- Sorting algorithms: why?
- Insertion Sort.
- Inversions

- Binary search standard example
- Also supports other kinds of search:
  - Are there two equal items in this set?
  - Are there two items in this set that both have the same value for property X?
  - What are my nearest neighbors?
- Used in numerous unexpected algorithms, such as convex hull (smallest convex polygon enclosing set of points).

arranges) a sequence of elements to brings them into order, according to some *total order.*

- A total order, $\preceq$, is:
  - **Total:** $x \preceq y$ or $y \preceq x$ for all x, y.
  - **Reflexive:** $x \preceq x$;
  - **Antisymmetric:** $x \preceq y$ and $y \preceq x$ iff $x = y$.
  - **Transitive:** $x \preceq y$ and $y \preceq z$ implies $x \preceq z$.
- However, our orderings may treat unequal items as equivalent:
  - E.g., there can be two dictionary definitions for the same word. If we sort only by the word being defined (ignoring the definition), then sorting could put either entry first.

ory.

- *External sorts* process large amounts of data in batches, keeping what won't fit in secondary storage (in the old days, tapes).
- *Comparison-based* sorting assumes only thing we know about keys is their order.
- *Radix sorting* uses more information about key structure.
- *Insertion sorting* works by repeatedly inserting items at their appropriate positions in the sorted sequence being constructed.
- *Selection sorting* works by repeatedly selecting the next larger (smaller) item in order and adding it to one end of the sorted sequence being constructed.

- The java library provides static methods to sort arrays in the class `java.util.Arrays`.
- For each primitive type `P` other than `boolean`, there are

```
  /** Sort all elements of ARR into non-descending
order. */
  static void sort(P[] arr) { ... }

  /** Sort elements FIRST .. END-1 of ARR
into non-descending
   *  order. */
  static void sort(P[] arr, int first,
int end) { ... }

  /** Sort all elements of ARR into non-descending
order,
   *  possibly using multiprocessing for
speed. */
```

```
  /** Sort elements FIRST .. END-1 of ARR
into non-descending
   *  order, possibly using multiprocessing
for speed. */
  static void parallelSort(P[] arr, int
first, int end) {...}
```


- For reference types, `C`, that have a *natural order* (that is, that implement `java.lang.Comparable`), we have four analogous methods (one-argument `sort`, three-argument `sort`, and two `parallelSort` methods):

```
  /** Sort all elements of ARR stably into
non-descending
   * order. */
  static <C extends Comparable<? super
C>> sort(C[] arr) {...}
```

  etc.

- And for all reference types, `R`, we have four more:


```
  /** Sort elements FIRST .. END-1 of ARR
  static <R> void sort(R[] arr, Comparator<?
super R> comp) {...}
```

  etc.

- **Q:** Why the fancy generic arguments?

- For reference types, `C`, that have a *natural order* (that is, that implement `java.lang.Comparable`), we have four analogous methods (one-argument `sort`, three-argument `sort`, and two `parallelSort` methods):

```
  /** Sort all elements of ARR stably into
non-descending
   * order. */
  static <C extends Comparable<? super
C>> sort(C[] arr) {...}
```

  etc.

- And for all reference types, `R`, we have four more:


```
  /** Sort elements FIRST .. END-1 of ARR
  static <R> void sort(R[] arr, Comparator<?
super R> comp) {...}
```

  etc.

- **Q:** Why the fancy generic arguments?

- **A:** We want to allow types that have `compareTo` methods that apply also to more general types.

two methods similar to the sorting methods for arrays of reference types:

```
  static <C extends Comparable<? super C>>
sort(List<C> lst) {...}
  etc.

  /** Sort all elements of LST stably into
non-descending
   *  order according to the ordering
defined by COMP. */
  static <R> void sort(List<R> ,
Comparator<? super R> comp) {...}
  etc.
```

- Also an instance method in the `List<R>` interface itself:

```
  /** Sort all elements of LST stably into
non-descending
   *  order according to the ordering
defined by COMP. */
  void sort(Comparator<? super R> comp)
{...}
```



```
import static java.util.Arrays.*;
import static java.util.Collections.*;
```

- Sort `X`, a `String[]` or `List<String>`, into non-descending order:

```
sort(X);    // or ...
```

- Sort `X` into reverse order (Java 8):

```
sort(X, (String x, String y) -> { return
y.compareTo(x); });
  // or
  sort(X, Collections.reverseOrder());
// or
  X.sort(Collections.reverseOrder());    //
for X a List
```

- Sort `X[10], ..., X[100]` in array or `List X` (rest unchanged):

```
sort(X, 10, 101);
```


  - starting with empty sequence of outputs.
  - add each item from input, *inserting* into output sequence at right point.
- Very simple, good for small sets of data.
- With vector or linked list, time for find + insert of one item is at worst $\Theta(k)$, where k is # of outputs so far.
- This gives us a $\Theta(N^2)$ algorithm (worst case as usual).
- Can we say more?

- Consider a typical implementation for arrays:

```
for (int i = 1; i < A.length; i += 1) {
  int j;
  Object x = A[i];
  for (j = i-1; j >= 0; j -= 1) {
    if (A[j].compareTo(x) <= 0)  /* (1) */
      break;
    A[j+1] = A[j];               /* (2) */
  }
  A[j+1] = x;
}
```

- #times (1) executes for each `j` $\approx$ how far `x` must move.
- If all items within K of proper places, then takes $O(KN)$ operations.
- Thus good for any amount of *nearly sorted* data.

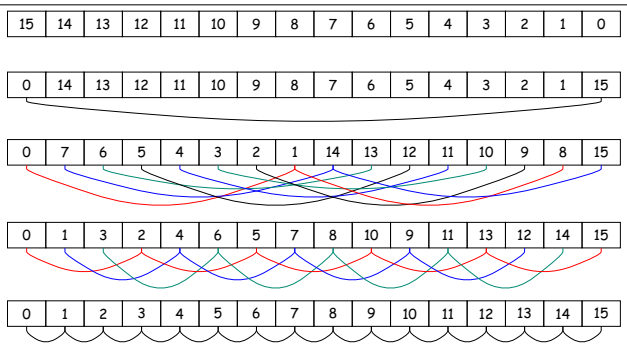
- Each execution of (2) decreases inversions by 1.

distant elements:

- First sort subsequences of elements $2^k - 1$ apart:
  - sort items #$0,\ 2^k-1,\ 2(2^k-1),\ 3(2^k-1),\ \ldots$, then
  - sort items #$1,\ 1+2^k-1,\ 1+2(2^k-1),\ 1+3(2^k-1),\ \ldots$, then
  - sort items #$2,\ 2+2^k-1,\ 2+2(2^k-1),\ 2+3(2^k-1),\ \ldots$, then
  - *etc.*
  - sort items #$2^k-2,\ 2(2^k-1)-1,\ 3(2^k-1)-1,\ \ldots$,
  - Each time an item moves, can reduce #inversions by as much as 2^k+1.
- Now sort subsequences of elements $2^{k-1}-1$ apart:

$1),\ 1+3(2^{k-1}-1),\ \ldots$,

- ⋮
- End at plain insertion sort ($2^0 = 1$ apart), but with most inversions gone.
- Sort is $\Theta(N^{3/2})$ (take CS170 for why!).



I: Inversions left.
C: Cumulative comparisons used to sort subsequences by insertion sort.