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

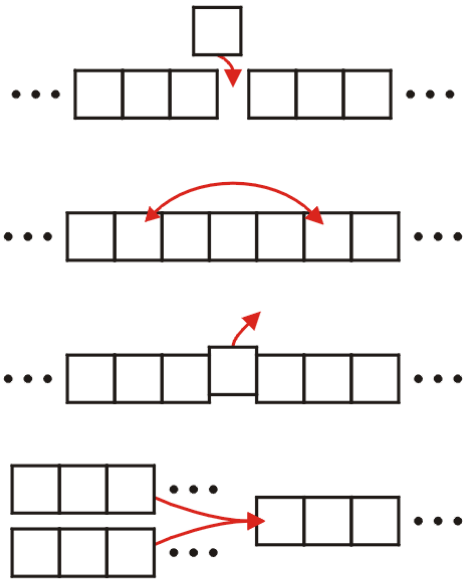
- We will be most interested in algorithms that can be performed *in-place*.
  - That is, requiring Θ(1) additional memory (a fixed number of local variables)
  - Some times, the definition is stated as requiring o(n) additional memory (i.e., strictly less than linear amount of memory); this allows recursive 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 key detail is that one of the conditions for that lower-bound is that it is applicable to sorting *arbitrary* (possibly random) data.
  - If the data is subject to some constraints, then we can indeed sort in linear time
    - And even then, one could get into a philosophical discussion making the case that it is not really linear time... But let's leave it at that...

Sorting

- Notice that we do have already a very straightforward way (at least straightforward to describe) to sort *n* values:
  - Insert them in an AVL tree, then do an in-order traversal to output the sorted sequence.
    - Each insertion takes $\Theta(\log n)$, and there are *n* of them
  - BTW, this was essentially the answer to the bonus marks question in the midterm — reduction from sort: if insertion in a BST was faster than $\log n$, then we would have a sort that is faster than $n \log n$.

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 (Well, the quantum bogosort is really the fastest, but ...)

Sorting

- Bogosort:
  - Randomly reorder the elements.
  - Check if they are sorted.
  - If not, repeat.

Sorting

- Bogosort:
  - Randomly reorder the elements.
  - Check if they are sorted.
  - If not, repeat.
- Run-time:
  - Best-case: $\Theta(n)$ (reordering can be done in linear time, and checking is obviously linear time).
  - Average-case: $\Theta(n!)$ (there are $n!$ Permutations!)
  - Worst-case: ... (you guys tell me?)

Sorting

- Bogosort:
  - Randomly reorder the elements.
  - Check if they are sorted.
  - If not, repeat.
- Run-time:
  - Best-case: $\Theta(n)$ (reordering can be done in linear time, and checking is obviously linear time).
  - Average-case: $\Theta(n!)$ (there are $n!$ Permutations!)
  - Worst-case: Unbounded (right?)

Sorting

- The interesting variation, the quantum bogosort does the following:
  - Use a quantum-level source of random binary events to randomly reorder the sequence.

Sorting

- The interesting variation, the quantum bogosort does the following:
  - Use a quantum-level source of random binary events to randomly reorder the sequence.
  - If the resulting sequence is not sorted, destroy the universe.

Sorting

- The interesting variation, the quantum bogosort does the following:
  - Use a quantum-level source of random binary events to randomly reorder the sequence.
  - If the resulting sequence is not sorted, destroy the universe.
  - Anyone wants to take a stab at seeing why it works, and why it is so fast? (linear time)

Sorting

- From the Many Worlds Interpretation (MWI) of Quantum Physics, an infinite number of universes (realities) exist simultaneously.

Sorting

- From the Many Worlds Interpretation (MWI) of Quantum Physics, an infinite number of universes (realities) exist simultaneously.
- Every outcome of a quantum-level random event does occur — it just splits the reality into all corresponding realities.

Sorting

- From the Many Worlds Interpretation (MWI) of Quantum Physics, an infinite number of universes (realities) exist simultaneously.
- Every outcome of a quantum-level random event does occur — it just splits the reality into all corresponding realities.
- So, all 2^n possible realities do take place (in parallel), but the $2^n - 1$ universes where the sort failed were destroyed, and in the universe that remains, the bogosort was completed with a single permutation — thus, it ran in $\Theta(n)$

Sorting

- True story ... !!

Sorting

- Back to our “single reality” paradigm, where we're interested in the reasonable sorting algorithms...

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:
  - Run time?

Sorting

- Selection sort:
  - Run time?
  - It's clearly quadratic: The first pass, we search through exactly $n-1$ elements (no difference between average-case and worst-case), then swap (constant time). Second time, $n-2$ elements, then $n-3$, etc.
  - We get (yet again!) the arithmetic sum — $\Theta(n^2)$

Sorting

- Selection sort:
  - Run time?
  - It's clearly quadratic: The first pass, we search through exactly $n-1$ elements (no difference between average-case and worst-case), then swap (constant time). Second time, $n-2$ elements, then $n-3$, etc.
  - We get (yet again!) the arithmetic sum — $\Theta(n^2)$
  - It is clearly an in-place algorithm (we swap elements in the array)

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

- Insertion sort:
  - Run time is clearly quadratic (analysis is similar to that of selection sort — emphasis on *similar*; not exactly the same)
    - At first glance, this may sound horribly inefficient, since for a single element we need to move $\Theta(n)$ elements to shift the values and open space for the current element.
    - However, the average-case benefits from the fact that for many elements, it is already greater than all the elements before, so we don't need to do anything (for selection sort, every single pass requires $\Theta(n)$ to search for the lowest!)

Sorting

- Insertion sort:
  - Bottom line: remarkably enough (perhaps *surprisingly* enough), on average, insertion sort outperforms selection sort
    - Yes, by a constant speedup factor, which plays no role in asymptotic analysis, but as we know, it does play a role in a practical, real-life implementation where we just want or need *the fastest* we can get!

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! :-))
  - 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?

Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):
  - Example:

    Sequence A: 11, 23, 40, 57, 78, 93
    Sequence B: 5, 9, 35, 36, 39, 63

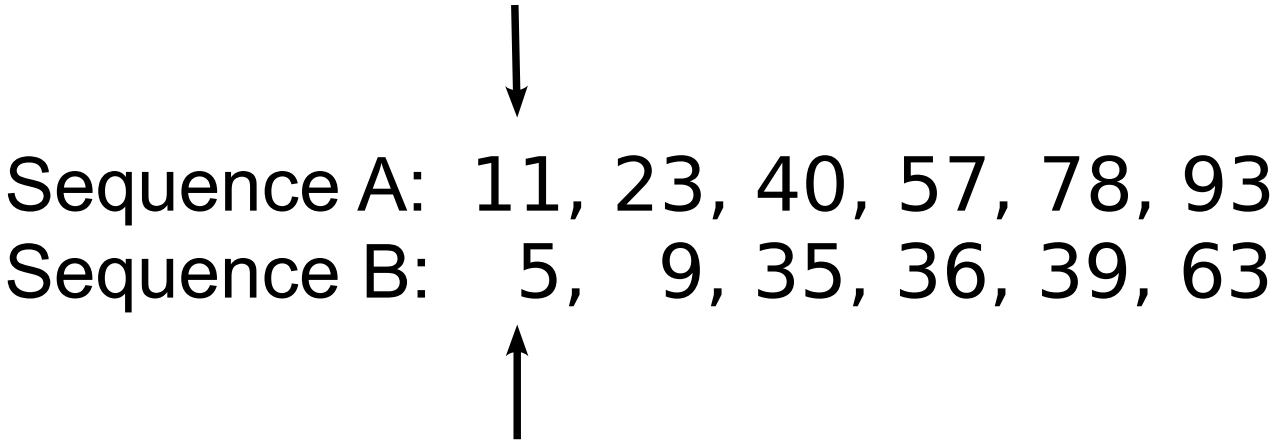
  - Can you guys go through the resulting merged sequence efficiently?

Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):
  - Because the sequences are in order, we don't need to check every element of one sequence for each element processed from the other — we just keep track of where we are on each sequence, and advance on the corresponding one.

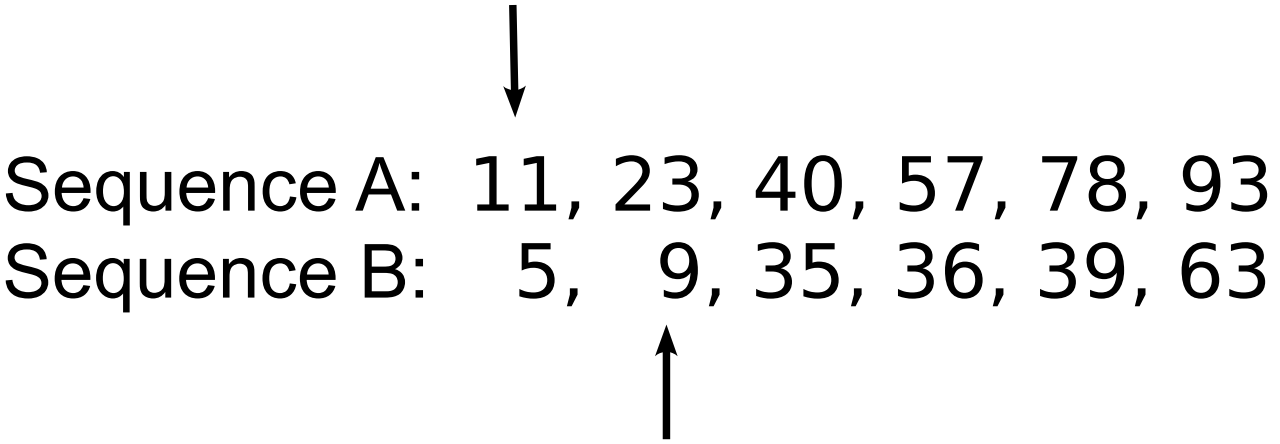
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



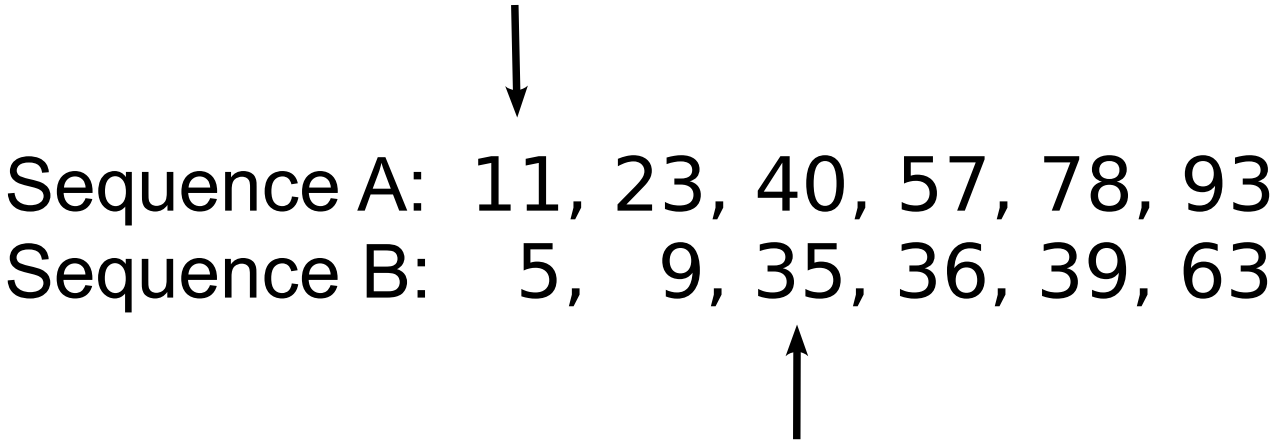
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



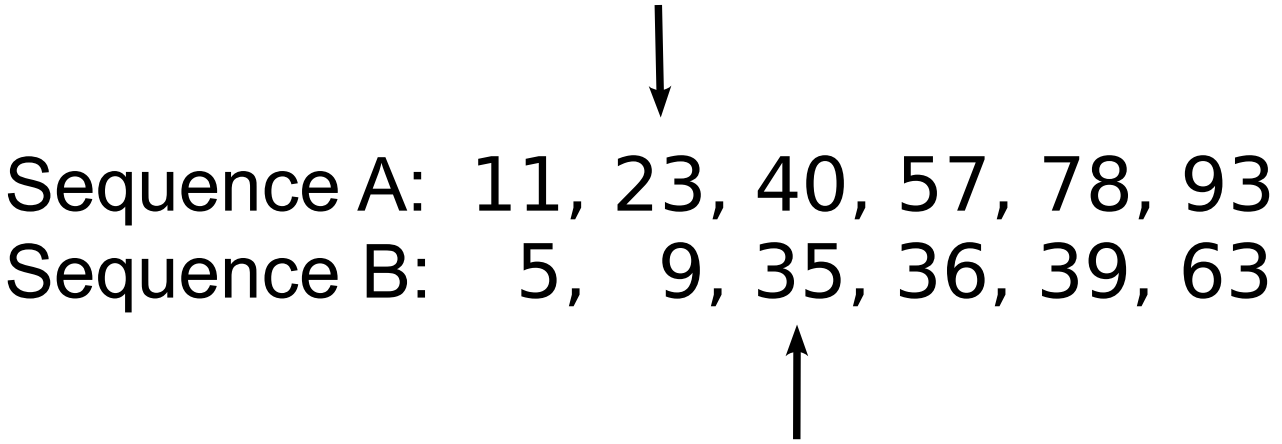
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



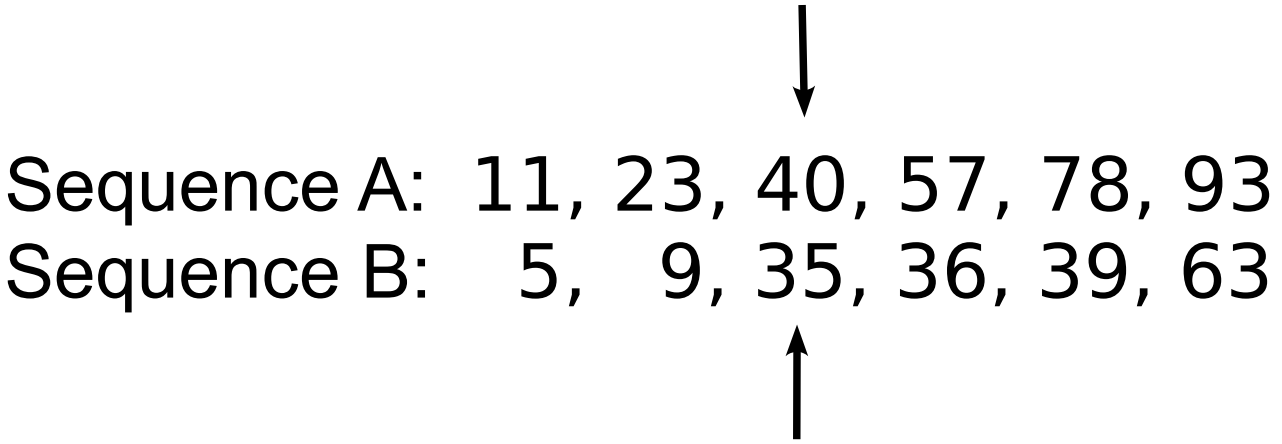
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



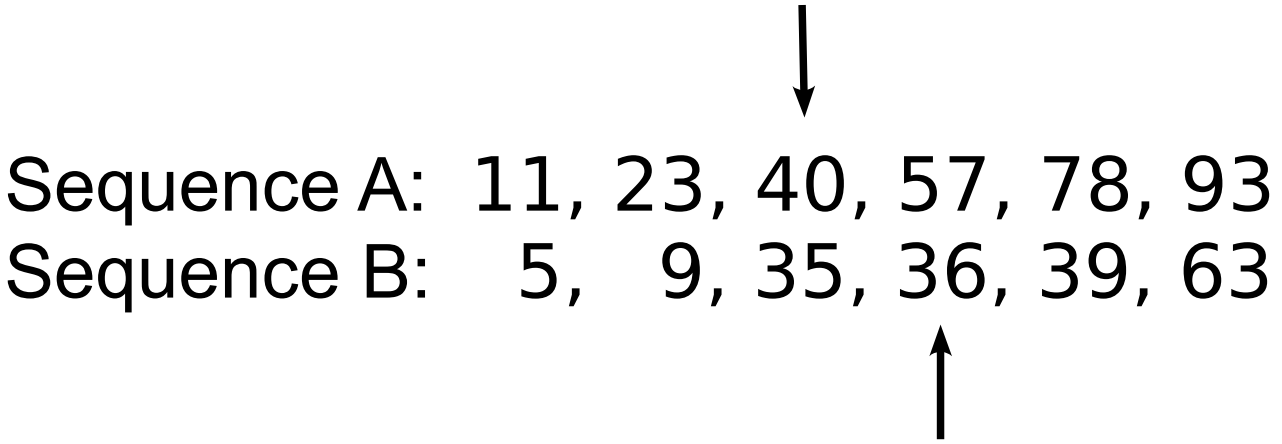
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



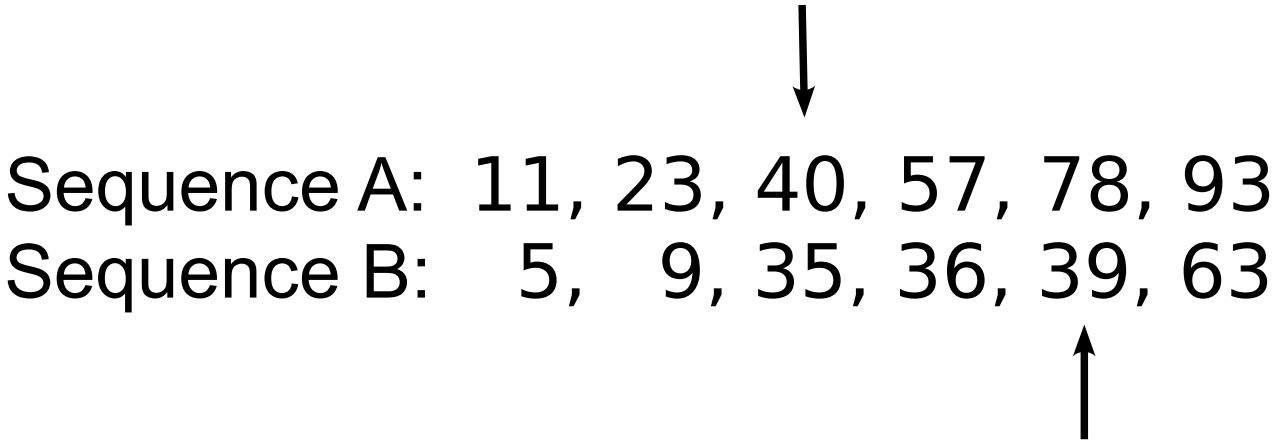
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



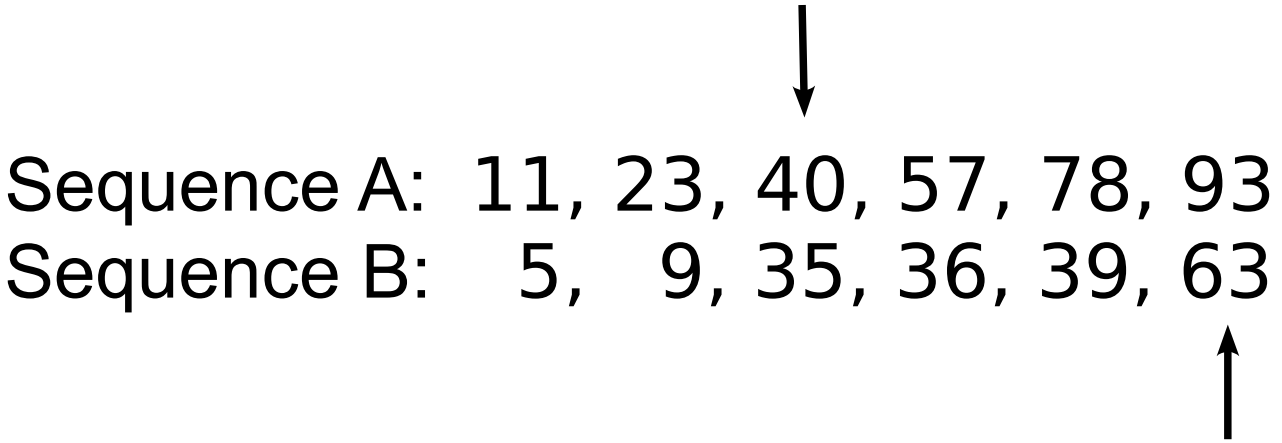
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



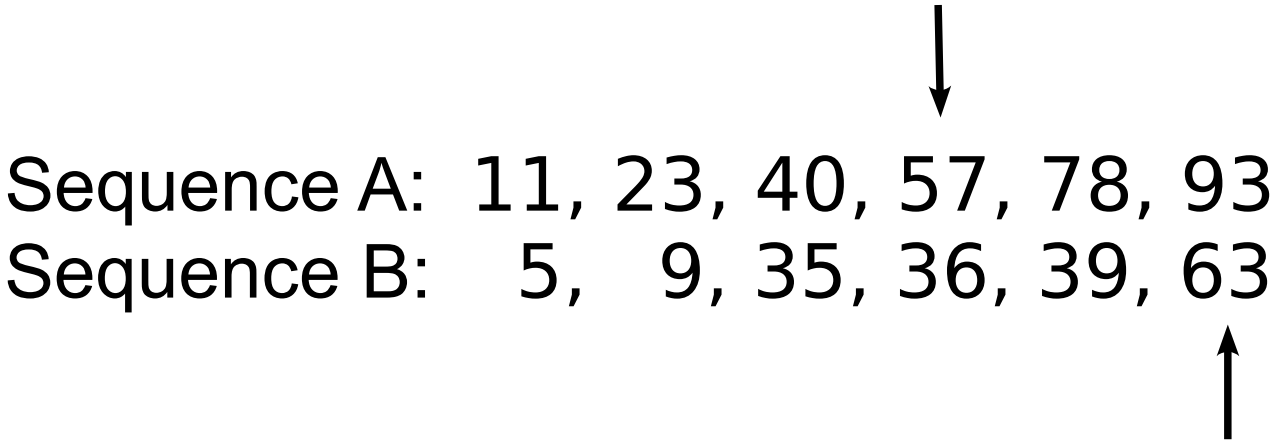
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



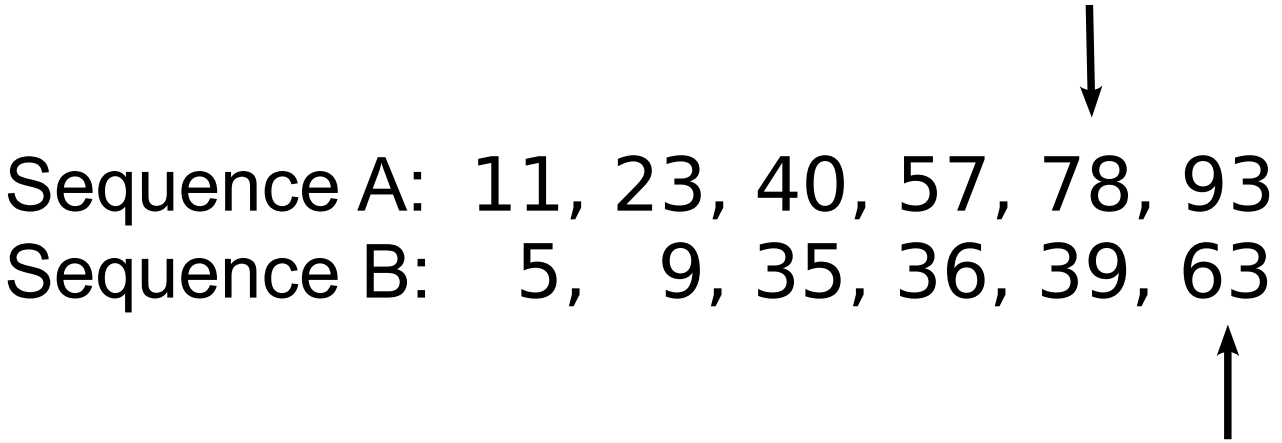
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



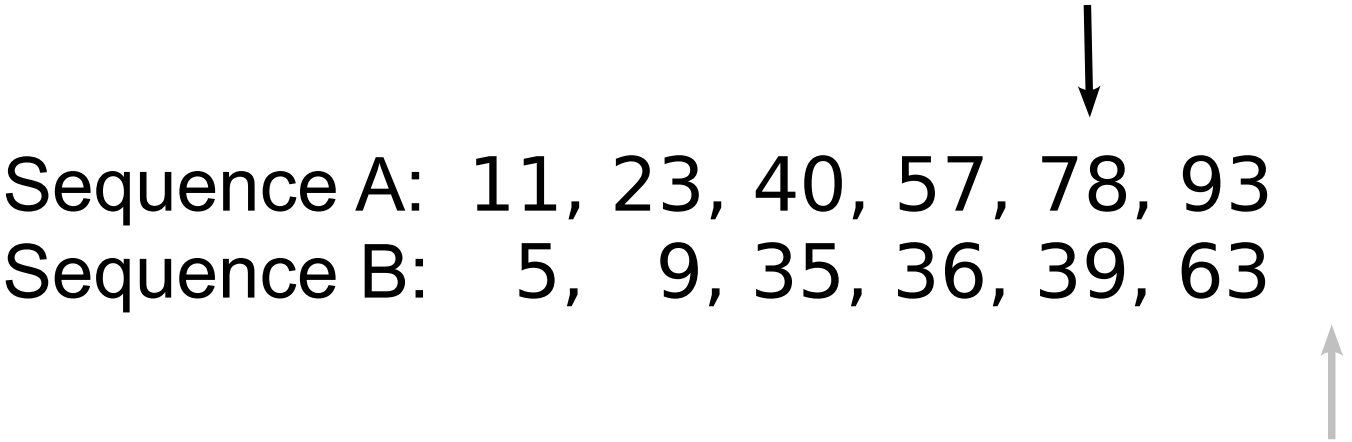
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



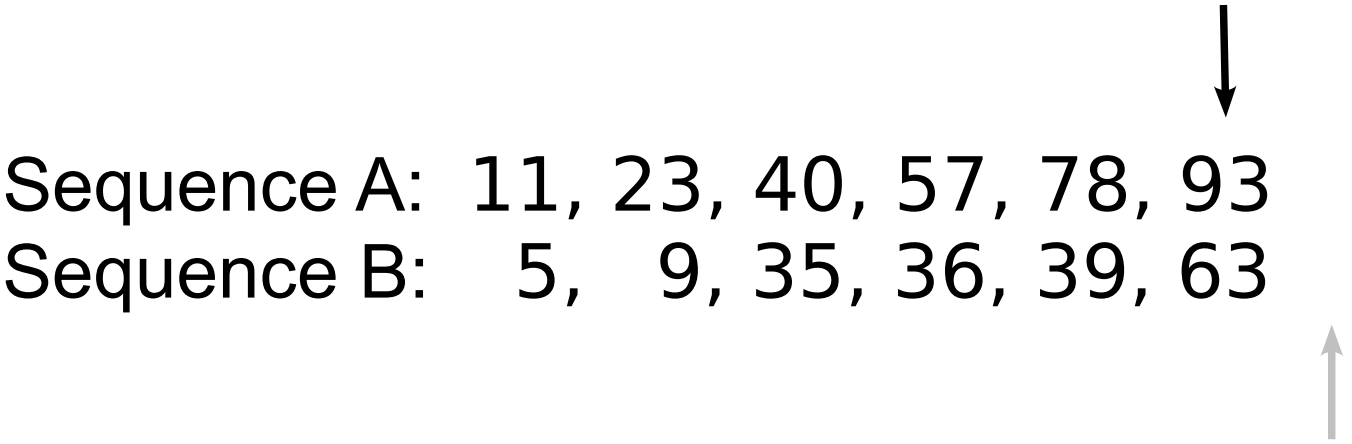
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



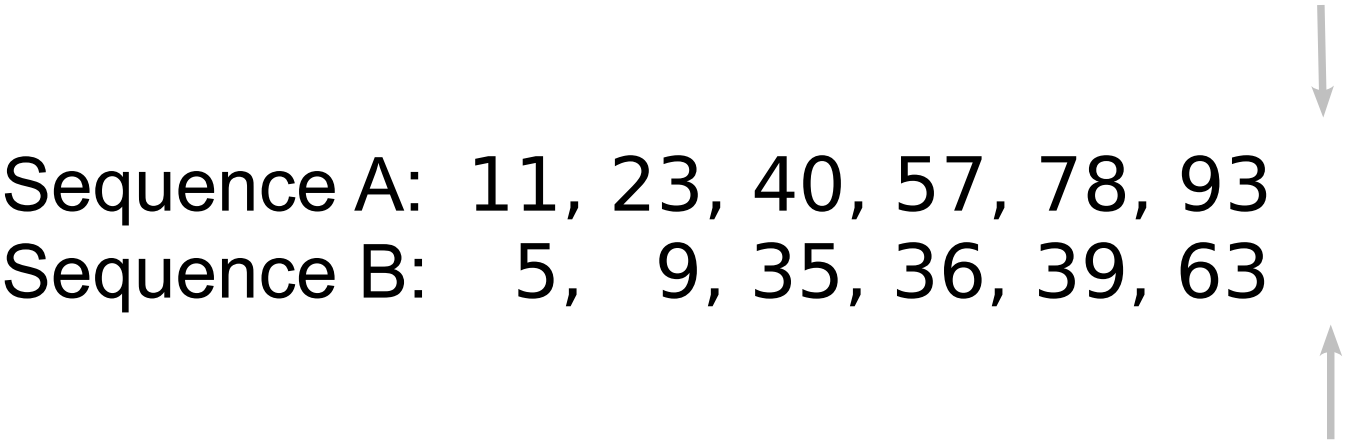
Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



Sorting

- Merging two sorted sequences into a single sorted sequence (in linear time):



Done!

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?

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?
    - Really not! The merge operation does require the extra space.

Sorting

- Now — how's this for a plot twist:
  - How do you guys think the performance of merge sort compares to the performance of, say, insertion sort, for arrays of size 10 or 20?

Sorting

- Now — how's this for a plot twist:
  - How do you guys think the performance of merge sort compares to the performance of, say, insertion sort, for arrays of size 10 or 20?
  - We have merge sort that takes $C_1 \cdot 10 \cdot \log(10)$, but because of its complexity, the constant C_1 is rather large!
  - Insertion sort takes $C_2 \cdot 10^2 = 100\ C_2$ with a C_2 considerably lower than C_1

Sorting

- Now — how's this for a plot twist:
  - Clearly, for low values of *n*, the “slow” algorithms actually outperform merge sort (the fast one).
  - Shouldn't be a surprise ... Merge sort is asymptotically faster, which is what we care about anyway .... right?

Sorting

- Now — how's this for a plot twist:
  - Clearly, for low values of *n*, the “slow” algorithms actually outperform merge sort (the fast one).
  - Shouldn't be a surprise ... Merge sort is asymptotically faster, which is what we care about anyway .... right?
- Ok, but that's not the plot twist ... Can someone see where this is going?

Sorting

- Now — how's this for a plot twist:
  - Clearly, for low values of *n*, the “slow” algorithms actually outperform merge sort (the fast one).
  - Shouldn't be a surprise ... Merge sort is asymptotically faster, which is what we care about anyway .... right?
- Ok, but that's not the plot twist ... Can someone see where this is going?
  - Hint: we want merge sort to be *as fast as possible*!

Sorting

- Now — how's this for a plot twist:
  - Clearly, for low values of *n*, the “slow” algorithms actually outperform merge sort (the fast one).
  - Shouldn't be a surprise ... Merge sort is asymptotically faster, which is what we care about anyway .... right?
- Ok, but that's not the plot twist ... Can someone see where this is going?
  - Hint: we want merge sort to be *as fast as possible*!
  - Hint 2: merge sort divides the size and sorts the two halves (recursively, of course — *how else*!!)

Sorting

- Now — how's this for a plot twist:
  - That's right !!! When the recursive process reaches a size for which insertion sort is faster, then we don't sort the sub-arrays with merge sort

Sorting

- Now — how's this for a plot twist:
  - That's right !!! When the recursive process reaches a size for which insertion sort is faster, then we don't sort the sub-arrays with merge sort (why would we, if we have another sort algorithm that is faster!!!!)
  - This threshold is typically obtained experimentally; often in the order of a few tens)

Sorting

- Now — how's this for a plot twist:
  - That's right !!! When the recursive process reaches a size for which insertion sort is faster, then we don't sort the sub-arrays with merge sort (why would we, if we have another sort algorithm that is faster!!!!)
  - This threshold is typically obtained experimentally; often in the order of a few tens)
- Final detail:
  - Does this change the asymptotic run time of merge sort? Is it no longer $\Theta(n \log n)$? What is it, if not?

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.