SORTING

ELEMENTARY SORTS

Ruben Acuña

Fall 2018

FOUNDATIONS

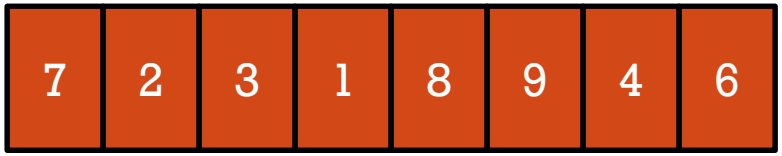
APPLICATIONS

- Sorting is useful for:
  - Ordering strings alphanumerically(e.g., contacts, filenames, titles, grades, vote count, etc.).
  - Ordering objects on a screen (what should be drawn when).
  - Ordering executing processes (what should be given priority?).
  - What else?

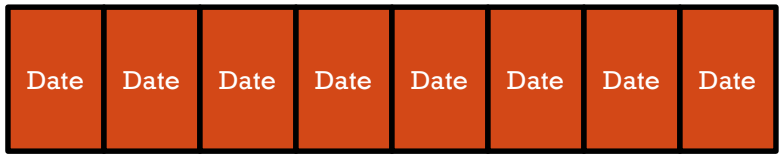
  - Sorting provides:
    - Structure! We know some useful (?!) property about the values.
      - Remember binary search?
      - Useful when doing alignments/mapping.
    - A decomposition of a list of values. Natural sections, or clusters, of data can be formed by picking some interval.

AN "ORDERED" VIEW

- Data may be simple:



- Or complex:



- Typically, we think about the datatype in an array as providing a ***key***: some subset of data that is sufficient to define an order.
- Really, it would be best to have some generic way to view the order of elements…
- For now let us assume we have a method *v.less(w)* that encapsulates comparisons and returns true if $v < w$ in the order defined for the domain of *v* and *w*.

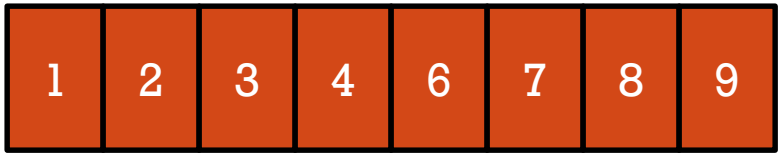
APPROACHES

- There are many algorithms to sort: insertion sort, selection sort, mergesort, quick sort, shellsort, radix sort, counting sort, bogosort, and serval dozen more.
- Sorting is not only a common problem but a classic area of algorithm analysis. It is easy to understand and can be approached in many ways.
- Some of these make different assumptions: radix sort requires integers.
- Many of these algorithms behave fundamentally different, leading to differences in performance in either worse case (e.g, $O(n^2)$ vs $O(nlogn)$, or best case (e.g., $\Omega(n^2)$, vs $\Omega(n)$).

Key point: there are many ways to solve any problem.

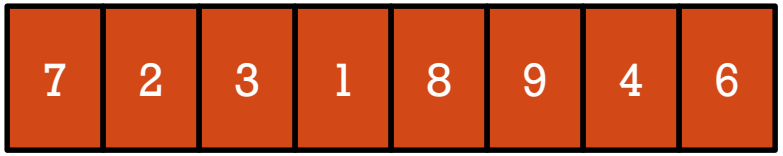
SORTED DATA

- What does it mean to be sorted anyway?



- A list is sorted if all elements are in order: $\forall n\ , a_n < a_{n+1}$, where order is computed from a key.
- Wait, is that all?

SORTED DATA



- A list is unsorted if an element is out of order: $\exists n\ s.t.\ a_{n+1} < a_n$.
- A metric: *inversions*.
- "An inversion is a pair of entries that are out of order in the array."
- Example: 3,1,8,2 has 3 inversions: 3-1,3-2,8-2.
  - Notice: the number of inversions with a particular key is proportional to disorder it adds to the data.
- Algorithms that perform "local" changes, e.g., insertion sort, fix one inversion at a time while algorithms that can perform "global" changes, e.g., selection sort, fixes multiple inversions.

JAVA CONTEXT

A BASIC FRAMEWORK

Book gives a basic framework for implementing sorting:

- Helper methods: *less* and *exch* provide common functionality.
- Testing methods: *isSorted*, *show*.
- Designed to be generic – uses *Comparable* to support more than just Strings.

```
public class Sort {
    public static void sort(Comparable[] a) {
        //
    }
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }
    private static void exch(Comparable[] a, int i, int j) {
        Comparable t = a[i]; a[i] = a[j]; a[j] = t;
    }
    private static void show(Comparable[] a) {
        for(int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
    public static boolean isSorted(Comparable[] a) {
        for (int i = 1; i < a.length; i++)
            if (less(a[i], a[i-1]))
                return false;
        return true;
    }
    public static void main(String[] args) {
        String[] a = {"S", "O", "R", "T", "E", "X", ...
        sort(a);
        assert isSorted(a);
        show(a);
    }
}
```

COMPARABLE OVERVIEW

- Trivially we can compare elements using <, >, =. However, we need to extend this to ADTs.

- Java supports this with the Comparable interface. This interface defines a method called compareTo, that can be used to compare the caller and a parameter:
  - return -1: v less than w
  - return 0: v and w equal
  - return 1: v greater than w

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Safety: should throw exception if wrong types are used. May also throw exception if parameters are null.

COMPARABLE EXAMPLE

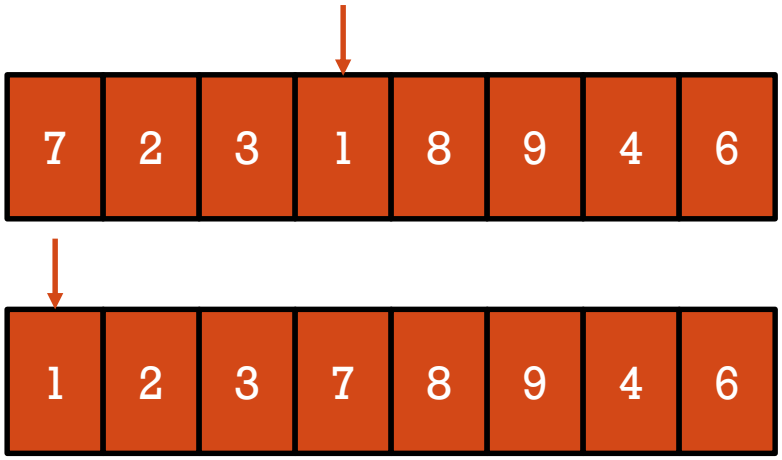
```
public class Date implements Comparable<Date> {
    private final int month, day, year;

    public Date(int m, int d, int y) {
      month = m;
      day   = d;
      year  = y;
   }

    @Override
    public int compareTo(Date that) {
        if (that.getClass() != this.getClass())
          throw new ClassCastException();

      if (this.year  < that.year ) return -1;
      if (this.year  > that.year ) return +1;
      if (this.month < that.month) return -1;
      if (this.month > that.month) return +1;
      if (this.day   < that.day  ) return -1;
      if (this.day   > that.day  ) return +1;
      return 0;
   }
}
```

SELECTION SORT

THE CONCEPT

- Basic Idea:
  - Find the smallest element in the data, starting at position i.
  - Swap it with the position i.
  - Repeat for all elements in the list.

- Consequently, a region is formed at the front of the list that is not only sorted but in final order.



Right image from Sedgewick and Wayne.

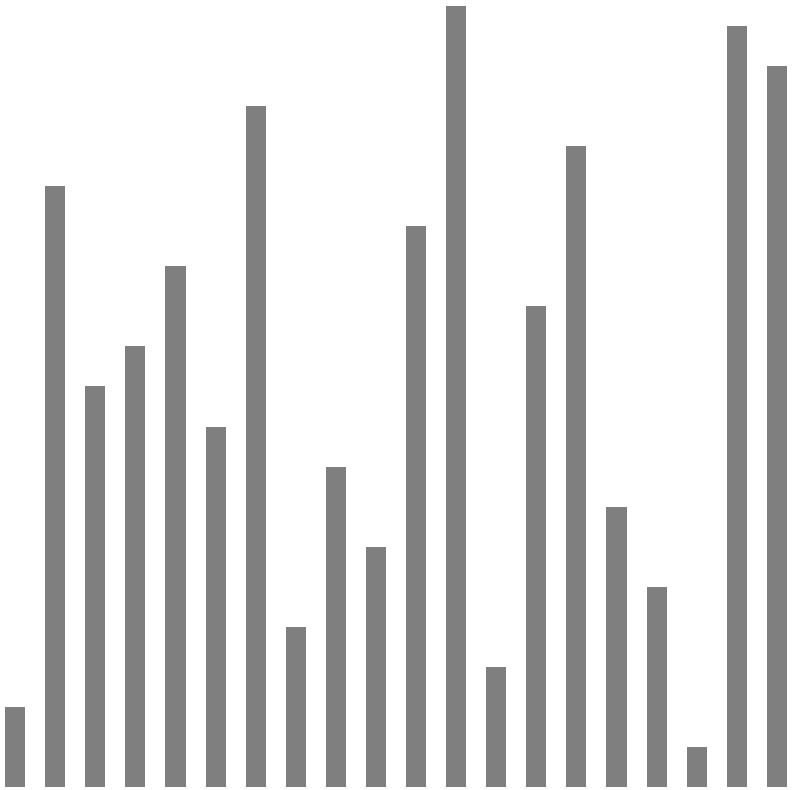
ALGORITHM TRACE

- Consider the following array: 7, 23, 25, 13, 2, 12, 3, 16, 43. Show a trace of execution for selection sort. The trace should include the initial state of the array, followed by the array's state after each swap is made.

IMPLEMENTATION

```
//Sedgewick and Wayne
public static void sort(Comparable[] a) {
    int N = a.length;

    for (int i = 0; i < N; i++)
    {
        // Exchange a[i] with smallest entry in a[i+1...N).
        int min = i; // index of minimal entry.
        for (int j = i+1; j < N; j++)
            if (less(a[j], a[min])) min = j;
        exch(a, i, min);
    }
}

//helper
private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}

//helper
private static void exch(Comparable[] a, int i, int j) {
    Comparable t = a[i]; a[i] = a[j]; a[j] = t;
}
```

RANDOM INPUT



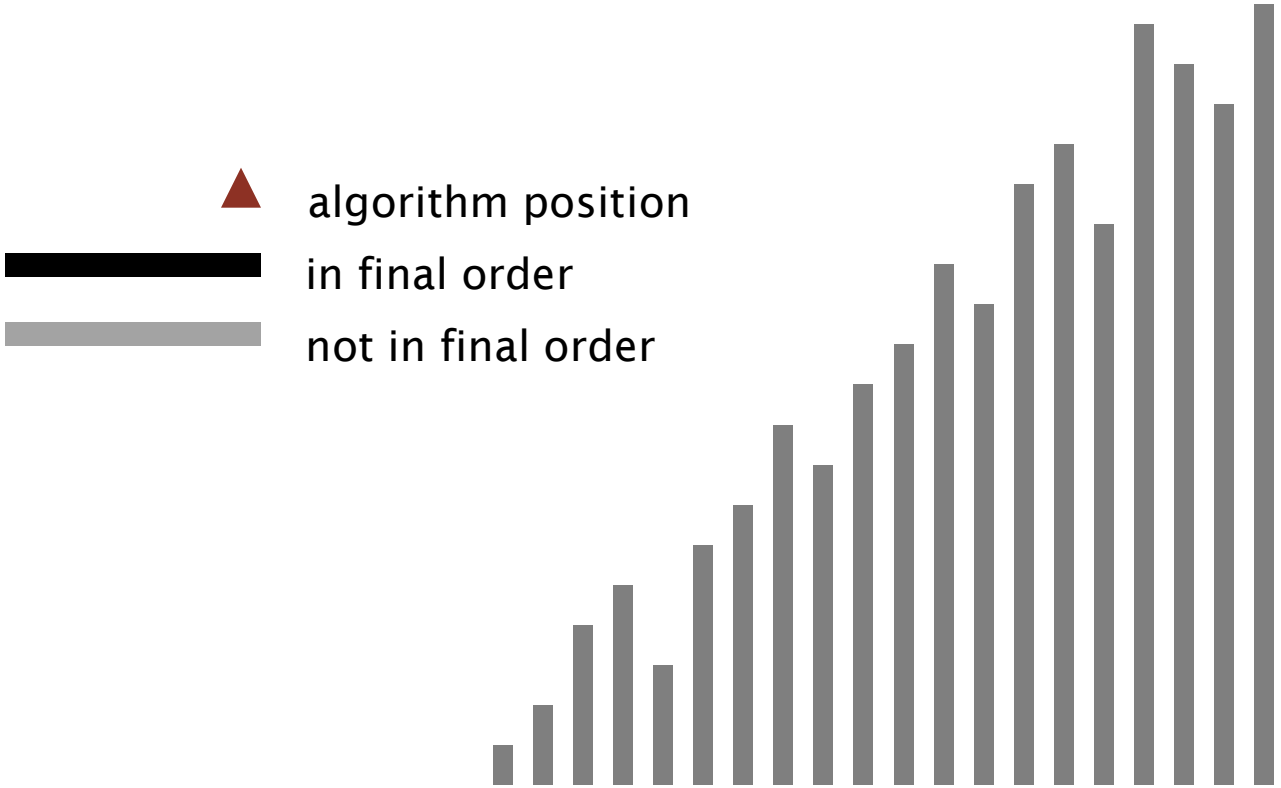
algorithm position

in final order

not in final order

From Sedgewick and Wayne, sorting-algorithms.com.

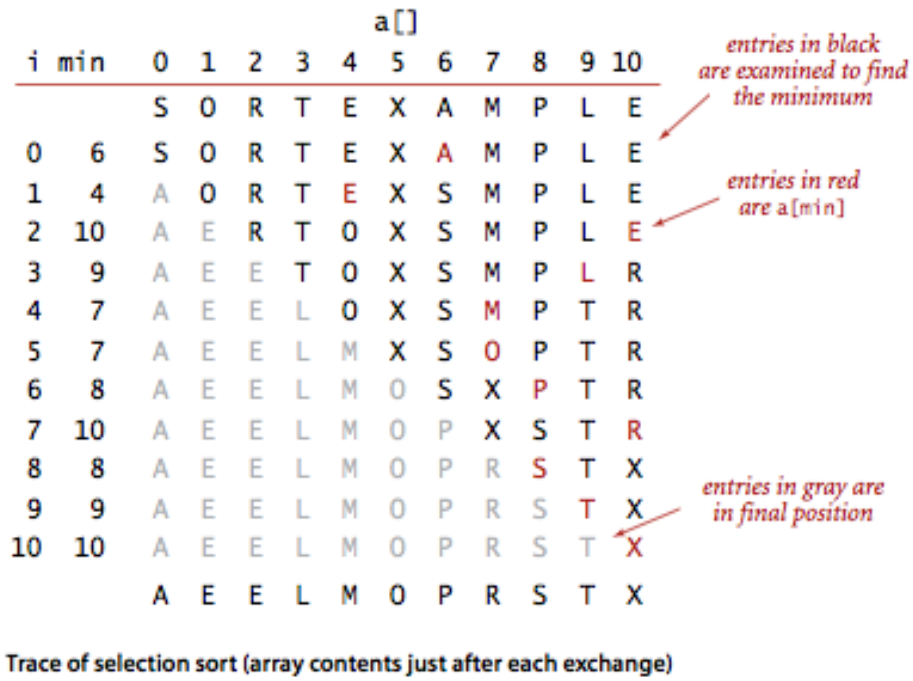
PARTIALLY SORTED INPUT



From Sedgewick and Wayne, sorting-algorithms.com.

PROPERTIES

```
public static void sort(Comparable[] a) {
    int N = a.length;

    for (int i = 0; i < N; i++)
    {
       int min = i;
       for (int j = i+1; j < N; j++)
           if (less(a[j], a[min])) min = j;
       exch(a, i, min);
    }
}
```



		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

- Compares: how many?
- Exchanges: how many?

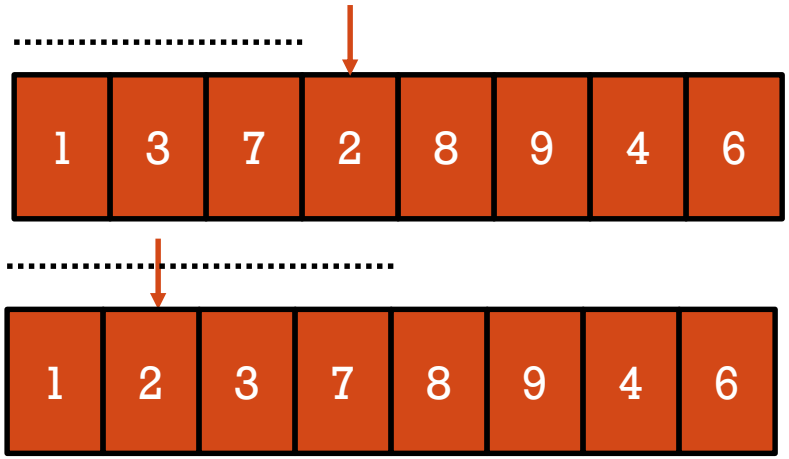
- Will the nature of input data effect the runtime of selection sort?

Image from Sedgewick and Wayne.

INSERTION SORT

THE CONCEPT

- Basic Idea:
  - (Assume there is a region of sorted elements at the front.)
  - Pick some element at position j.
  - Insert the element into a sorted position in the sorted region.
  - Repeat for all elements in the list.
- If the sorted elements are kept on the left side of the array, a region is formed that is sorted but whose elements may not be in their final position.



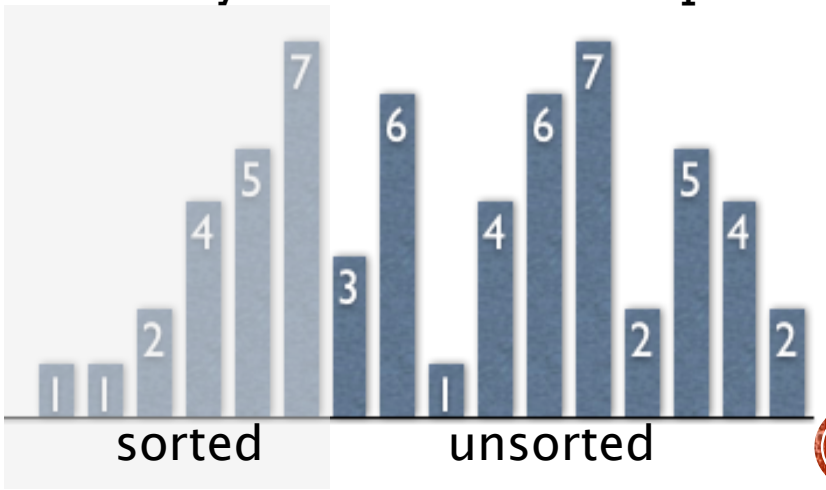


Image from Sedgewick and Wayne.

ALGORITHM TRACE

- Consider the following array: 7, 23, 25, 13, 2, 12, 3. Show a trace of execution for insertion sort. The trace should include the initial state of the array, followed by the array's state after each pass is made.

IMPLEMENTATION

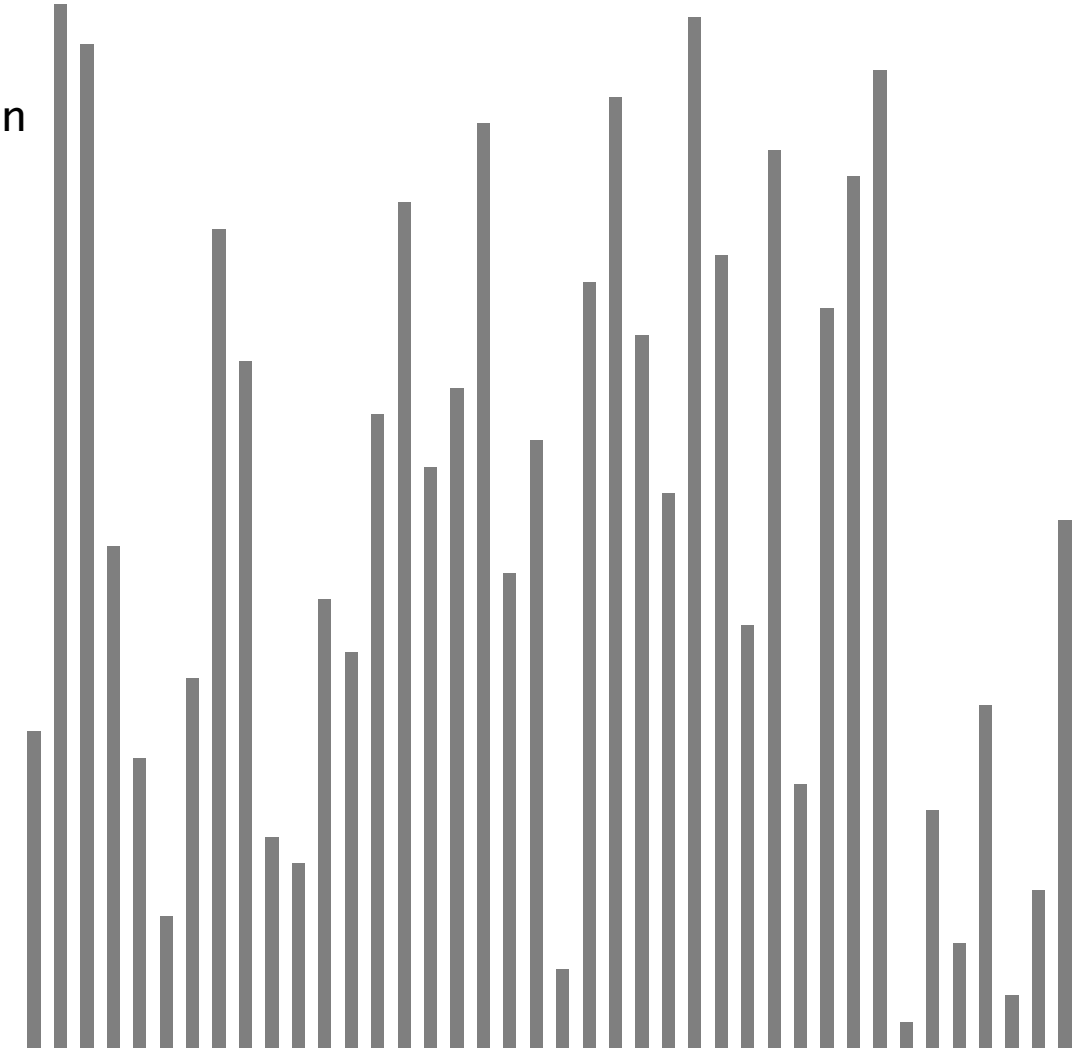
```
public static void sort(Comparable[] a) {
        int N = a.length;

        for (int i = 1; i < N; i++)
        {
            // Insert a[i] among a[i-1], a[i-2], a[i-3]... ..
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
        }
    }

    //helper
    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    //helper
    private static void exch(Comparable[] a, int i, int j) {
        Comparable t = a[i]; a[i] = a[j]; a[j] = t;
    }
```

RANDOM INPUT

algorithm position

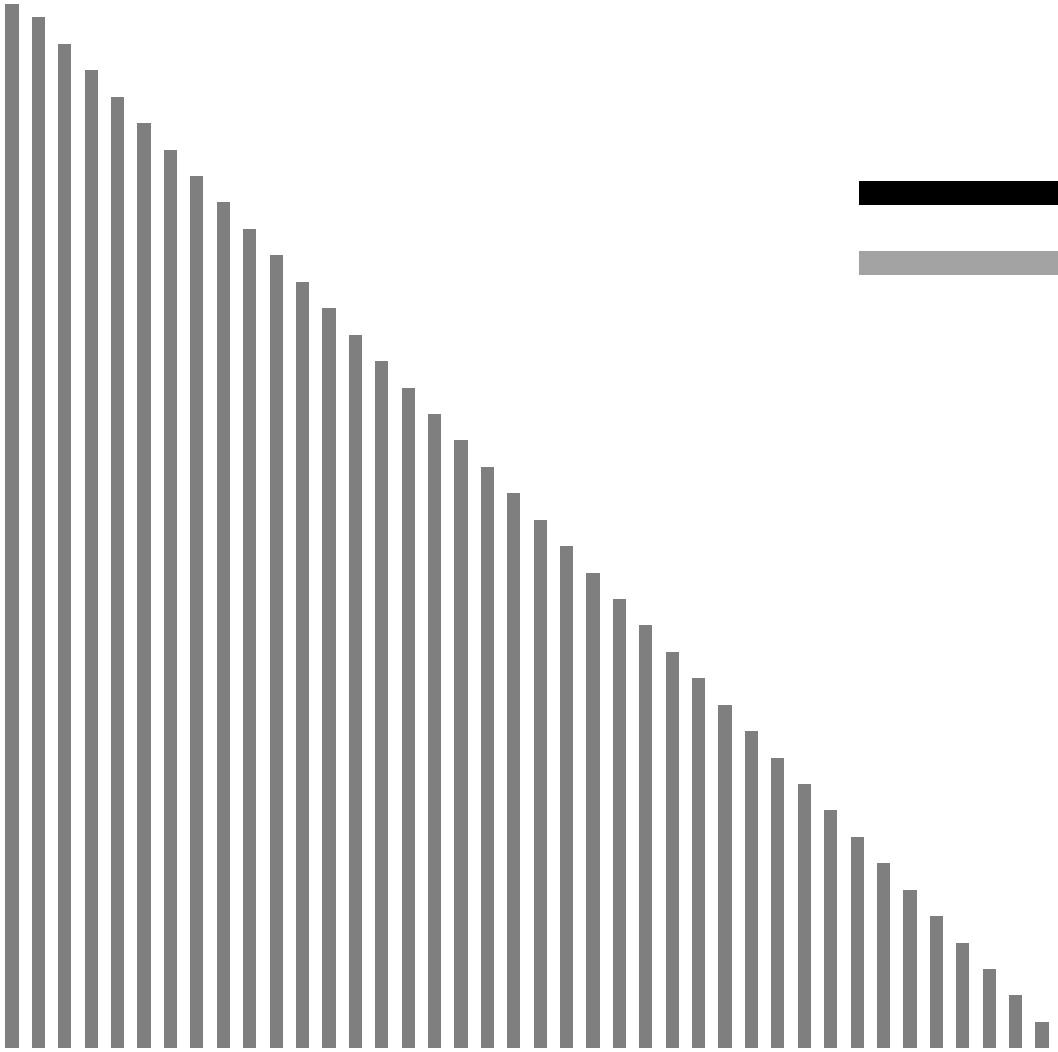
in final order

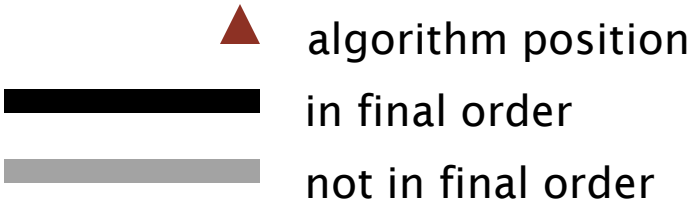
not in final order



From Sedgewick and Wayne, sorting-algorithms.com.

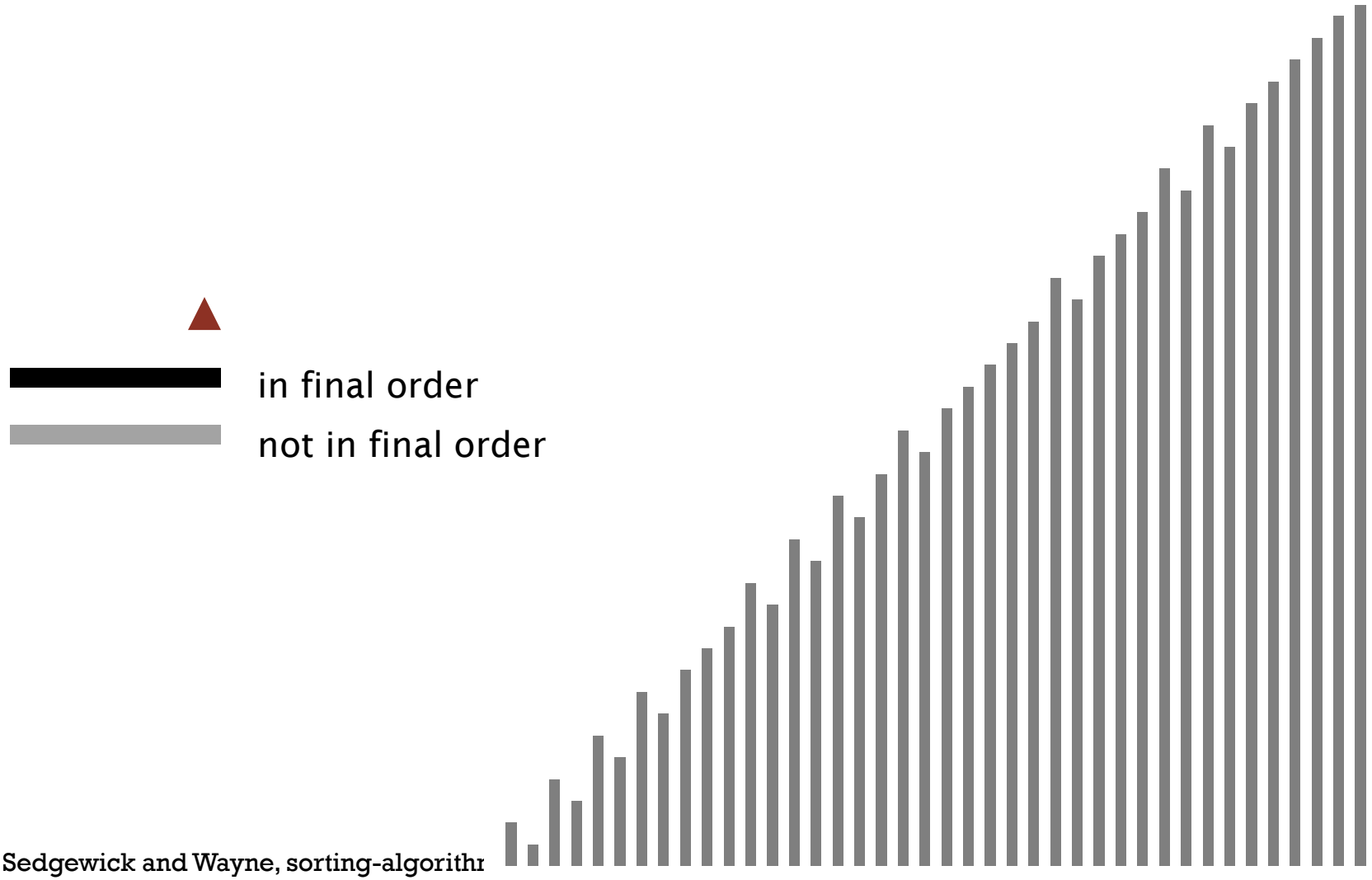
REVERSED INPUT





From Sedgewick and Wayne, sorting-algorithms.com.

PARTIALLY SORTED INPUT



From Sedgewick and Wayne, sorting-algorithı

PROPERTIES

For a (randomly ordered) input, the *average* case is:

- Comparisons: $\sim \frac{n^2}{4}$
- Exchanges: $\sim \frac{n^2}{4}$

For a (reverse input), the worse case is:

- Comparisons: $\sim \frac{n^2}{2}$
- Exchanges: $\sim \frac{n^2}{2}$

For a (sorted) input, the best case is:

- Comparisons: $n - 1$
- Exchanges: 0

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

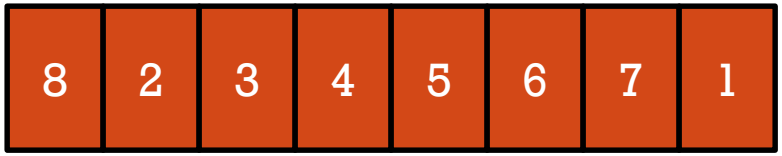
Trace of insertion sort (array contents just after each insertion)

```
public static void sort(Comparable[] a) {
    int N = a.length;

    for (int i = 1; i < N; i++)
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exch(a, j, j-1);
}
```

SHELLSORT

INSERTION SORT: INVERSIONS

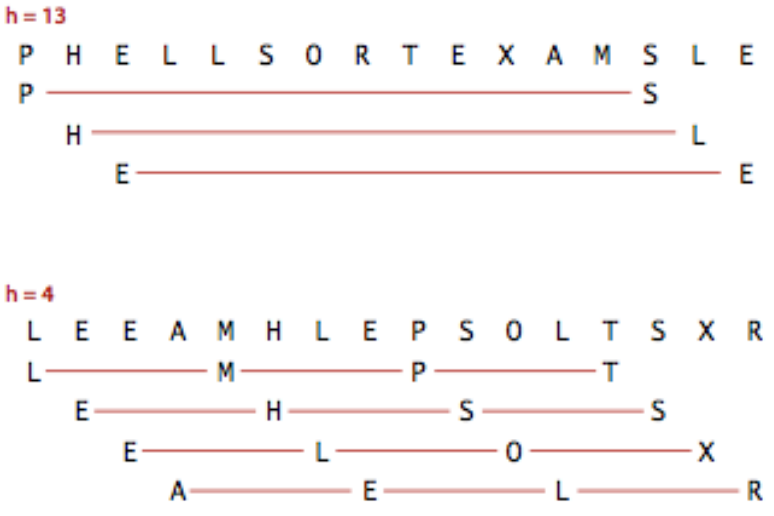
- Consider the following array:



- How many inversions are there?
- How many exchanges would insertion require to sort this?
- Working by hand, is it possible to do with fewer exchanges?
- Any ideas how to improve insertion sort?

SHELLSORT*

- Basic Idea:
- Perform an insertion sort starting at the ends of the array, and gradually shrink the internal until it is 1 (i.e., converges to normal insertion sort).
- Moves far away things first!
  - Creates a rough bound that says elements will only need to be moved so far.



*The algorithm is not named after its mechanism.

EXAMPLE

Input (11 elements)

S O R T E X A M P L E

7-sort

S O R T E X A M P L E
M O R T E X A S P L E
M O R T E X A S P L E
M O L T E X A S P R E
M O L E E X A S P R T

3-sort

M O L E E X A S P R T
E O L M E X A S P R T
E E L M O X A S P R T
E E L M O X A S P R T
A E L E O X M S P R T
A E L E O X M S P R T
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T

1-sort

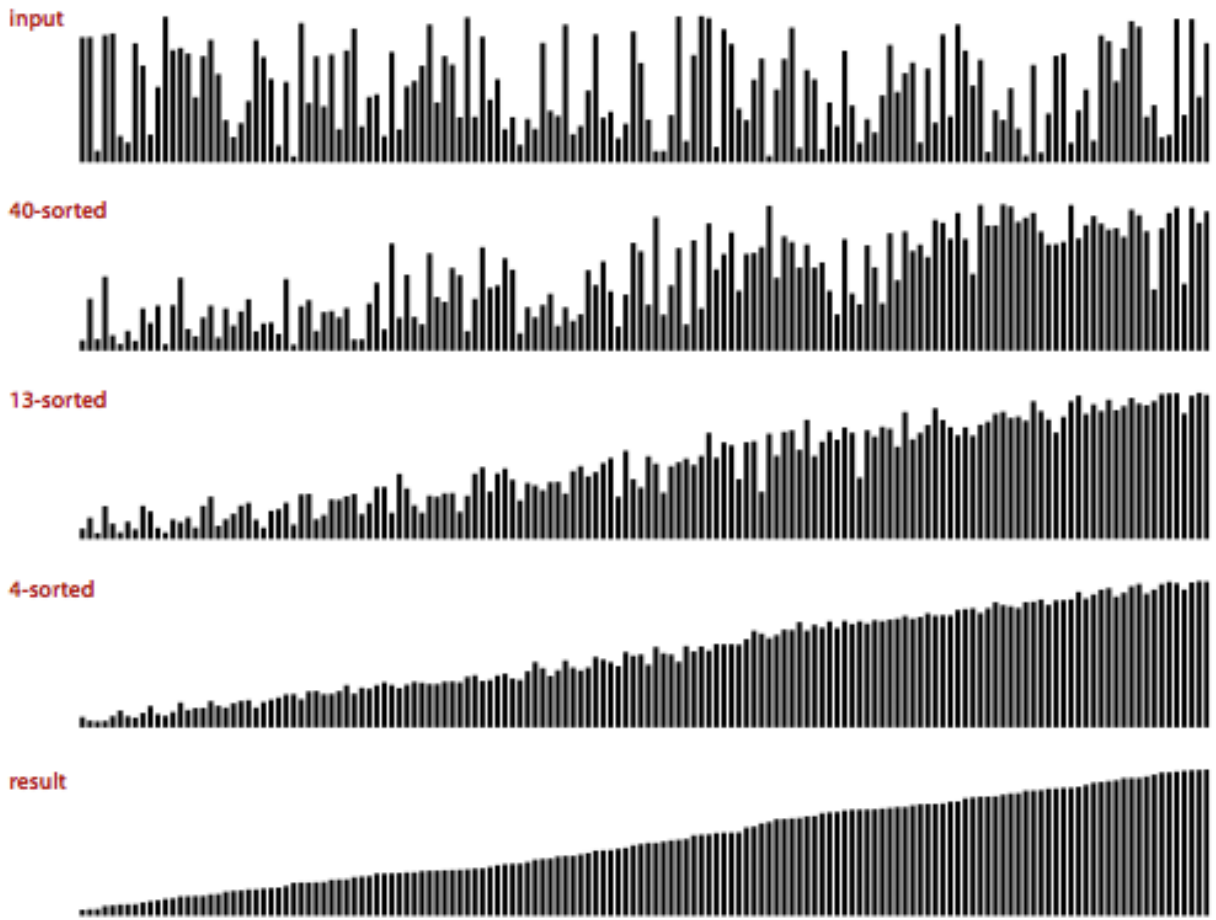
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T
A E E L O P M S X R T
A E E L O P M S X R T
A E E L M O P S X R T
A E E L M O P S X R T
A E E L M O P S X R T
A E E L M O P R S X T
A E E L M O P R S T X

result

A E E L M O P R S T X

Image from Sedgewick and Wayne.

TRACE



Visual trace of shellsort

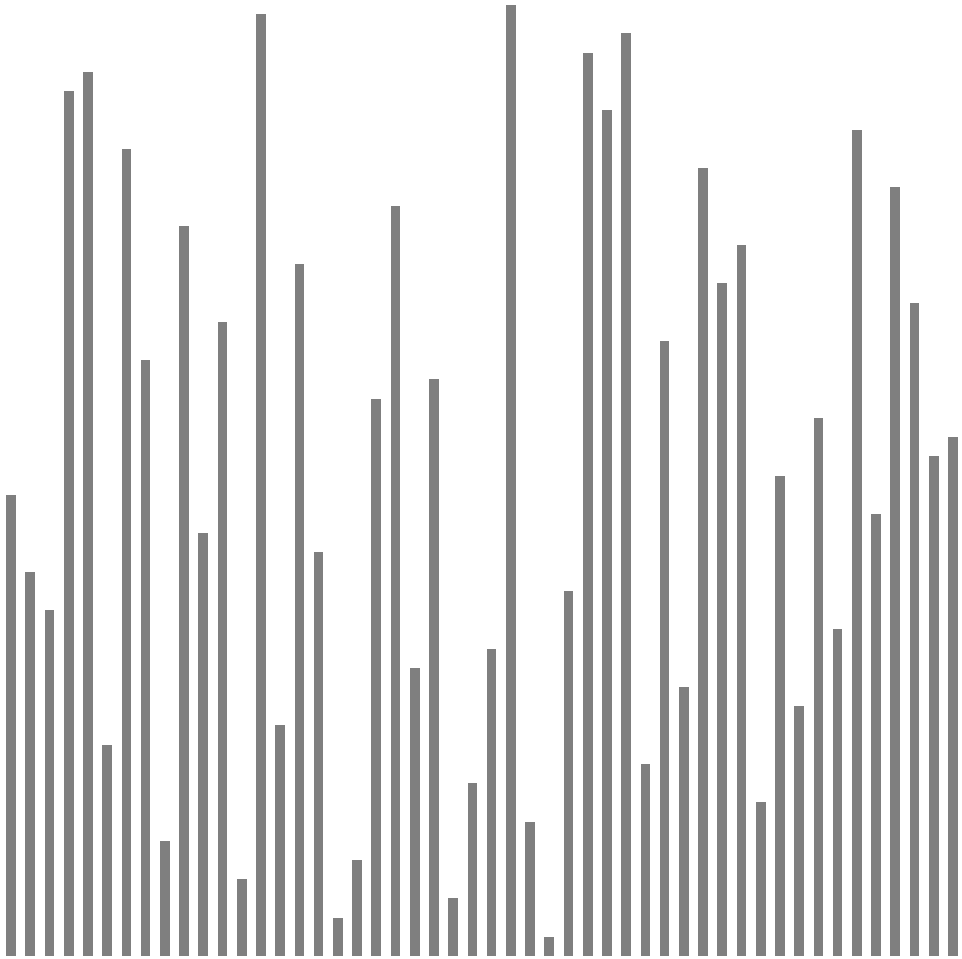
Image from Sedgewick and Wayne.

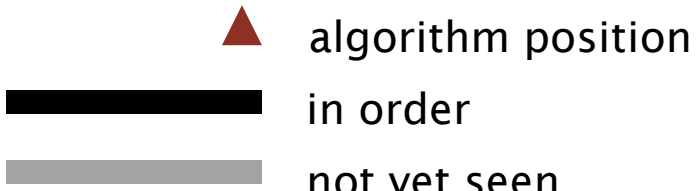
IMPLEMENTATION

What does the code look like if h = 1?

```
//Sedgewick and Wayne
public static void sort(Comparable[] a) {
       int N = a.length;
       int h = 1;

       while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...

       while (h >= 1) {
           // h-sort the array.
           for (int i = h; i < N; i++) {
               // Insert a[i] among a[i-h], a[i-2*h], a[i-3*h]... .
               for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                   exch(a, j, j-h);
           }
           h = h/3;
       }
   }

   //helper
   private static boolean less(Comparable v, Comparable w) {
       return v.compareTo(w) < 0;
   }

   //helper
   private static void exch(Comparable[] a, int i, int j) {
       Comparable t = a[i]; a[i] = a[j]; a[j] = t;
   }
```

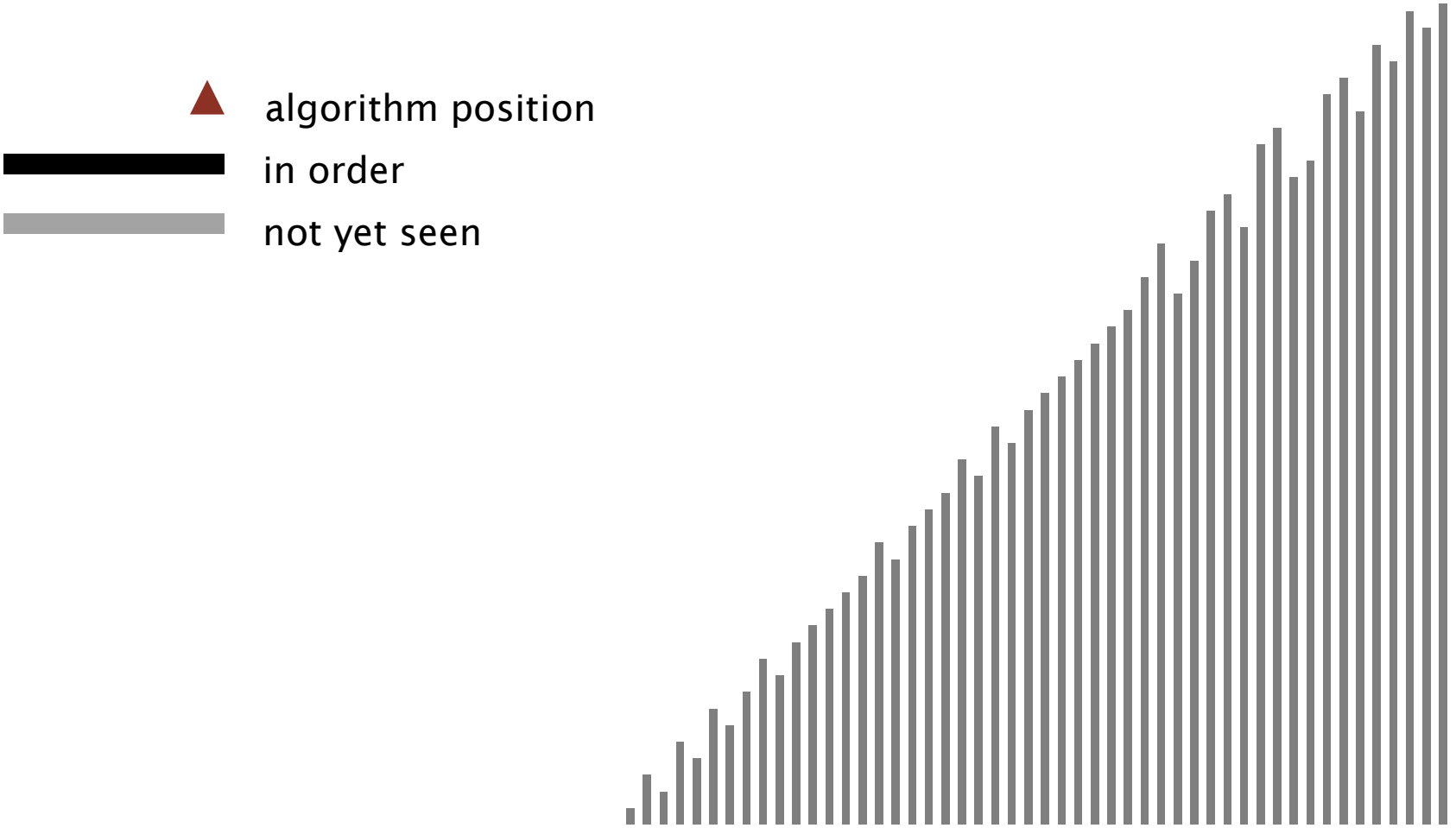
RANDOM INPUT





From Sedgewick and Wayne, sorting-algorithms.com.

PARTIALLY SORTED INPUT



From Sedgewick and Wayne, sorting-algorithms.com.

SHELLSORT PROPTERIES

- The textbook author has spent almost two decades analyzing shellsort.
- He still has not found the average run time of shellsort…

SUMMARY

PERFORMANCE

algorithm	best	average	worst
selection sort	N^2	N^2	N^2
insertion sort	N	N^2	N^2
Shellsort (3x+1)	$N \log N$	?	$N^{3/2}$
goal	N	$N \log N$	$N \log N$

What's this?

Table from Sedgewick and Wayne.