CS61B Lecture 26

Monday, March 30, 2020

Announcements

- Details on Midterm 2 have been sent out. The assignment will be on Gradescope, and the test will run from 5 to 7 PM on Wednesday.

Sorting

Why is there such great interest in the sorting algorithm? One of the things computers do is to search through data, and sorting makes this easier. In addition, we can use search to answer questions such as:

- 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?

We also see search in numerous unexpected algorithms, such as the convex hull problem.

A **sorting algorithm** (or sort) permutes a sequence of elements to bring them into order, according to some total order.

A **total order**,

$$\leq$$

is:

- total:

$$x < y \text{ or } y < x, \text{for all } x, y$$

- reflexive:

$$x \leq x$$

- antisymmetric:

$$x \leq y \text{ and } y \leq x \text{ iff } x = y$$

- transitive:

$$x \leq y \text{ and } y \leq z \text{ implies } x \leq z$$

However, our orderings may treat unequal items as equivalent:

- For example, there can two dictionary definitions for the same word. If we sort only by the word being defined, then sorting could put either entry first
- A sort does not change the relative order of equivalent entries (compared to the input) is called **stable**.

There are many types of sorting:

- Internal sorts keep all data in primary memory.

- 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.

Sorting Arrays of Primitive Types in the Java Library

This is another example of where we are being taught these algorithms and concepts, but in the real world, we are using somebody else's work because it is more efficient and one less thing for you to do. The reason we are taught these concepts is to understand the tools under the hood.

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

```
/** 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. */
static void parallelSort(P[] arr) { ... }

/** 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) {...}
```

Normally, we should not make a distinction between `sort` and `parallelSort`, and let the algorithm decide whether to use parallelism or not on its own, as part of abstraction. The reason that Java makes this not the case is that parallelism can get a little funky if you try to do too many tasks in parallel at once, so it's nice to be able to control it.

Sorting Arrays of Reference Types in the Java Library

With reference types, there is a different story. For a reference type, C, that have a natural order (they implement `Comparable`), we have four analogous methods like before (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) { ... }
```

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

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

Here's a question: why are the generic arguments so fancy? The reason is that we want to allow types that have `compareTo` methods that apply also to more general types. In other words, there's no reason that objects of type `C` can only be comparable to other objects of type `C`; it could also be comparable to objects of its supertype.

Sorting Lists in the Java Library

The `Collections` class contains two methods similar to the sorting methods for arrays of reference types:

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

/** 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) { ... }
```

In addition, there is 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) { ... }
```

This is also analogous to Python, which has both static and instance methods.

Examples

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

/* Sort X, a String[] or List<String> into non-descending order. */
sort(X);

/* 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());

/* Sort X[10], ..., X[100] in array or List X,
leaving the rest unchanged. Uses views to achieve this. */
sort(X, 10, 101);

/* Sort L[10], ..., L[100] in list L(rest unchanged). */
sort(L.sublist(10, 101));
```

Sorting by Insertion

Let's start with a simple idea. Start with an empty sequence of outputs, and add each item from the input, inserting into the output sequence at the right point.

This is very simple and good for small sets of data. For vectors and linked lists, the time to kind and insert is, at worst, linear to the number of outputs so far, which gives us a quadratic algorithm in the worst case.

Inversions

There is a typical algorithm for doing insertion sort, which can run in linear time if already sorted:

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

The number of times block (1) is executed for each `j` is how far `x` must move. If all items are within `K` of proper places, then it takes O(`KN`) operations.

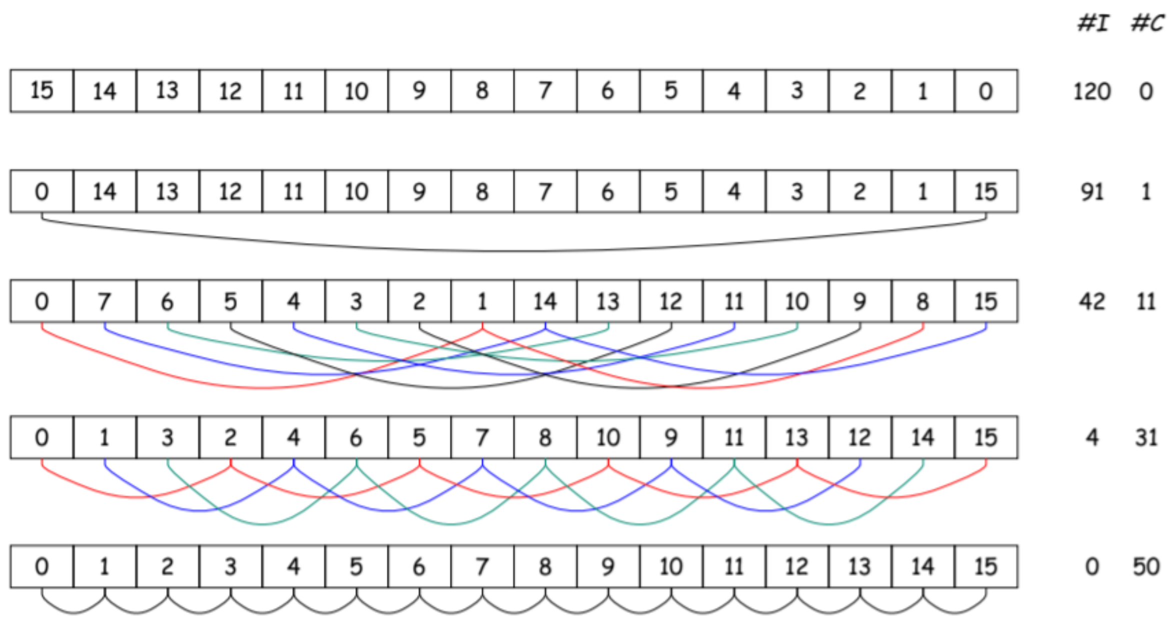
Thus, it is good for any amount of *nearly sorted* data. How do we measure unsortedness? We can measure it as the number of inversions -- the number of pairs that are out of order (0 when sorted, N(N-1)/2 when reversed).

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

Shell's Sort

In any case, this algorithm is quite slow in the worst case. Maybe we can use this insight into inversion to speed it up a bit, by first sorting distant elements, and get them closer together, which may decrease the number of elements that are inverted.

- 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:
  - sort items #$0,\ 2^{k-1}-1,\ 2(2^{k-1}-1),\ 3(2^{k-1}-1),\ \ldots$, then
  - sort items #$1,\ 1+2^{k-1}-1,\ 1+2(2^{k-1}-1),\ 1+3(2^{k-1}-1),\ \ldots$,
  - $\vdots$
- 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 #C
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 120 0
0 14 13 12 11 10 9 8 7 6 5 4 3 2 1 15 91 1
0 7 6 5 4 3 2 1 14 13 12 11 10 9 8 15 42 11
0 1 3 2 4 6 5 7 8 10 9 11 13 12 14 15 4 31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 50