Quick sort

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

Quick sort

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

Quick sort

- Previously, on ECE-250...
  - Introduced sorting and related concepts
  - Discussed some of the important sorting algorithms.
  - In particular, merge sort and heap sort

Quick sort

- Previously, on ECE-250...
  - Introduced sorting and related concepts
  - Discussed some of the important sorting algorithms.
  - In particular, merge sort and heap sort
    - With merge sort, we split in two halves, sort each one, then merge the sorted halves
      - Divide-and-conquer !

Quick sort

- During today's class:
  - We'll look into quick sort

Quick sort

- During today's class:
  - We'll look into quick sort, the most widely used sorting algorithm (at least in general-purpose libraries and software out there)
    - Discuss the basic idea behind the algorithm
    - Look into its run time, with emphasis on average-case vs. Worst-case (interesting plot twist in this area!)
    - Discuss some of the strategies to avoid the worst-case
  - Work through some examples.

Quick sort

- To introduce the basic idea, let's first recall what the main problem was with merge sort...

Quick sort

- To introduce the basic idea, let's first recall what the main problem was with merge sort...
  - The merge operation can not be done in-place

Quick sort

- To introduce the basic idea, let's first recall what the main problem was with merge sort...
  - The merge operation can not be done in-place
    - This not only has the immediate implication that we require the extra space, but it also has an implication on the performance — the data is being moved a lot, and so this has a performance penalty!
      - Even more critical if the data is expensive to copy (e.g., strings, data that is really on disk or some extra-slow hardware devices)

Quick sort

- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:

Sequence A: 11, 23, 40, 57, 68, 93
Sequence B: 5, 16, 35, 46, 79, 96

Output sequence:

Quick sort

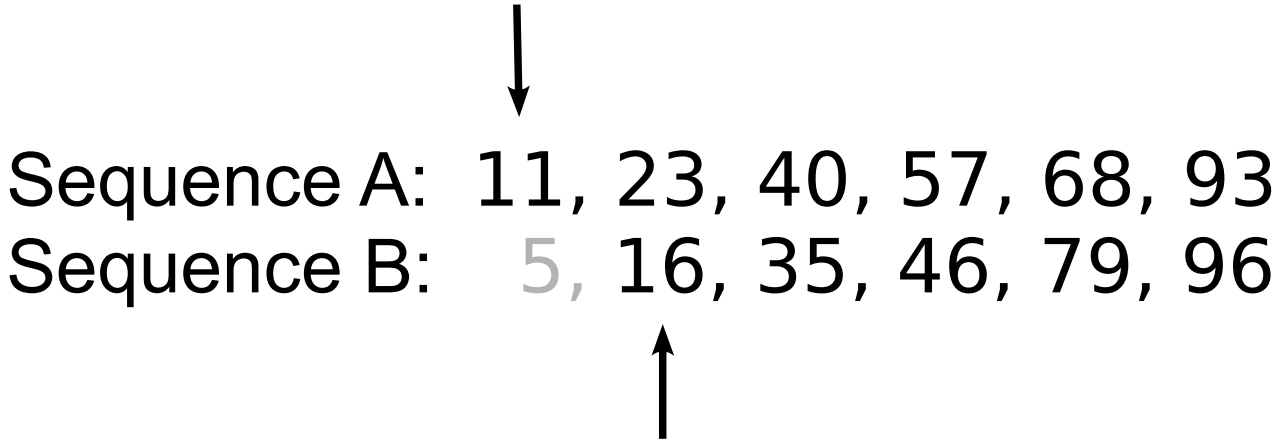
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:

Sequence A: 11, 23, 40, 57, 68, 93
Sequence B: 5, 16, 35, 46, 79, 96

Output sequence:

Quick sort

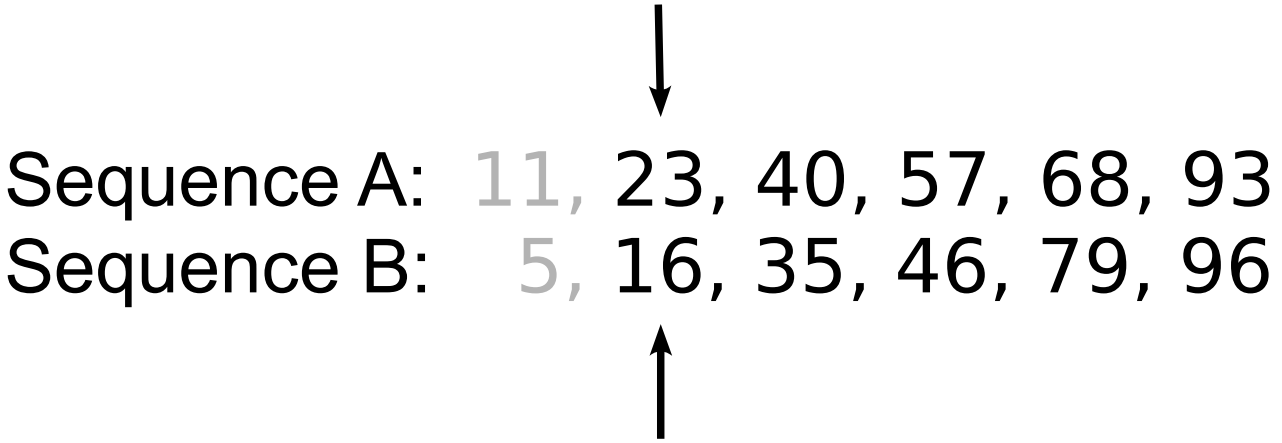
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5

Quick sort

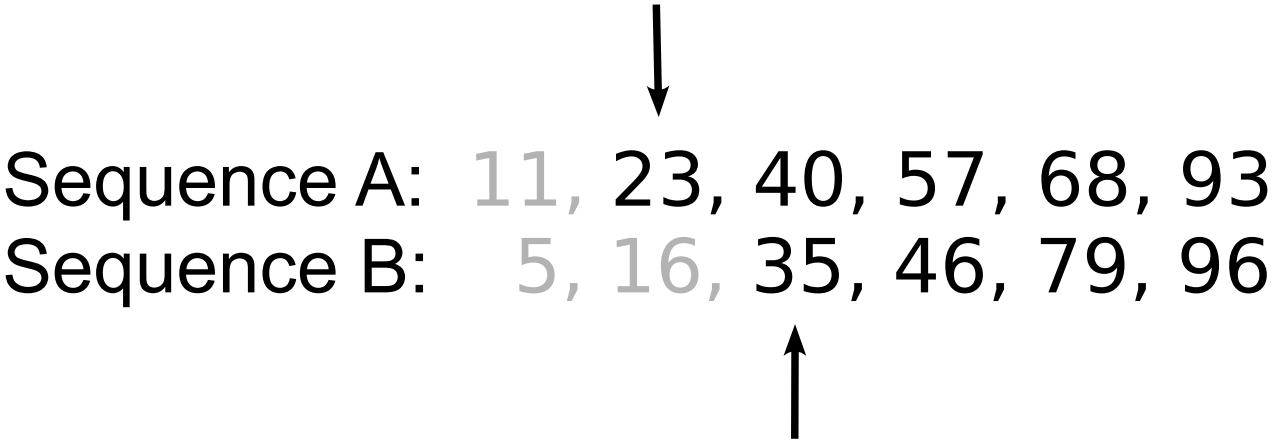
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5, 11

Quick sort

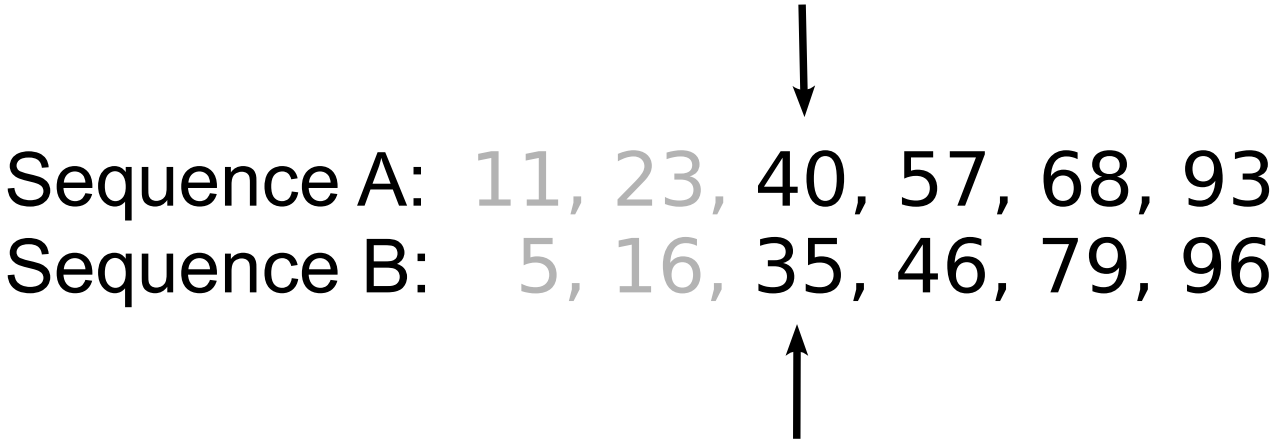
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5, 11, 16

Quick sort

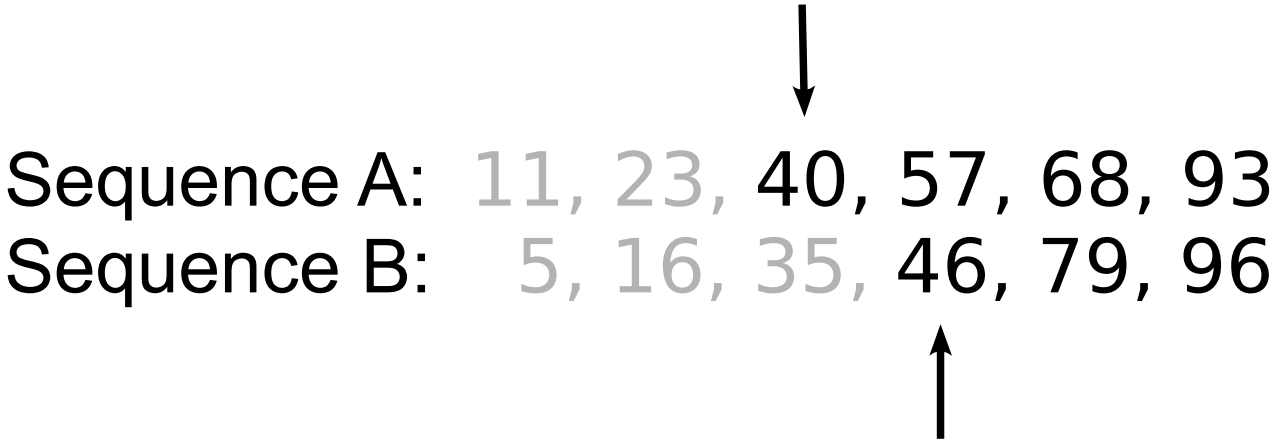
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5, 11, 16, 23

Quick sort

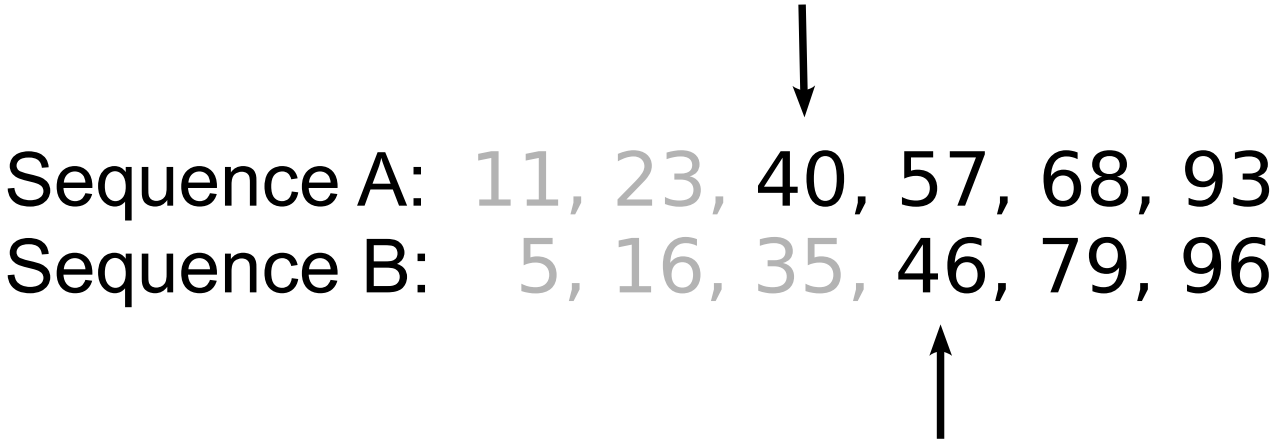
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5, 11, 16, 23, 35

Quick sort

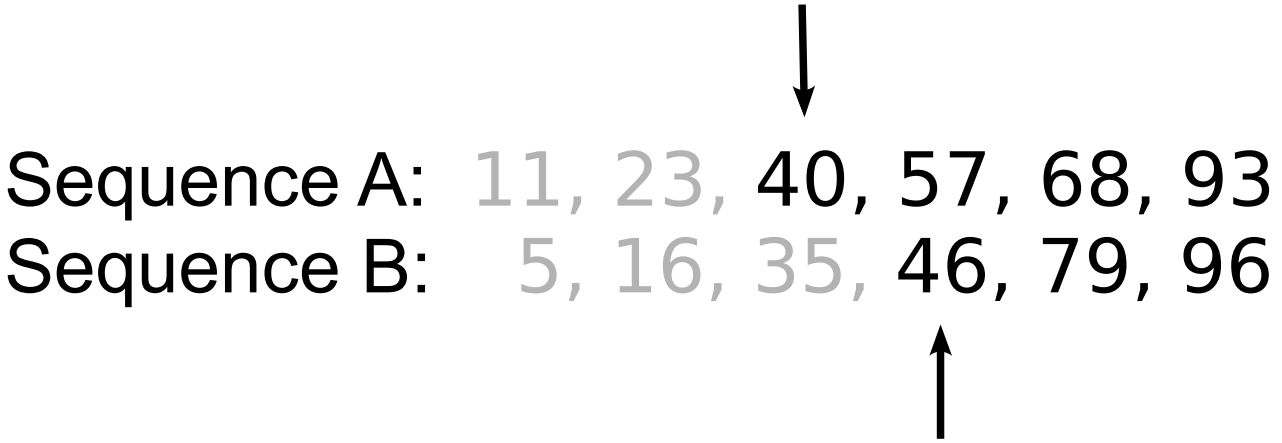
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



Output sequence: 5, 11, 16, 23, 35 · · · etc.

Quick sort

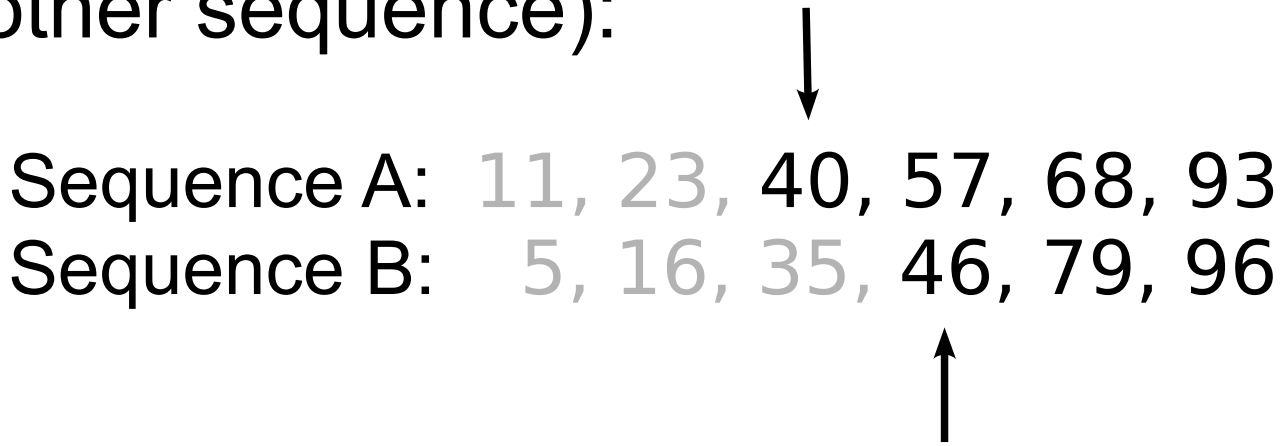
- The obvious question is: can we do a merge sort without requiring extra storage?
  - Let's try merging the following two sorted sequences:



We might be tempted to think of the items at the left of the arrows as available space ... But ...

Quick sort

- We don't have room in either sequence to place the output values from the other sequence (we have no control on how the pointer from one sequence advances with respect to the one for the other sequence):



Quick sort

- Let's make the long story short, and just face it: it *can not* be done!!!

Quick sort

- Bottom line with merge sort — the problem seems to be due to the fact that we first sort each half, then combine the results.

Quick sort

- Bottom line with merge sort — the problem seems to be due to the fact that we first sort each half, then combine the results.
  - And yes, one might think "*well, duh! of course we do that — what else are we going to do?*"

Quick sort

- Bottom line with merge sort — the problem seems to be due to the fact that we first sort each half, then combine the results.
  - And yes, one might think "*well, duh! of course we do that — what else are we going to do?*"
  - But that's actually an excellent question! Is there something else we could do?

Quick sort

- Bottom line with merge sort — the problem seems to be due to the fact that we first sort each half, then combine the results.
  - And yes, one might think "*well, duh! of course we do that — what else are we going to do?*"
  - But that's actually an excellent question! Is there something else we could do?
    - Perhaps going the other way around — could we do some processing first such that the elements end up in a way that when we recursively sort each half we're done?

Quick sort

- Let's reformulate that question:
- Suppose that I give you sequence A and sequence B, one after the other one, as shown below:

$$\underbrace{a_1, a_2, \cdots, a_{n-1}, a_n}_{\text{Sequence } A} \quad \underbrace{b_1, b_2, \cdots, b_{n-1}, b_n}_{\text{Sequence } B}$$

Quick sort

- Let's reformulate that question:
- Suppose that I give you sequence A and sequence B, one after the other one, as shown below:

$$\underbrace{a_1, a_2, \cdots, a_{n-1}, a_n}_{\text{Sequence } A} \quad \underbrace{b_1, b_2, \cdots, b_{n-1}, b_n}_{\text{Sequence } B}$$

- And suppose I told you that if we sort sequence A and sort sequence B, then the complete sequence will be sorted — what does that tell you about the values in both sequences?

Quick sort

- This is the central aspect behind quicksort's idea/functionality:

Quick sort

- This is the central aspect behind quicksort's idea/functionality:
  - If every value in sequence A is less than every value in sequence B, then when we (recursively) sort each sequence, we would be done (that is, the entire sequence would be sorted)

Quick sort

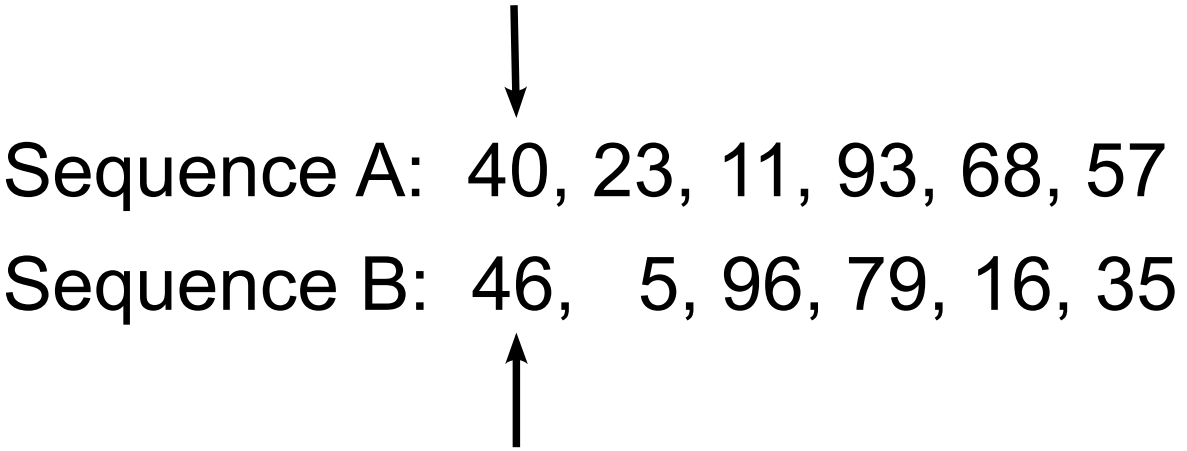
- Let's try the following (hopefully in linear time):
  - Back to sequences A and B from earlier, we'll take a value (say, 45) to compare — if a value from sequence A is > 45, then it should be moved to B, and if a value from sequence B is < 45, then it should be moved to A.

Quick sort

- Let's try the following (hopefully in linear time):
  - Back to sequences A and B from earlier, we'll take a value (say, 45) to compare — if a value from sequence A is > 45, then it should be moved to B, and if a value from sequence B is < 45, then it should be moved to A.
  - We can definitely do this in linear time — scan each of the arrays for elements matching the condition, then swap them!

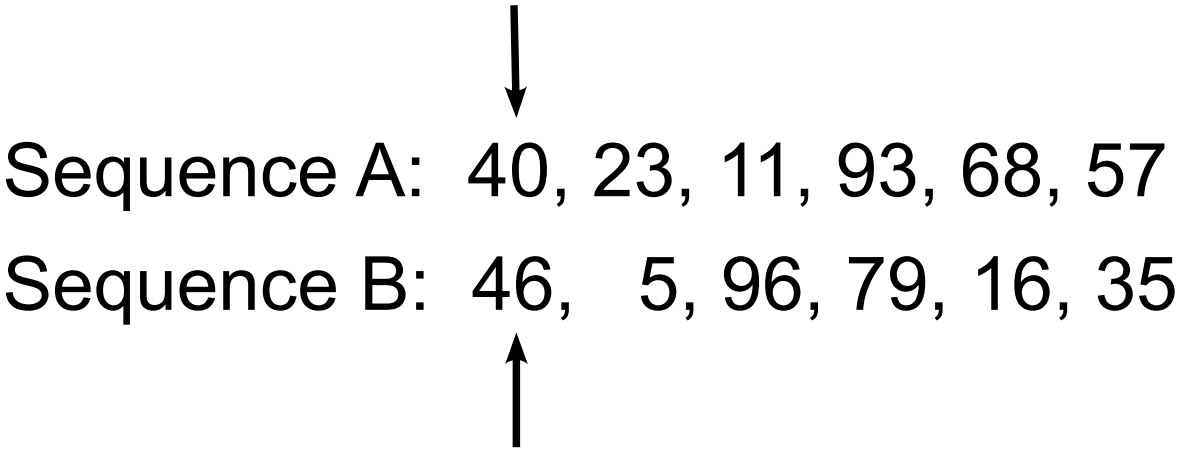
Quick sort

- Partitioning process:



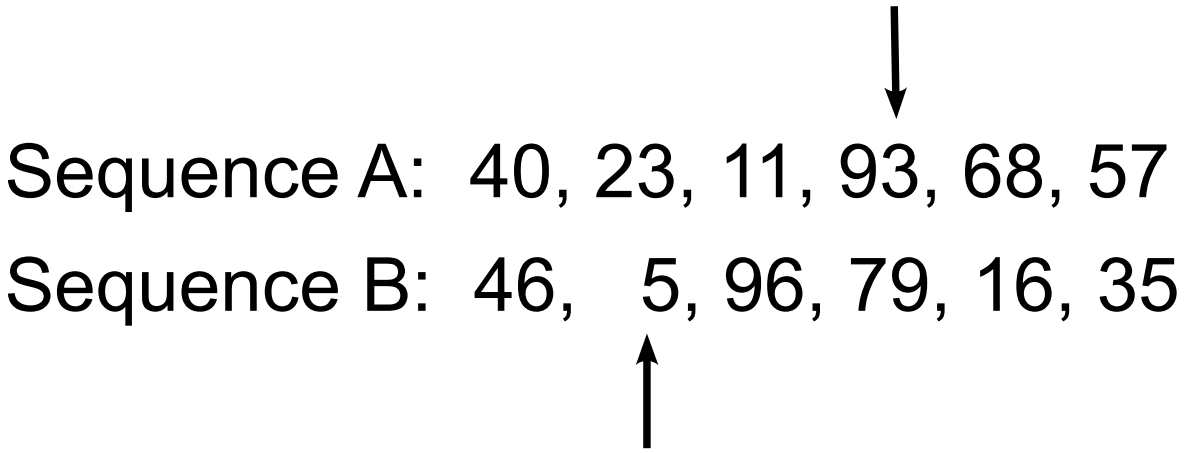
Quick sort

- Partitioning process:
  - Look for a value > 45 in A, and a value < 45 in B:



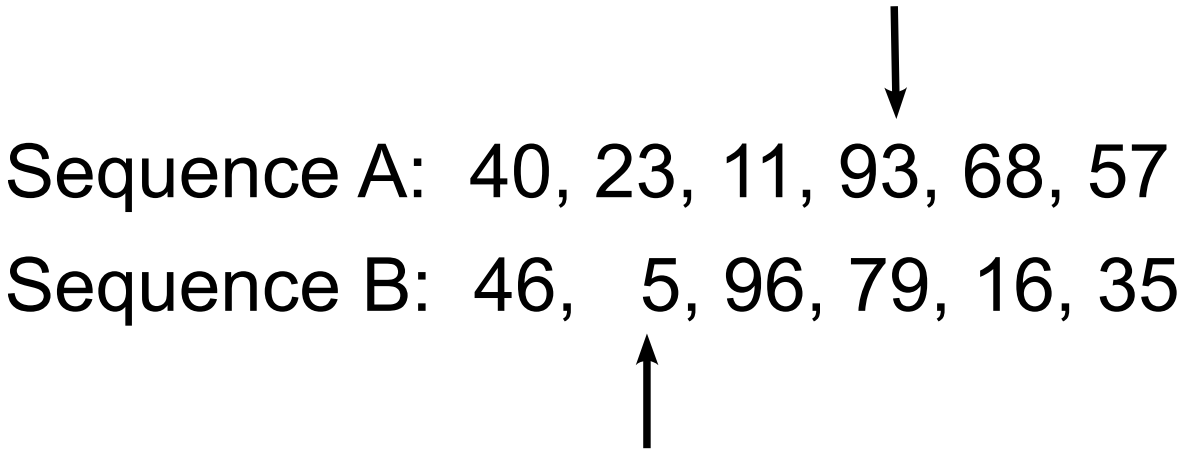
Quick sort

- Partitioning process:
  - Look for a value > 45 in A, and a value < 45 in B:



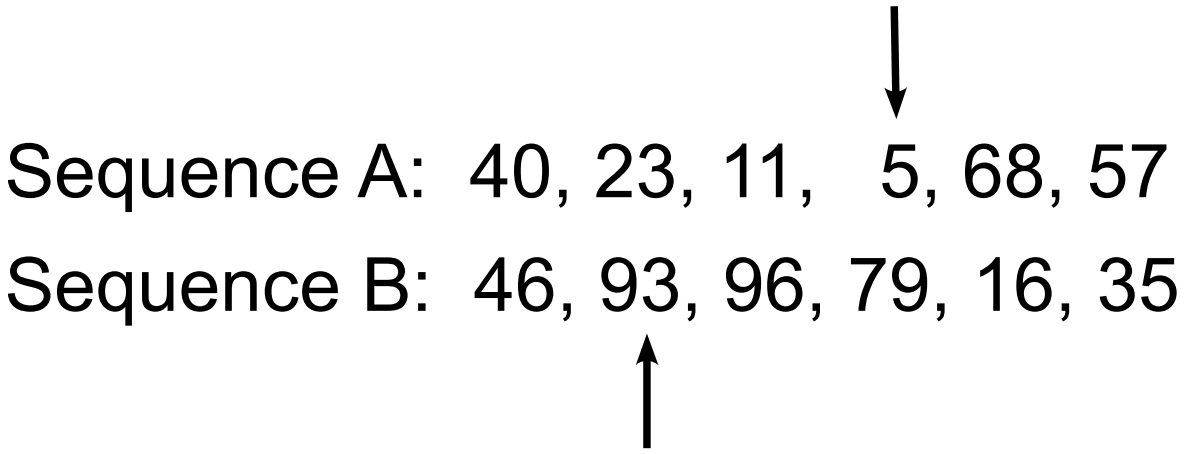
Quick sort

- Partitioning process:
  - Look for a value > 45 in A, and a value < 45 in B:
    - Then swap them



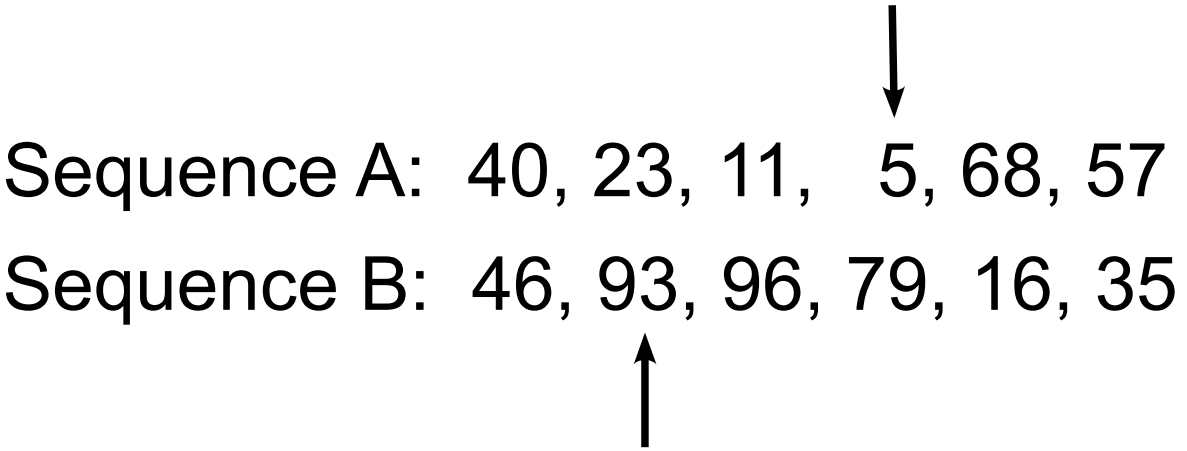
Quick sort

- Partitioning process:
  - Look for a value > 45 in A, and a value < 45 in B:
    - Then swap them



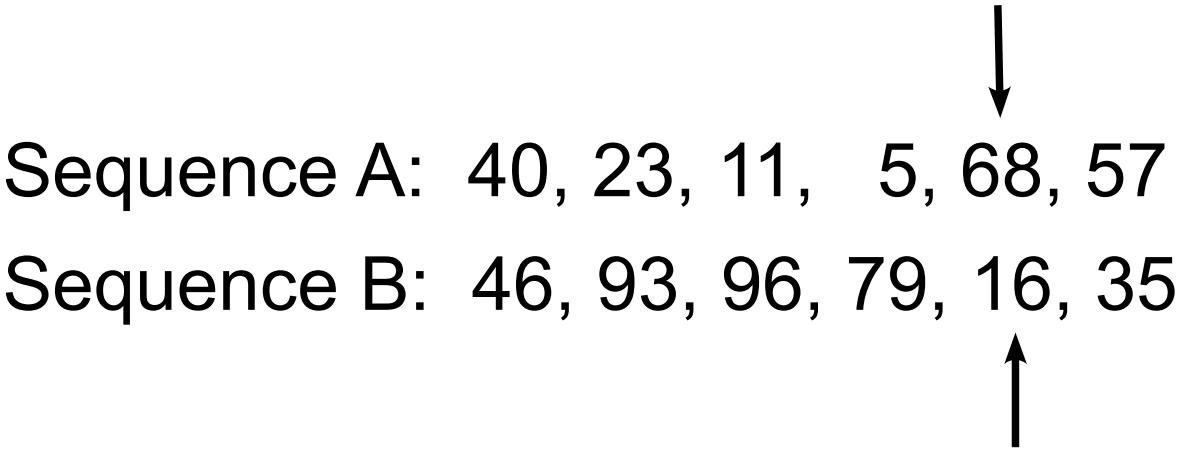
Quick sort

- Partitioning process:
  - Next...



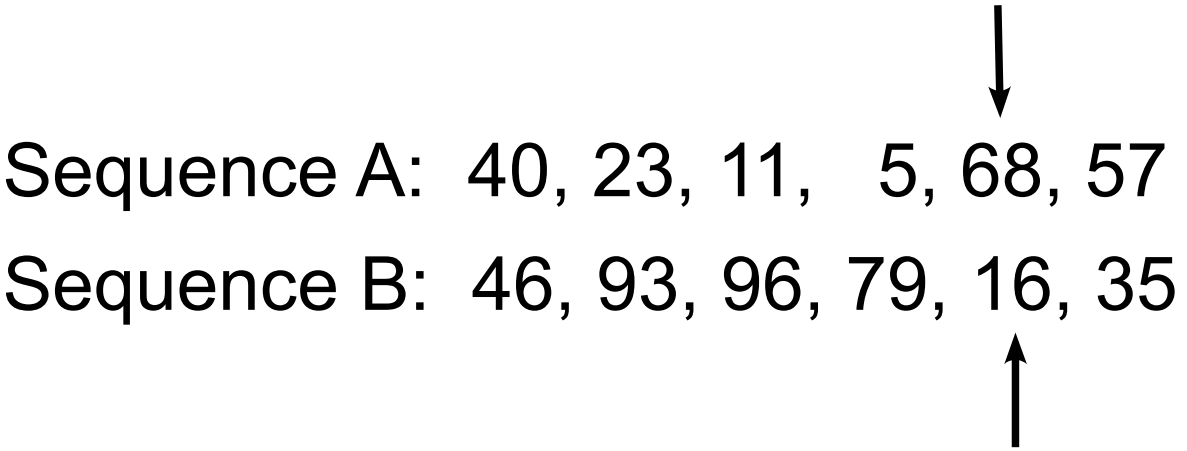
Quick sort

- Partitioning process:
  - Next...



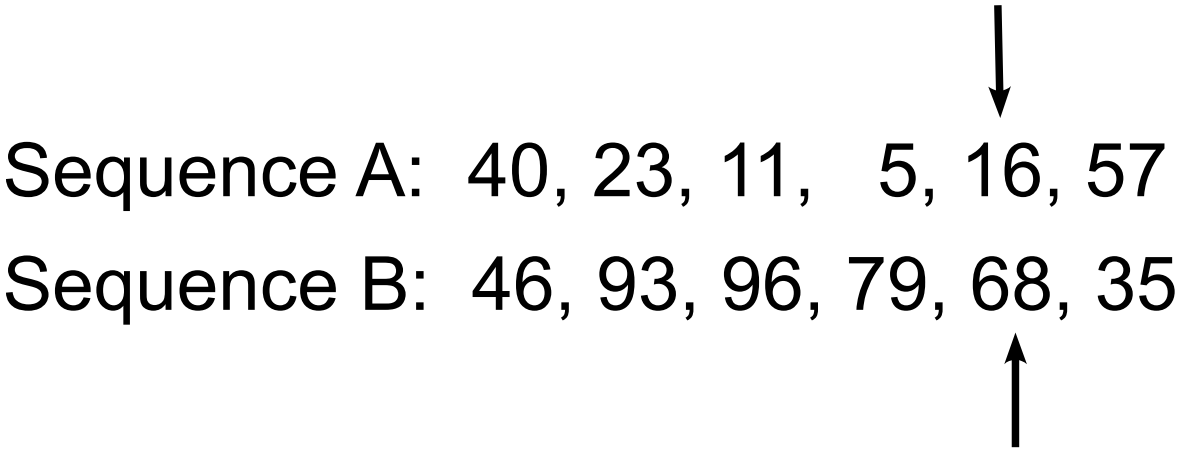
Quick sort

- Partitioning process:
  - Next... then swap them



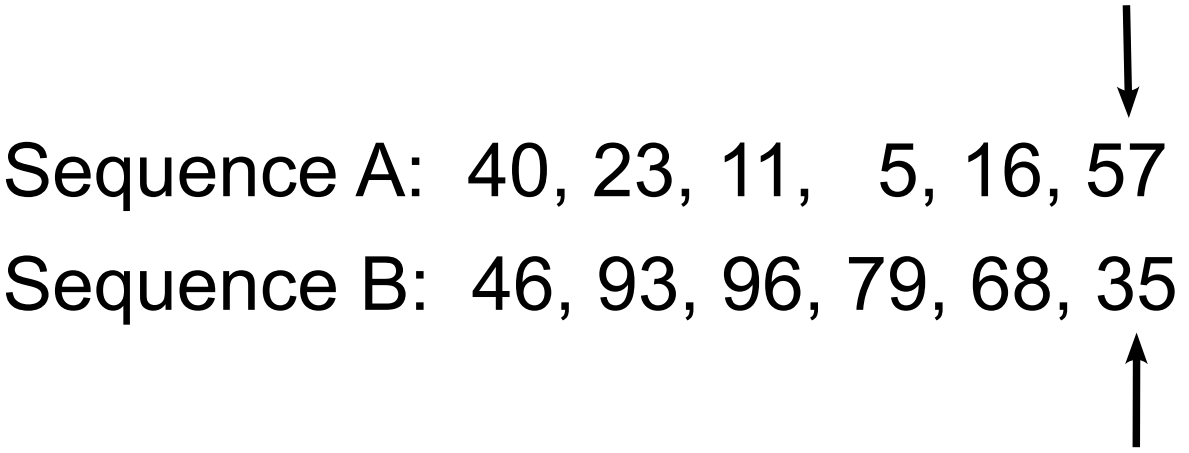
Quick sort

- Partitioning process:
  - Next... then swap them



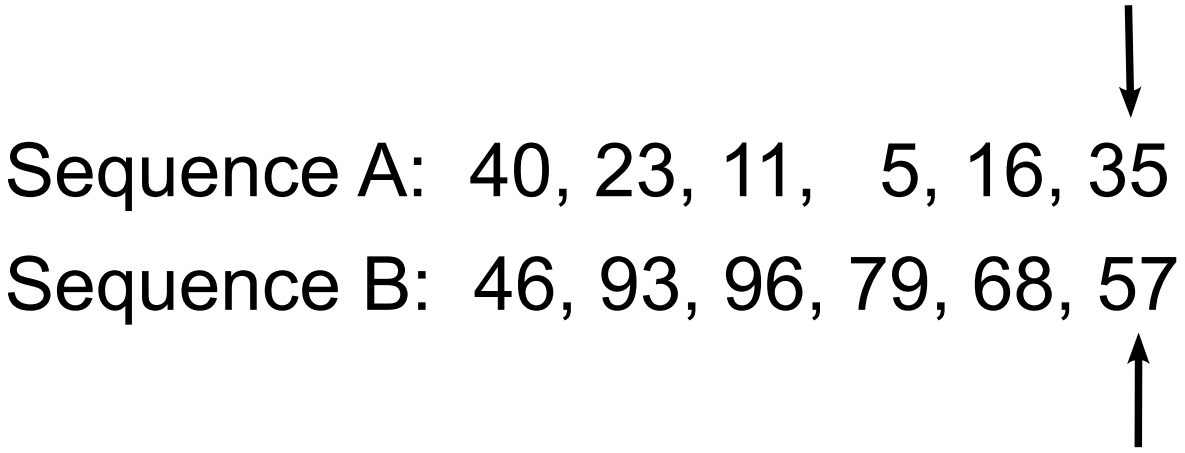
Quick sort

- Partitioning process:
  - Next...



Quick sort

- Partitioning process:
  - Next... then swap them



Quick sort

- Partitioning process:
  - Done! Now every value in sequence A is less than every value in sequence B (we now sort A, then sort B, and then sorting of the whole sequence is completed!)

Sequence A: 40, 23, 11, 5, 16, 35

Sequence B: 46, 93, 96, 79, 68, 57

Quick sort

- Anyone sees the *severe* flaw with this example?

Sequence A: 40, 23, 11, 5, 16, 35

Sequence B: 46, 93, 96, 79, 68, 57

Quick sort

- How did (or how could) we know that we had to compare against 45 to swap?

Quick sort

- How did (or how could) we know that we had to compare against 45 to swap?
- Or equivalently, once picked a value, how do we know that we have room enough in sequence A for all elements < the value and in sequence B for all elements > the value?

Quick sort

- How did (or how could) we know that we had to compare against 45 to swap?
- Or equivalently, once picked a value, how do we know that we have room enough in sequence A for all elements < the value and in sequence B for all elements > the value?
  - The answer is: *we don't* know that we'll have enough room — because we won't have enough room in general.

Quick sort

- If we had chosen, say, 20, then we would have had 3 values that had to go in sequence A, and 9 values in sequence B

Quick sort

- If we had chosen, say, 20, then we would have had 3 values that had to go in sequence A, and 9 values in sequence B
  - Ermm... wait! Why is that a problem??

Quick sort

- If we had chosen, say, 20, then we would have had 3 values that had to go in sequence A, and 9 values in sequence B
  - Ermm... wait! Why is that a problem??
  - Could we not just recursively sort the 3-element sequence, then sort the 9-element sequence?

Quick sort

- We have two obstacles if we do that:
  - If we pick some value without knowing the sizes of the two resulting subsequences, how do we know where to swap?

Quick sort

- We have two obstacles if we do that:
  - If we pick some value without knowing the sizes of the two resulting subsequences, how do we know where to swap? (we'll see in a minute that this one can be addressed rather easily)

Quick sort

- We have two obstacles if we do that:
  - If we pick some value without knowing the sizes of the two resulting subsequences, how do we know where to swap? (we'll see in a minute that this one can be addressed rather easily)
  - The other problem is: don't we need to partition into two equal size chunks to obtain $\Theta(n \log n)$ run time?

Quick sort

- The answer is, no — with partitions of unequal size, we still obtain $\Theta(n \log n)$, just with a larger proportionality constant (so, it *is* slower — just not in terms of asymptotic notation)

Quick sort

- The answer is, no — with partitions of unequal size, we still obtain $\Theta(n \log n)$, just with a larger proportionality constant (so, it *is* slower — just not in terms of asymptotic notation)
  - Provided that the sizes of the partition are proportional to n — that is, so long as the partitions have sizes an and $(1-a)n$.

Quick sort

- The answer is, no — with partitions of unequal size, we still obtain $\Theta(n \log n)$, just with a larger proportionality constant (so, it *is* slower — just not in terms of asymptotic notation)
  - Provided that the sizes of the partition are proportional to n — that is, so long as the partitions have sizes an and $(1-a)n$.
  - What if they don't? In particular, what if the partitions, for some unlucky coincidence, always end up being size 1 and size $n-1$??

Quick sort

- In fact, if we're doing a *worst-case* run time analysis, we would have to consider this outcome — partitions are always 1 and $n-1$.

Quick sort

- In fact, if we're doing a *worst-case* run time analysis, we would have to consider this outcome — partitions are always 1 and $n-1$.
  - What's the run time of this?

Quick sort

- In fact, if we're doing a *worst-case* run time analysis, we would have to consider this outcome — partitions are always 1 and $n-1$.
  - What's the run time of this?
  - Again we recognize the arithmetic sum — a sort of size n leads to one of size $n-1$, plus one of size $n-2$, plus $\cdots$ all the way down to 1.
  - And this is $\Theta(n^2)$

Quick sort

- So, perhaps the more interesting question is:

Quick sort

- So, perhaps the more interesting question is:
  - Why on earth are we talking about this algorithm, that we're seeing has quadratic run time ??!!!

Quick sort

- So, perhaps the more interesting question is:
  - Why on earth are we talking about this algorithm, that we're seeing has quadratic run time ??!!!
  - Let's say that this is definitely one of the remarkable aspects of this algorithm: it does have worst-case run time $\Theta(n^2)$, but:

Quick sort

- So, perhaps the more interesting question is:
  - Why on earth are we talking about this algorithm, that we're seeing has quadratic run time ??!!!
  - Let's say that this is definitely one of the remarkable aspects of this algorithm: it does have worst-case run time $\Theta(n^2)$, but:
    - On average, quick sort outperforms every other sorting algorithm known!

Quick sort

- So, perhaps the more interesting question is:
  - Why on earth are we talking about this algorithm, that we're seeing has quadratic run time ??!!!
  - Let's say that this is definitely one of the remarkable aspects of this algorithm: it does have worst-case run time $\Theta(n^2)$, but:
    - On average, quick sort outperforms every other sorting algorithm known!
    - Equally (or more) importantly: if implemented properly, we can ensure the chances of hitting the worst-case to be essentially negligible.

Quick sort

- Just one more detail before we're ready to see quick sort in action!
- Let's look at the notion of the *median* of a set of values.

Quick sort

- Just one more detail before we're ready to see quick sort in action!
- Let's look at the notion of the *median* of a set of values.
  - The *median* is a statistical measure of a set of values, somewhat similar to the *mean*, or average, but quite interesting:
  - Given a set of values $\{x_1, x_2, \cdots, x_{n-1}, x_n\}$ the median is one of the values in the sequence, x_m, such that there are as many elements $x_i < x_m$ as there are elements $x_k > x_m$

Quick sort

- Visually, this notion of the median corresponds to the following: place the elements in order; then, pick the middle element.

Quick sort

- Visually, this notion of the median corresponds to the following: place the elements in order; then, pick the middle element.
- The problem is, algorithms to find the median are not very efficient — in many cases, one simply sorts and then picks the middle element.
  - So, if we're needing the median as part of a sorting procedure, we're kind of stuck, like a dog chasing its tail !!

Quick sort

- What about an approximation of the median??
  - Something that, on average, is close to the median.
  - As it turns out, this is quite useful for quicksort !!

Quick sort

- Let's do the following experiment:
  - We'll work in groups of three or four; you'll shuffle a quarter of a deck of cards, with values from 1 to 13 (obviously, we associate A ↔ 1, J ↔ 11, Q ↔ 12, and K ↔ 13)
  - Now, pick the first, last, and middle cards, and you'll write down the median of those three cards (for example, if you get 8, 3, and Q, the median is 8)

Quick sort

- At the time that I'm writing this, I don't have the results of the experiment, but it will most likely be something in the range of an average of 2 or 3 of deviation with respect to the true median (which is 7 in this case).
  - That means that on average, we're splitting the range into 65% − 35% chunks; not too bad in terms of the performance that we get.
    - We want something that is as close as possible to the actual median, so that the partition is as close as possible to 50% − 50% .... But we also want something that is efficient!

Quick sort

- Side note — as we all saw during class, the average deviation was 2.11
  - Indeed, something close to a 65% − 35% partitioning on average.

Quick sort

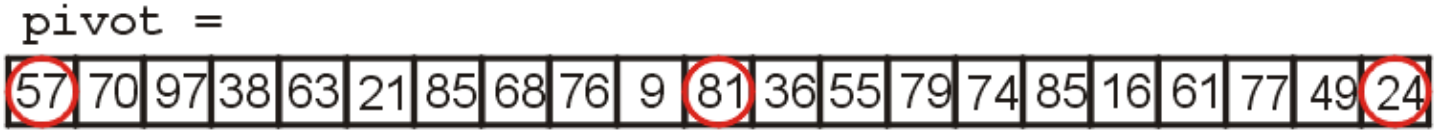
- We're ready to see quick sort in action!
  - Given the sequence of *n* elements, we choose, using perhaps the *median of three*, the element called the *pivot*. (the value that will be used to compare and determine when to swap values)
    - Thus, the value that will determine the resulting partition.

Quick sort

- We're ready to see quick sort in action!
  - Given the sequence of *n* elements, we choose, using perhaps the *median of three*, the element called the *pivot*. (the value that will be used to compare and determine when to swap values)
    - Thus, the value that will determine the resulting partition.
  - Then, we start at the beginning looking for values that should go to the second chunk, and at the end looking for values that should go to the first chunk, and we swap when we find them.

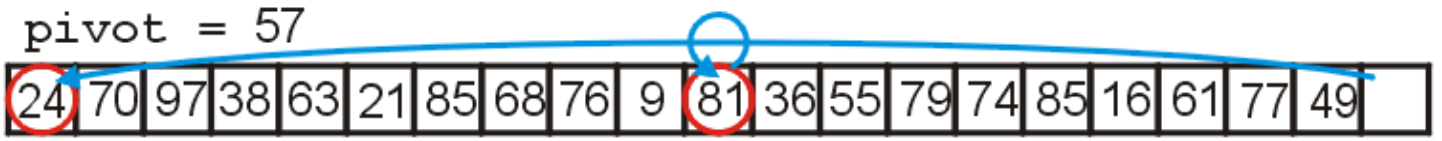
Quick sort

- Consider the following array, and inspect the first, middle, and last elements:



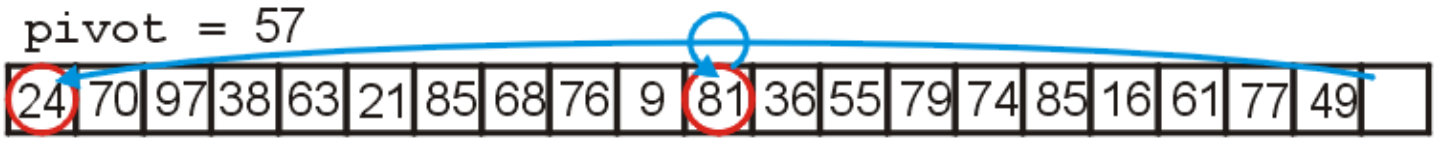
Quick sort

- We select 57 (the median of {57,81,34}) as the pivot:
  - We send the lower value to the beginning, and leave the pivot out of this for the moment...



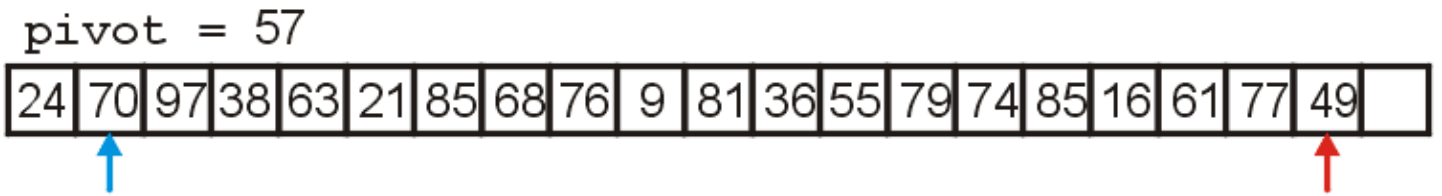
Quick sort

- Now we start scanning from the second location looking for values > 57, and from the second-to-last, searching backwards for values < 57



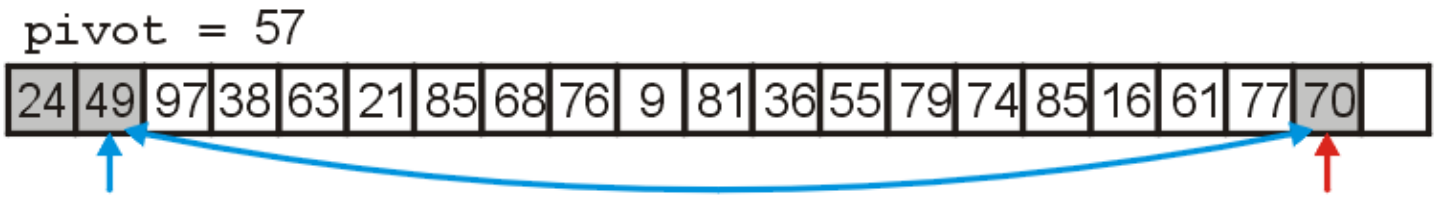
Quick sort

- We find:
  - 70 > 57, and
  - 49 < 57



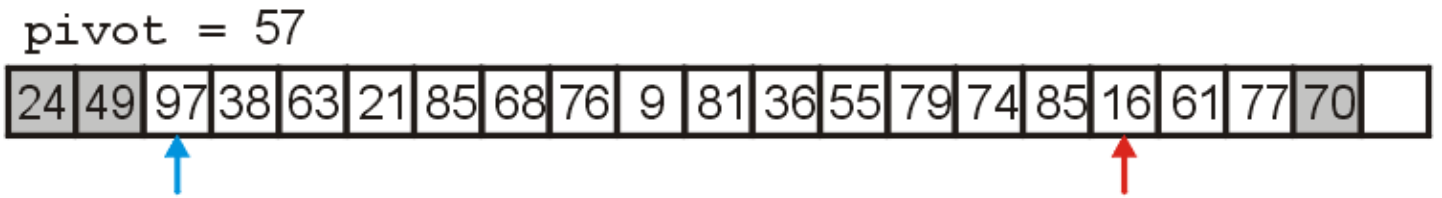
Quick sort

- So we swap them:



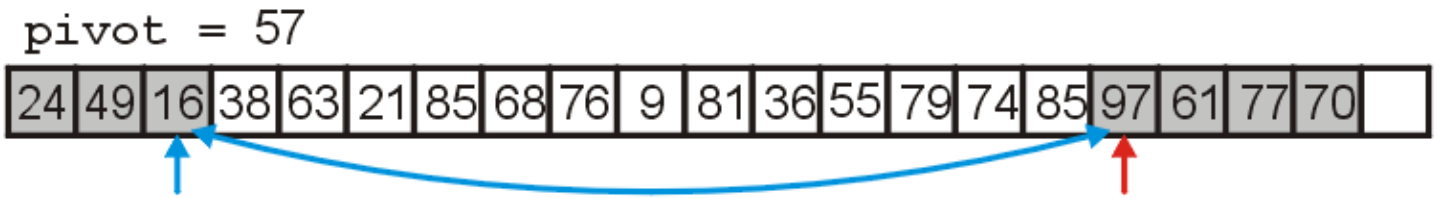
Quick sort

- Then we scan forward until we find 97 > 57 and search backward until we find 16 < 57



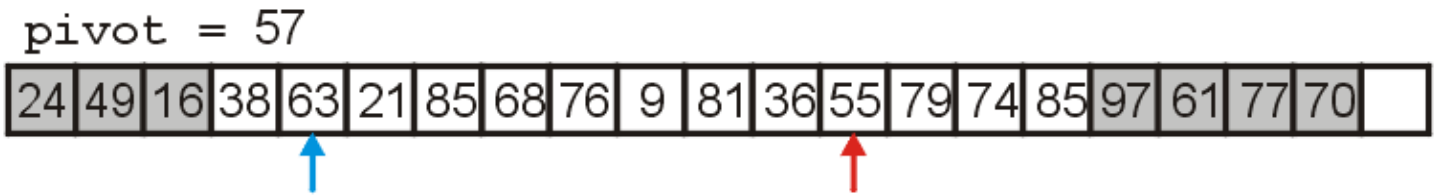
Quick sort

- So we swap them.



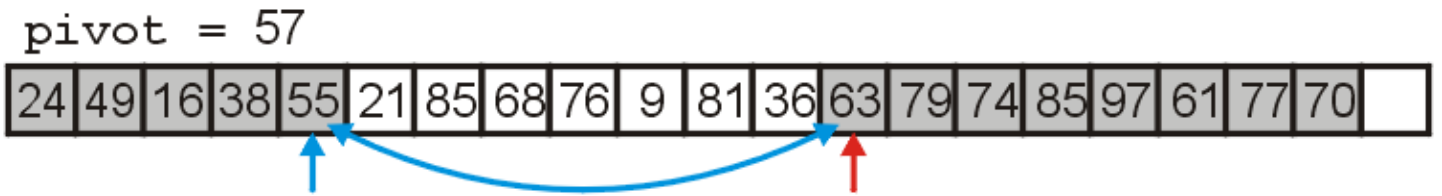
Quick sort

- Then continue ...



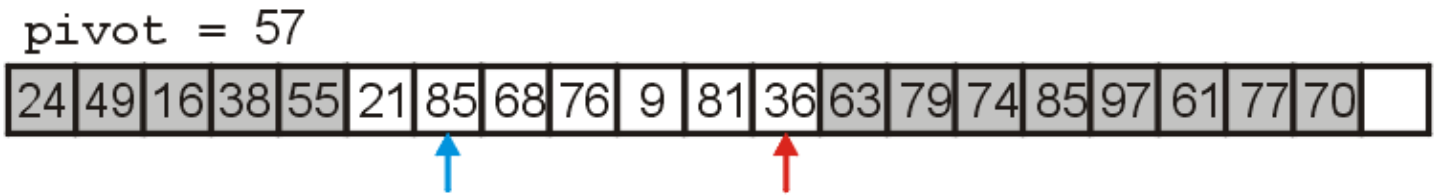
Quick sort

- Then continue ...



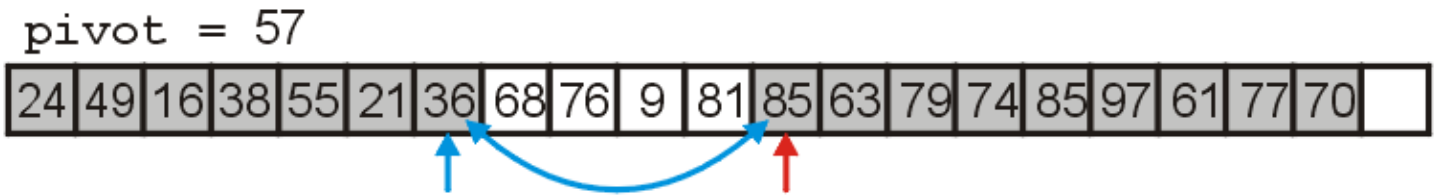
Quick sort

- Then continue ...



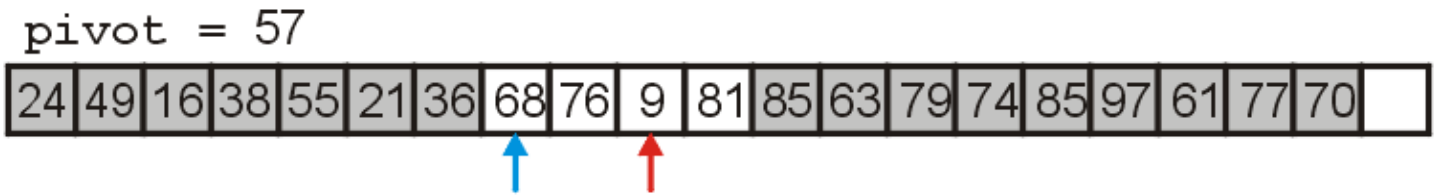
Quick sort

- Then continue ...



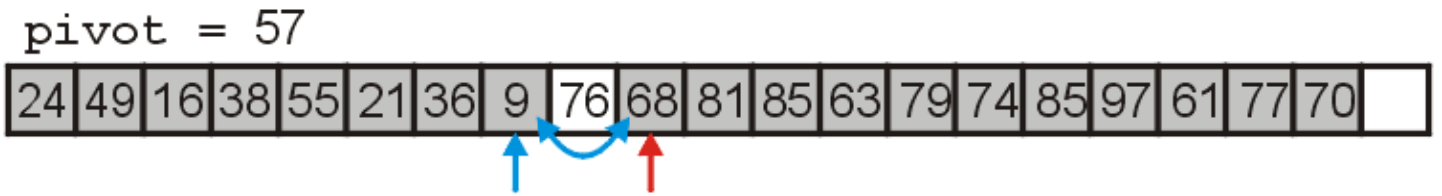
Quick sort

- Then continue ...



Quick sort

- Then continue ...

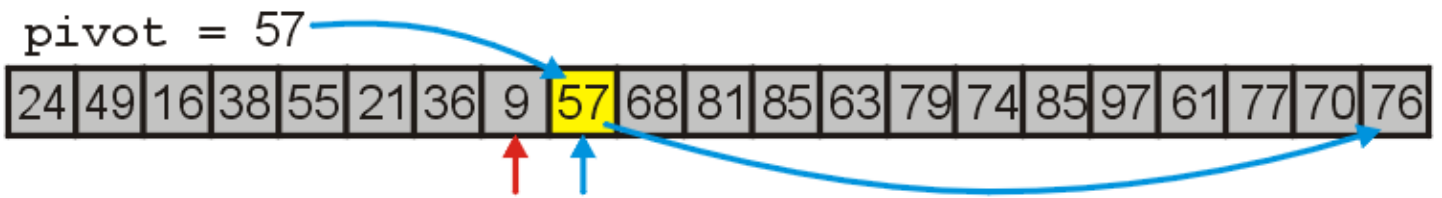


Quick sort

- We search forward until finding 76 > 57 and search backward until we find 9 < 57
  - But now the indices are reversed, so that means that we completed the loop (i.e., indices in the wrong order is the stop condition for the loop).

Quick sort

- We move the element at the larger index to the end, and place the pivot at the empty location in the middle.



Quick sort

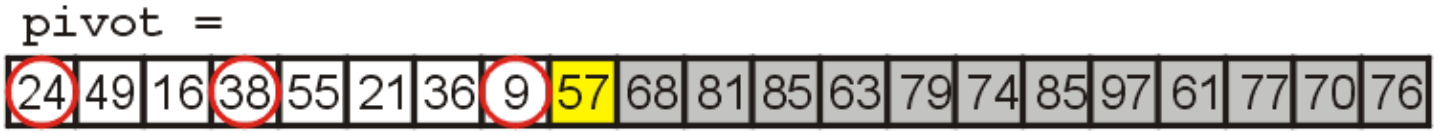
- Now the pivot, 57, is at the right position, and everything before that position it is less than 57, and everything after that position is greater than 57.

Quick sort

- Now the pivot, 57, is at the right position, and everything before that position it is less than 57, and everything after that position is greater than 57.
  - If we now (recursively) sort the first chunk, then the second chunk, we end up with a sorted array!

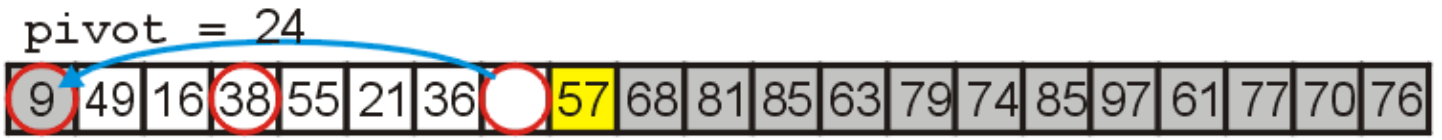
Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:



Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:



Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:

Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:

Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:
  - Indices are reversed, so we're done with this loop.

Quick sort

- For the first chunk (elements *before* 57, not including it), things would go like this:
  - Send 38 (the element at the larger index) to the last position, and the pivot, 24, to the middle position.

Quick sort

- Etc.

Quick sort

- Not much of a plot twist this time (Heap sort did it !), when we narrow down the chunk to a small size, then we use insertion sort, instead of recursively calling quick sort (for small enough sizes, insertion sort is faster!)

Quick sort

- If this example didn't clarify things enough, you may want to look at this video clip of Quick sort with Hungarian folk dance:

  http://www.youtube.com/watch?v=ywWBy6J5gz8

Summary

- During today's class:
  - Introduced quick sort
  - Discussed the basic idea behind the algorithm and its run time.
    - Investigated the average-case vs. worst-case issue.
    - Discussed strategies to avoid the worst-case
  - Looked at an example of operation.