Sorting

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

Photo courtesy of ellenmc / Flickr:
http://www.flickr.com/photos/ellenmc/4508741746/

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Sorting

Standard reminder to set phones to silent/vibrate mode, please!

Sorting

- During today's class:
  - Introduce Sorting and related concepts
  - Discuss some aspects of the run time of sorting algorithms
  - Look into some of the basic (inefficient) algorithms
  - Introduce merge sort
    - Discuss its run time

Sorting

- Basic definition:
  - Process of taking a list of objects with a linear ordering

$$\left(a_{1,}\, a_{2,}\ \cdots\ , a_{n-1}, a_n\right)$$

and output a permutation of the list

$$\left(a_{k_1}, a_{k_2},\ \cdots\ , a_{k_{n-1}}, a_{k_n}\right)$$

such that

$$a_{k_1} \leqslant a_{k_2} \leqslant\ \cdots\ \leqslant a_{k_{n-1}} \leqslant a_{k_n}$$

Sorting

- More often than not, we're interested in sorting a list of “records” in order by some field (e.g., we have a list of students with the various grades — assignments, midterm, labs, etc.) and we want to sort by student ID, or we want to sort by final grade, etc.

Sorting

- However, in our discussions of the various sort algorithms, we will assume that:
  - We're sorting values (objects that can be directly compared).
    - The more general case can be seen as an implementation detail
  - We're using arrays for both input and output of the sequences of values.

Sorting

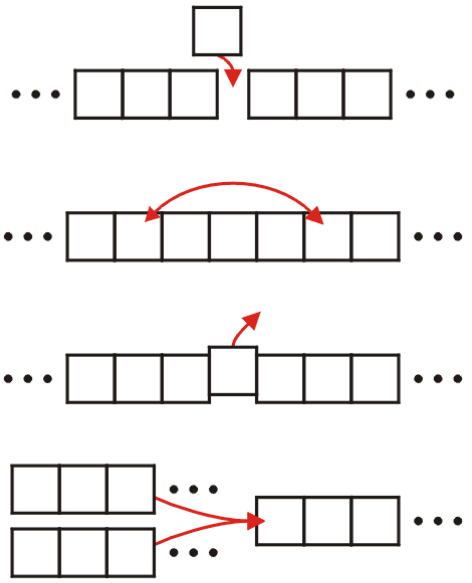
- We will be most interested in algorithms that can be performed *in-place*.
  - That is, requiring Θ(1) additional memory (a fixed number of local variables)
  - Some times, the definition is stated as requiring o(n) additional memory (i.e., strictly less than linear amount of memory); this allows recursive methods that divide the range in two parts and proceed recursively (Θ(log *n*) recursive calls, which means a "hidden" memory usage of Θ(log *n*)

Sorting

- The intuition of in-place refers to doing all the operations in the same area where the input values are, without requiring allocation of a second array of the same size.

Sorting

- Some of the typical operations used by sorting algorithms are:
  - Insertion
  - Swapping
  - Selection
  - Merging



Sorting

- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$

Sorting

- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$
- However, there are algorithms that run in linear time

Sorting

- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$
- However, there are algorithms that run in linear time (huh???)

Sorting

- We've already discussed that (under some more or less standard assumptions), no sort algorithm can have a run time better than $n \log n$
- However, there are algorithms that run in linear time (huh???)
  - No, I'm not pulling your leg — in fact, if you read carefully the first paragraph you should notice that the second paragraph does not necessarily contradict the first one.

Sorting

- The ones we'll study are either:
  - $\Theta(n \log n)$
    - Merge sort, heap sort, quick sort
  - $\Theta(n^2)$
    - Insertion sort, selection sort, bubble sort
  - $\omega(n^2)$
    - Bogosort (though this one has a remarkable variation, the Quantum bogosort, that executes in linear time — unconditionally, and without any assumptions or constraints on the data!!)

Sorting

- We'll go in reverse — starting by the slower ones (we'll discuss bogosort and its variation, the quantum bogosort, in class — i.e., not in these “pre-lecture” slides)

Sorting

- The simpler algorithms (both in terms of their description and in terms of their actual operation) tend to be the slower ones:
  - Selection sort is one of the simplest sorting algorithms. It's *really* simple:

Sorting

- Selection sort:
  - At the first iteration:
    - Find the lowest value (in particular, its position in the array); then, swap it with the first element of the array.
  - Second iteration:
    - Find the lowest value starting at the second element; then, swap it with the second element of the array.
  - And so on, until the second to last element.

Sorting

- Selection sort:
  - Can you determine its run time?
  - Also: is it an in-place algorithm?

Sorting

- Insertion sort is another very simple algorithm:
  - Traversing the array from first to last element, for element *k*, search for the right position for that value in the range 0 to *k*-1 of the array, and insert it at that position.
    - By “insert”, we mean: shift the remaining elements one position forward, to open up the one location where to put the element.
  - As a slight optimization, we usually check first whether the value is less than its previous value (if it's not, then we don't need to do anything at this iteration).

Sorting

- Insertion sort:
  - Run time?
  - Is it in-place?
  - A non-trivial question: Is it faster than selection sort? (we'd like to compare average-case run times and worst-case run times)

Sorting

- Bubble-sort is perhaps the one with the simplest implementation among all sorting algorithms — at least in its unoptimized form:
  - Do *n* times the following:
    - Traverse the array, checking each element with its next element (neighbouring element) — if they're in the wrong order, swap them (noticing that the loop continues, and now the “next” position will be the element that we just swapped)

Sorting

- Merge sort:
  - We've already seen a partial description of this one (assignment 2).
  - We saw that its run time is $n \log n$ (well, assuming that you believed me that the merge function runs in linear time — hopefully you did believe me! :-))

Sorting

- Merge sort:
  - We've already seen a partial description of this one (assignment 2).
  - We saw that its run time is $n \log n$ (well, assuming that you believed me that the merge function runs in linear time — hopefully you did believe me! :-))

Sorting

- Merge sort:
  - We've already seen a partial description of this one (assignment 2).
  - We saw that its run time is $n \log n$ (well, assuming that you believed me that the merge function runs in linear time — hopefully you did believe me! :-))
  - This one falls in the *divide-and-conquer* category of algorithms: divide the problem into simpler parts (to the point where the problem turns into a much easier — often trivial — problem, so we “conquer” the original problem)

Sorting

- Merge sort:
  - So simple — really, soooooo simple:
    - Split the array into two halves.
    - Sort (using the same merge sort) the first half
    - Then, sort the second half
    - Then, merge them (since they are ordered sequence, it should be easy to merge them *in linear time* into a single ordered sequence... right?)
      - You guys tell me how (we'll go over it in class)

Sorting

- Merge sort:
  - In this case, the “simpler” base case is really a *trivial* algorithm — when the array size reaches one, the sort procedure is really the *null* procedure (an array of one element is already a sorted array, so we do nothing).

Sorting

- Merge sort:
  - Like I mentioned, we already saw that the run time is $\Theta(n \log n)$
  - What about space? Can it be executed in-place? (you have to first answer the "how do we merge?" question before addressing this one)

Summary

- During today's lesson:
  - We discussed sorting and related concepts
  - Looked into the run time of sorting algorithms
  - Introduced insertion sort, selection sort, and bubble sort.
  - Introduced and analyzed merge sort.