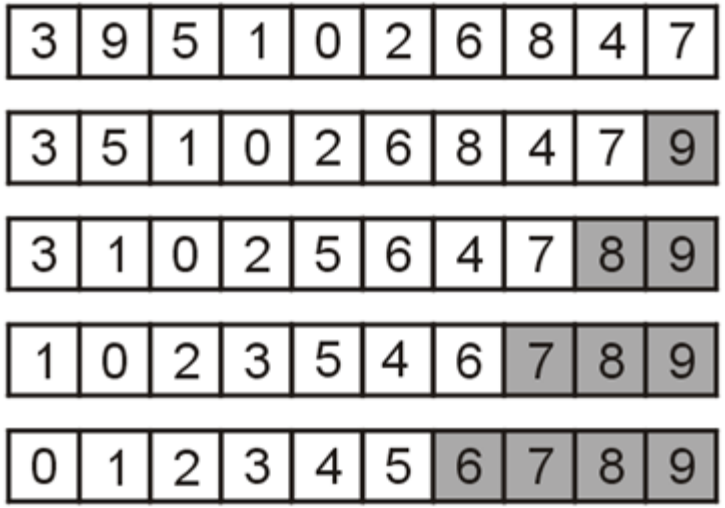
```
template <typename Type>
void bubble( Type *const array, int const n ) {
  for ( int i = n - 1; i > 0; --i ) {
        Type max = array[0];                    // assume a[0] is the max
        for ( int j = 1; j <= i; ++j ) {
              if ( array[j] < max ) {
                    array[j - 1]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

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

Range-limiting Bubble Sort

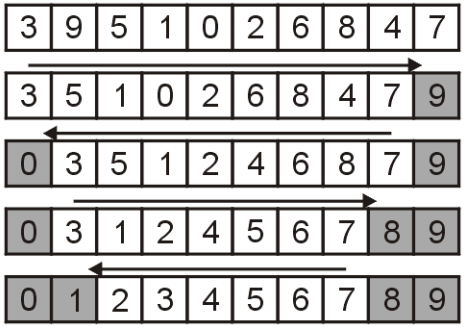
Update **i** to at the place of the last swap

```
template <typename Type>
void bubble( Type *const array, int const n ) {
  for ( int i = n - 1; i > 0; ) {
        Type max = array[0];
        int ii = 0;
        for ( int j = 1; j <= i; ++j ) {
              if ( array[j] < max ) {
                    array[j - 1] = array[j];
                    ii = j - 1;
              } else {
                    array[j – 1] = max;
                    max = array[j];
              }
        }
        array[i] = max;
        i = ii;
  }
}
```

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



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, run-time analysis isn't that easy:

- There are numerous unnecessary comparisons

Empirical Analysis

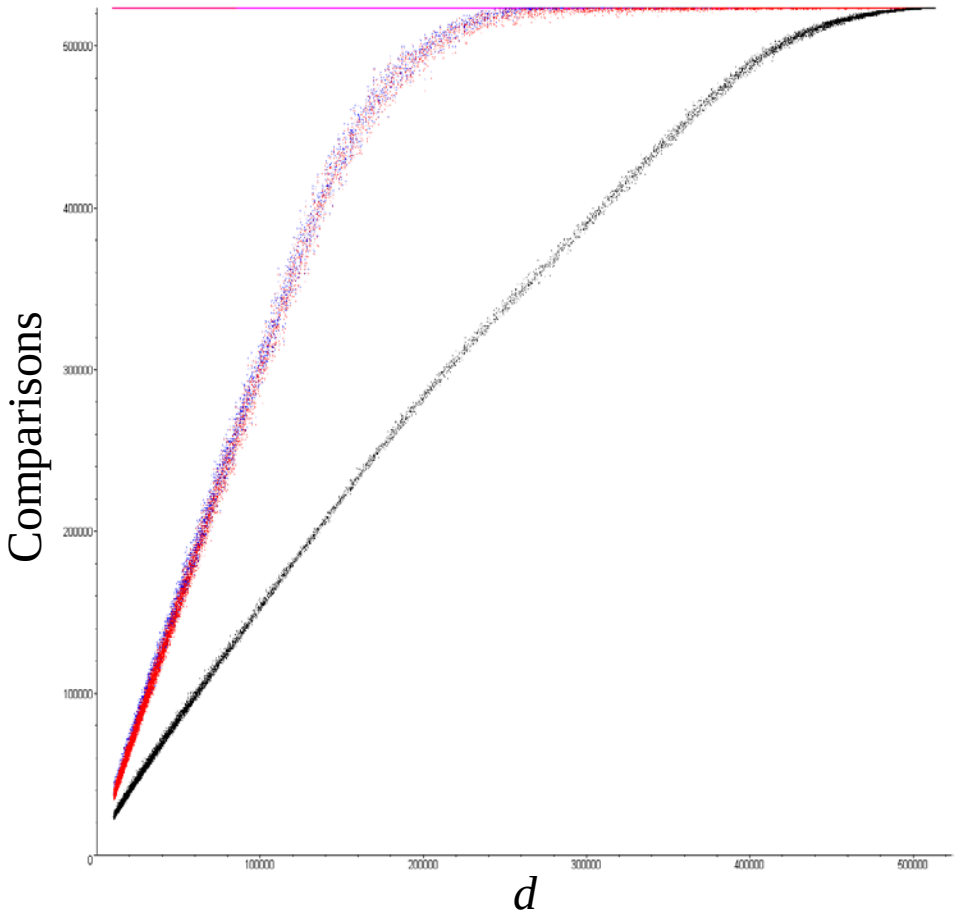
The next slide map the number of required comparisons necessary to sort 32768 arrays of size 1024 where the number of inversions range from 10000 to 523776

- 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

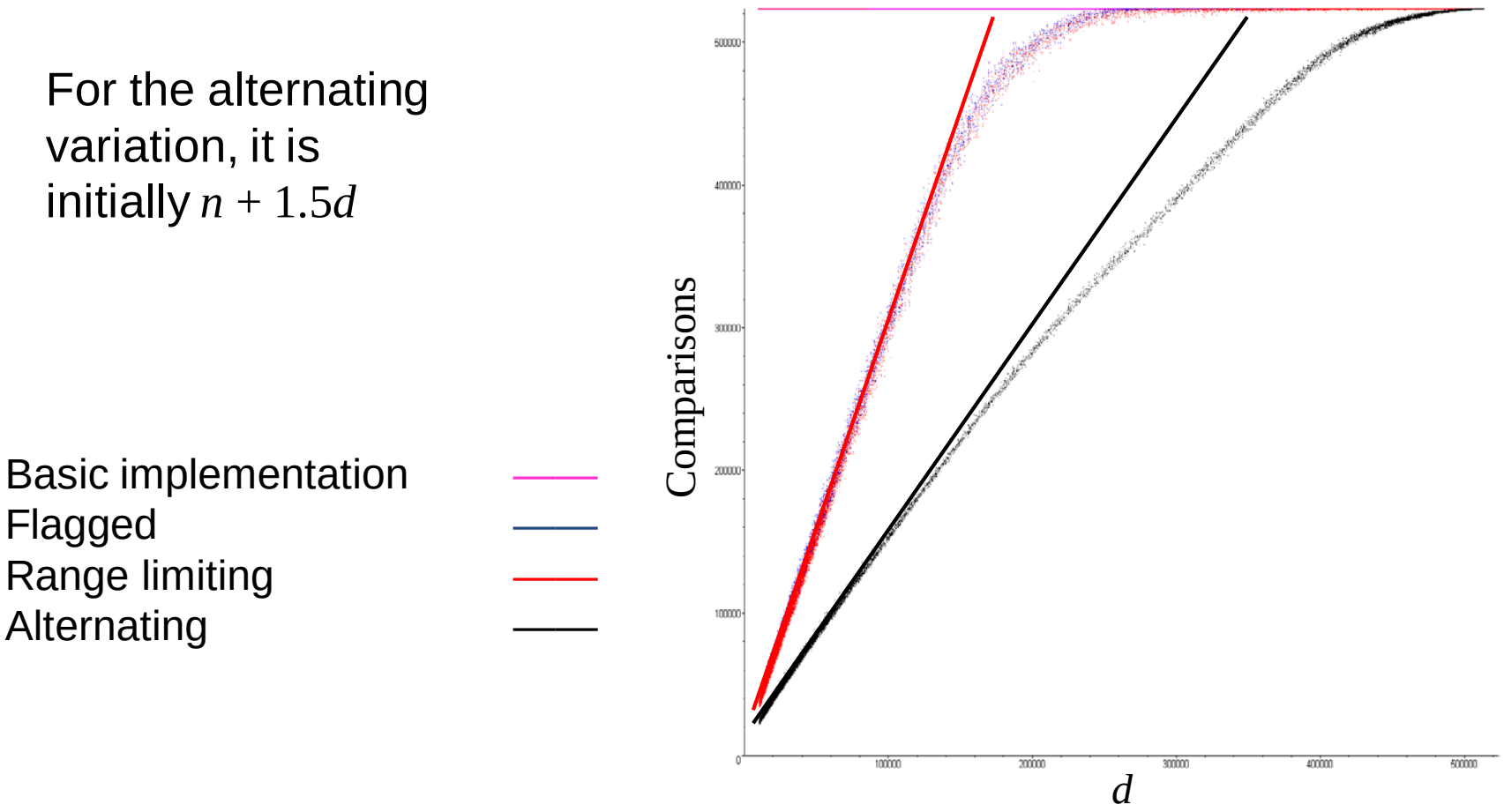
Basic implementation
Flagged
Range limiting
Alternating



Empirical Analysis

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

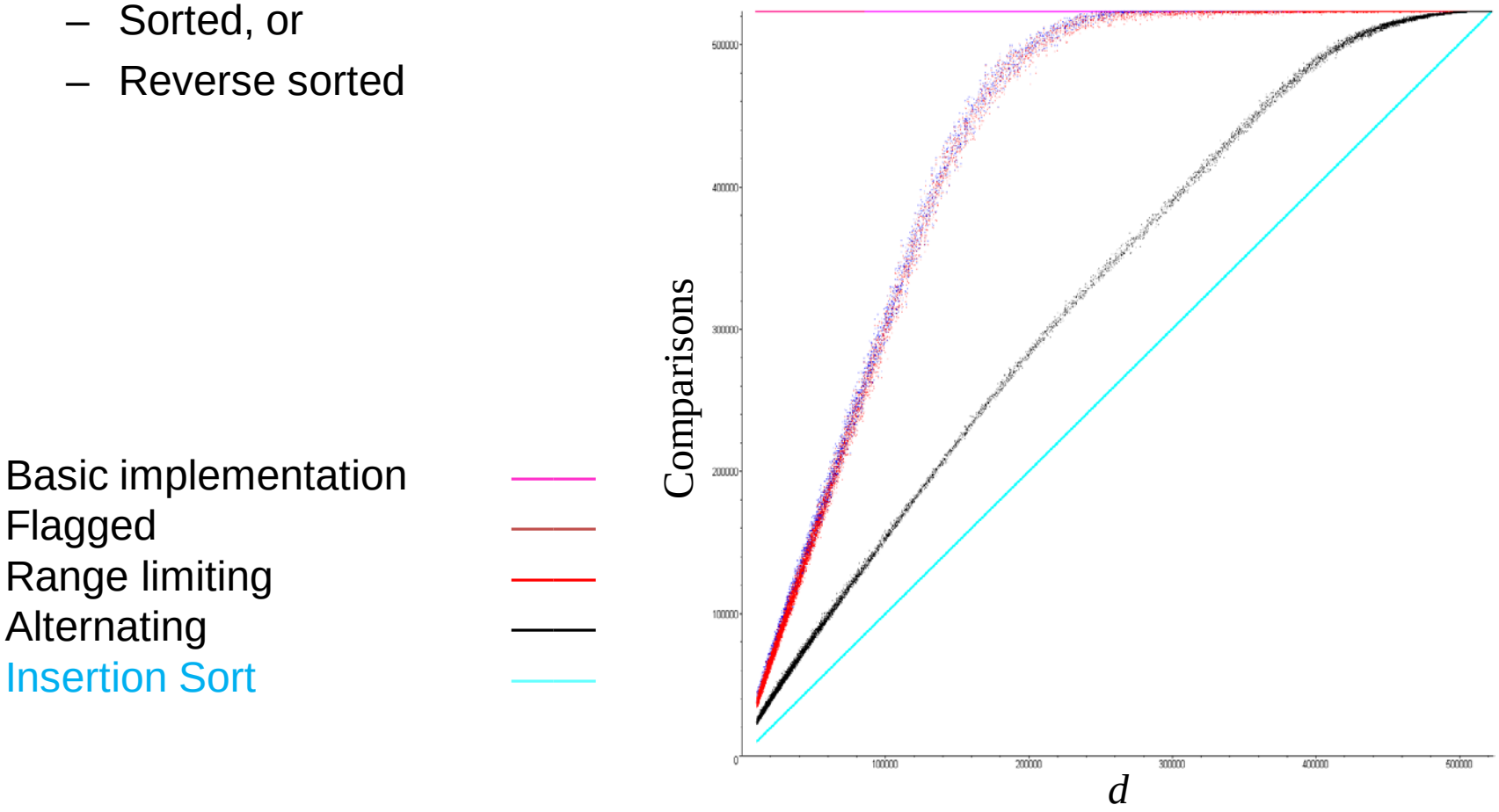
For the alternating variation, it is initially $n + 1.5d$



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



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...