CS61B Lecture #26

ithms: why?

rt.

Purposes of Sorting

orts searching

h standard example

s other kinds of search:

two equal items in this set?

two items in this set that both have the same value for X?

my nearest neighbors?

rous unexpected algorithms, such as convex hull (small-olygon enclosing set of points).

Some Definitions

orithm (or *sort*) *permutes* (re-arranges) a sequence of brings them into order, according to some *total order.*

, $\preceq$, is:

y or $y \preceq x$ for all x, y.

: $x \preceq x$;

etric: $x \preceq y$ and $y \preceq x$ iff $x = y$.

: $x \preceq y$ and $y \preceq z$ implies $x \preceq z$.

orderings may treat unequal items as equivalent:

can be two dictionary definitions for the same word. t only by the word being defined (ignoring the defini- sorting could put either entry first.

at does not change the relative order of equivalent en- pared to the input) is called *stable.*

Classifications

ts keep all data in primary memory.

ts process large amounts of data in batches, keeping it in secondary storage (in the old days, tapes).

based sorting assumes only thing we know about keys is

uses more information about key structure.

rting works by repeatedly inserting items at their ap- sitions in the sorted sequence being constructed.

rting works by repeatedly selecting the next larger m in order and adding it to one end of the sorted se- constructed.

---

ays of Primitive Types in the Java Library

ary provides static methods to sort arrays in the class `rrays`.

nitive type `P` other than `boolean`, there are

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

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

 all elements of ARR into non-descending order,
bly using multiprocessing for speed. */
id parallelSort(P[] arr) { ... }

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

---

ys of Reference Types in the Java Library

e types, `C`, that have a *natural order* (that is, that im- `a.lang.Comparable`), we have four analogous methods nt `sort`, three-argument `sort`, and two `parallelSort`

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

eference types, `R`, we have four more:

```
 all elements of ARR stably into non-descending order
ding to the ordering defined by COMP. */
> void sort(R[] arr, Comparator<? super R> comp) {...}
```

fancy generic arguments?

ys of Reference Types in the Java Library

ce types, `C`, that have a *natural order* (that is, that im-
a.lang.Comparable), we have four analogous methods
nt `sort`, three-argument `sort`, and two `parallelSort`

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

eference types, `R`, we have four more:

```
 all elements of ARR stably into non-descending order
ding to the ordering defined by COMP. */
> void sort(R[] arr, Comparator<? super R> comp) {...}
```

fancy generic arguments?

to allow types that have `compareTo` methods that apply
general types.

---

orting Lists in the Java Library

va.util.Collections contains two methods similar to
nethods for arrays of reference types:

```
 all elements of LST stably into non-descending
. */
 extends Comparable<? super C>> sort(List<C> lst) {...}

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

nce method in the `List<R>` interface itself:

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

Examples

```
atic java.util.Arrays.*;
atic java.util.Collections.*;
```

ing[] or `List<String>`, into non-descending order:

```
    // or ...
```

everse order (Java 8):

```
(String x, String y) -> { return y.compareTo(x); });

ollections.reverseOrder());   // or
llections.reverseOrder());    // for X a List
```

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

```
0, 101);
```

 ..., `L[100]` in list `L` (rest unchanged):

```
blist(10, 101));
```

Sorting by Insertion

ith empty sequence of outputs.

item from input, *inserting* into output sequence at right

good for small sets of data.

 or linked list, time for find + insert of one item is at
where k is # of outputs so far.

 a $\Theta(N^2)$ algorithm (worst case as usual).

ore?

Inversions

$N)$ comparisons if already sorted.

pical implementation for arrays:

```
; i < A.length; i += 1) {

[i];
; j >= 0; j -= 1) {
ompareTo(x) <= 0)  /* (1) */

[j];               /* (2) */
```

xecutes for each `j` $\approx$ how far `x` must move.

ithin K of proper places, then takes $O(KN)$ operations.

r any amount of *nearly sorted* data.

 of unsortedness: # of *inversions:* pairs that are out
 when sorted, $N(N-1)/2$ when reversed).

ion of (2) decreases inversions by 1.

Shell's sort

 insertion sort by first sorting *distant* elements:

bsequences of elements $2^k - 1$ apart:

s #$0,\ 2^k-1,\ 2(2^k-1),\ 3(2^k-1),\ \ldots$, then
s #$1,\ 1+2^k-1,\ 1+2(2^k-1),\ 1+3(2^k-1),\ \ldots$, then
s #$2,\ 2+2^k-1,\ 2+2(2^k-1),\ 2+3(2^k-1),\ \ldots$, then

s #$2^k-2,\ 2(2^k-1)-1,\ 3(2^k-1)-1,\ \ldots$,
 an item moves, can reduce #inversions by as much as

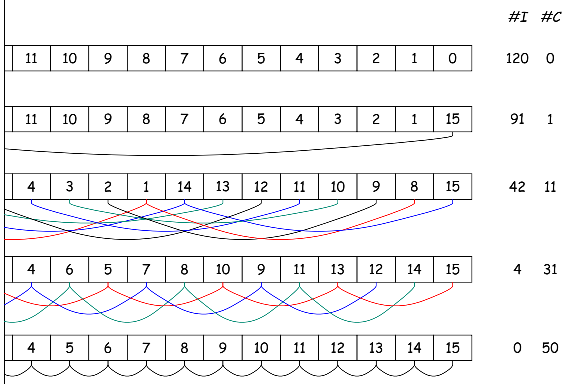
sequences of elements $2^{k-1} - 1$ apart:

s #$0,\ 2^{k-1}-1,\ 2(2^{k-1}-1),\ 3(2^{k-1}-1),\ \ldots$, then
s #$1,\ 1+2^{k-1}-1,\ 1+2(2^{k-1}-1),\ 1+3(2^{k-1}-1),\ \ldots$,

insertion sort ($2^0 = 1$ apart), but with most inversions

$/2)$ (take CS170 for why!).

Example of Shell's Sort



												#I	#C
11	10	9	8	7	6	5	4	3	2	1	0	120	0
11	10	9	8	7	6	5	4	3	2	1	15	91	1
4	3	2	1	14	13	12	11	10	9	8	15	42	11
4	6	5	7	8	10	9	11	13	12	14	15	4	31
4	5	6	7	8	9	10	11	12	13	14	15	0	50

ft.

mparisons used to sort subsequences by insertion sort.