

COMP1002

DATA STRUCTURES AND

ALGORITHMS

LECTURE 9: ADVANCED SORTING



Curtin University

Discipline of Computing

Last updated: [October 6, 2019](#)

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulation 1969

WARNING

This material has been copied and communicated to you by or on behalf of
Curtin University of Technology pursuant to Part VB of the Copyright
Act 1968 (the Act)

The material in this communication may be subject to copyright under the
Act. Any further copying or communication of this material by you may be
the subject of copyright protection under the Act.

Do not remove this notice

This Week

- Sorting recap
- Sorting algorithms
 - MergeSort
 - QuickSort
- We'll use a simulation to help understand the sorts:
 - <https://visualgo.net/en/sorting>

Sorting: Why Sort?

- For clear presentation of data:
 - We often need to present data in an organised manner to a person so that they can make sense of it
 - Sorting is usually a good way to organise data
 - Just imagine using a randomly-ordered phone book!
- To facilitate efficient processing:
 - Selecting a range is simple if the data is in a sorted list
 - e.g., in a supermarket database, finding all transactions between 01/01/2010 and 31/01/2010 is easy if the data is sorted by date
 - Sorting also allows us to search for an item quickly
 - Analogy: finding a name in a phone book
 - you go directly to first letter, then second, etc

Sorting: Time Complexity

- Sorting is concerned with manipulating *all* N items
 - Specifically: to put them in sorted order
- So $O(1)$ for sorting is impossible
 - *At minimum* we must check all N items to see what kind of order they are in – so $O(N)$ is an absolute lower limit
 - e.g., if the elements are already in sorted order
 - ... which almost never happens!
- In the average case, we can expect worse than $O(N)$
 - Not a surprise – sorting is pretty involved

Sorting: Time Complexity

- So what average-case sorting time can we hope for?
 - Naïve approaches to sorting quickly becomes $O(N^2)$
 - More sophisticated sorts are $O(N \log N)$
 - In fact, it has been mathematically proven that no general sorting algorithm can be faster than $O(N \log N)$ in the average case
 - Some sorts are faster only by exploiting characteristics of the data
 - e.g., radix sort is $O(kN)$: needs integer data with known min & max
- Remember: the focus is on average and worst cases
 - Why put much stock in a best case when most of the time it will be the average case that is happening?
 - Worst case also important: shows how bad things might get!

Scaling with N

- The following table illustrates how different complexities scale

N	$N \log_2 N$	N^2
4	8	16
8	24	64
16	64	256
32	160	1,024
64	384	4,096
128	896	16,384
256	2,048	65,536
512	4,608	262,144
1,024	10,240	1,048,576
2,048	22,528	4,194,304
4,096	49,152	16,777,216
8,192	106,496	67,108,865
16,384	229,376	268,435,456
32,768	491,520	1,073,741,824
65,536	1,048,576	4,294,967,296

Sorting Visualisations Revisited

- Each semester we find new sorting visualisation videos
- Here's some more colourful ones:
 - <https://www.youtube.com/watch?v=ljIViETya5k>
 - <https://www.youtube.com/watch?v=bcwwM6EoveA>
- And my new favourite:
 - <https://www.youtube.com/watch?v=QOYcpGnHH0g>
- And if you've got lots of time...
 - <https://www.youtube.com/watch?v=YyerMJlmtts>
 - <https://www.youtube.com/watch?v=lyZQPjUT5B4>

$O(N^2)$ Sorts – Comparison

- That's the $O(N^2)$ -class sorting algorithms that we will be looking at
 - So how do they compare to one another?

Bubble Sort

- ☑ Simple to implement
- ☑ Can finish early (*i.e.*, faster) if data is almost-sorted
 - But 'almost-sorted' has a *very* specific meaning here!
 - In particular, it requires that smaller elements start *not very far* from their *final sorted position*
 - This rarely happens, hence the $O(N)$ best case *degrades* *very quickly* to $O(N^2)$ on pretty much all but already-sorted data
- ☒ Lots of work per pass – constantly swapping
- ☒ Very slow on reverse-ordered data
 - Ends up swapping *every* element on *each* pass

Insertion sort

- ☑ Very fast with almost-sorted data (minimal swaps/compares)
 - Plus, performance degrades ‘gently’ from best case
 - e.g., single out-of-place elements don’t destroy efficiency
 - Hence effective with most arrays that are partially ordered
- ☒ Stable sort (discussed in later slides)
- ☒ Conceptually trickier than other $O(N^2)$ sorts
- ☒ Lots of swaps – need to shuffle larger elements up
- ☒ Very slow on reverse-ordered data

Selection Sort

- ☑ Simple to implement
- ☑ Minimal work per pass – only one swap
 - Thus generally faster in the average case than the others
- ☑ Best case = average case = worst case
 - *i.e.*, always has to perform the full $N-1$ passes, and each pass always involves the same amount of work no matter what the unsorted array looks like
 - P checks + 1 swap
 - But this consistency could be considered a positive too, depending on the situation
- ☑ In-place (discussed next)
- ☒ Unstable sort (we'll discuss what this means next)

Other Factors in Sorting

- Speed is not the only consideration in algorithms
- Extra memory use (memory overhead) is another
 - In sorting, extra memory is often needed for temporary storage to help organise the data
 - In-place vs not in-place sorting
- And different problem domains have their own particular issues
 - For sorting, one of them is whether two identical values will stay in the same relative order after sorting
 - Stable vs unstable sorting

In-Place Sorting

- All the $O(N^2)$ sorting algorithms we've looked at didn't need much temporary storage
 - Only enough to handle the swap, *i.e.*, one temp element
- This makes them 'in-place' sorting algorithms
 - *i.e.*, they sort the array in the same place as the array, without needing to copy chunks to another temp array
- Some sorting algorithms are not in-place
 - For very large data sets that fill up almost all RAM, the extra space of non-in-place sorting can be a problem!

Stable vs Unstable Sorting

- If you look closely at the $O(N^2)$ sorting algorithms, you'll see that the compares were done **carefully**
 - *i.e.*, we chose *very deliberately* whether to do $>$ or $>=$
 - Goal: choose the compare that will ensure that duplicate values will **remain in the same order w.r.t. each other**
- Why bother? So what if identical items get swapped?
 - Ah, but we rarely sort just a list of *single values*
 - Instead, we usually sort *rows* of data, by choosing one of the *columns* to use as the sorting key (*i.e.*, the **comparer**)
 - The other columns get 'dragged along' with the sort column
 - These other columns *won't* be identical, hence we'd like to preserve the relative order of rows with identical sort keys

Stable vs Unstable Sorting

- Thus a 'stable sort' is one which *guarantees* that identically-valued sort keys will keep their ordering
 - And conversely, an **unstable sort** is one that does not
- Bubble sort and Insertion sort are both **stable**
 - But only because we carefully selected $>$ or $>=$
 - (they all ended up as $>$, but that's not a blanket rule!)
- Selection sort is **unstable**
 - Consider selection sort on the list of values 8, 8, 3
 - The first 8 will swap with the 3, then no more swaps
 - This puts the first 8 after the second 8 – unstable
 - No matter how you code it, Selection Sort cannot guarantee stability for all possible arrays

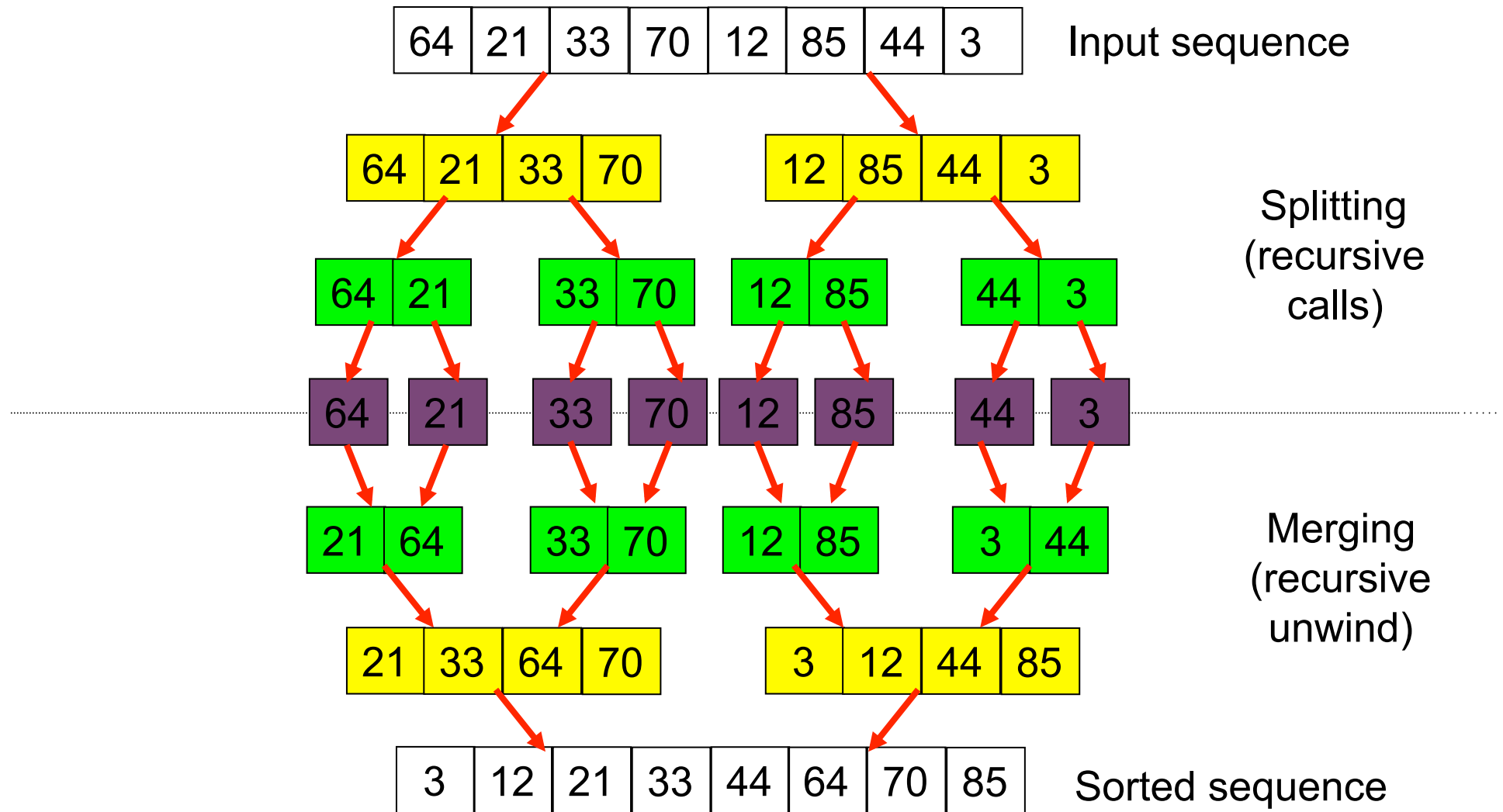
MergeSort – Strategy

- MergeSort is based on the fact that merging two sorted arrays to output a single sorted array is quite simple...
- Just treat the two arrays as queues: look at the front entry in each array and transfer the one which is lower into the output array
 - Later elements are larger and so don't need to be considered yet
- Repeat until all elements are transferred to the output array

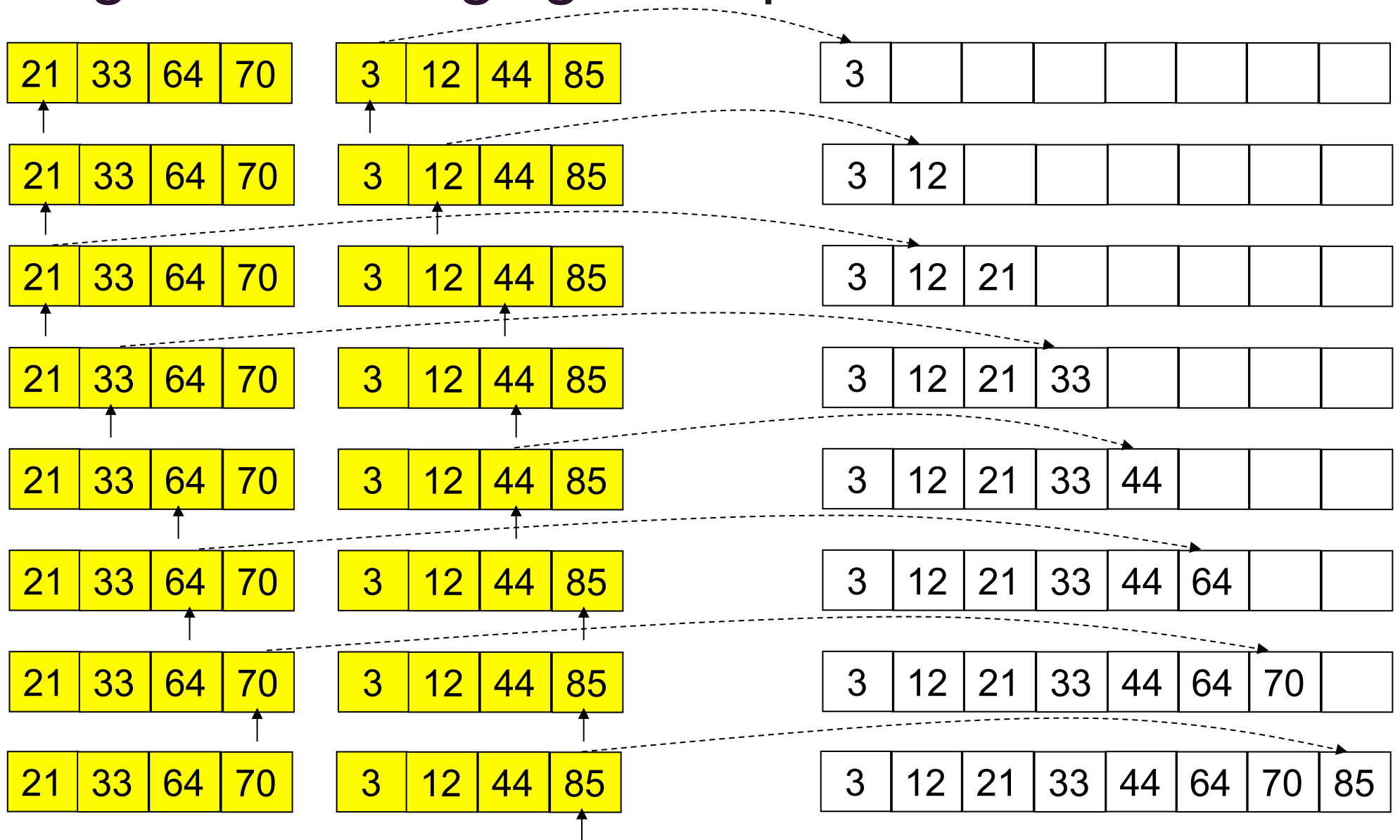
MergeSort – Strategy

- So how do we make the original unsorted array into two sorted sub-arrays so that we can merge easily?
 - Recursion! Split the array in half repeatedly until the problem becomes trivial (a one-element array is sorted by defn!) then start merging
 - Creates a ‘recursion tree’ of splitting the array into sub-arrays
 - These sub-arrays are merged as the recursion ‘unwinds’

MergeSort – Recursion Example



MergeSort – Merging Example



MergeSort – Time Complexity

- Split complexity:
 - Splitting in half increases the number of sub-arrays exponentially: there are 2^L arrays for level L
 - With N elements, we hit one-element arrays when $2^L = N$
 - Thus number of split levels is as follows:
$$2^L = N$$
$$\log_2 (2^L) = \log_2 N$$
$$L * \log_2 2 = \log_2 N$$
$$L = \log_2 N$$
 - *i.e.*, we can split the array $\log_2 N$ times

MergeSort – Time Complexity

- Merging complexity:
 - During merges, we always end up copying all N elements at *each level* in the recursion tree
 - apportioned in different numbers of sub-arrays at each level, but still merging all N elements
 - Number of **compares** can be $< N$, but not the number of **copies**
 - eg: merge {1 2 3} and {10 11 12}: **3 compares**, **6 copies**
 - eg: merge {1 10 12} and {2 3 11}: **5 compares**, **6 copies**
- So, $\log_2 N$ levels, each with N steps = **$O(N \log N)$**
 - Best, average and worst cases are all the same
 - *Much* better average/worst cases than $O(N^2)$ algorithms
 - Check out “Scaling With N ” slide to see this

MergeSort – Algorithm (1/3)

```
METHOD MergeSort IMPORT array, leftIdx, rightIdx EXPORT array
```

```
IF (leftIdx < rightIdx) THEN  
  midIdx ← (leftIdx + rightIdx) / 2
```

```
  MergeSort(array, leftIdx, midIdx)    ← Recurse: Sort left half of the current sub-array
```

```
  MergeSort(array, midIdx+1, rightIdx) ← Recurse: Sort right half of the current sub-array
```

```
  Merge(array, leftIdx, midIdx, rightIdx) ← Merge the left and right sub-arrays
```

```
//ELSE
```

```
//  array is already sorted (only one element!)    ← ie: Base case
```

```
ENDIF
```

```
ENDMergeSort
```

MergeSort – Algorithm (2/3)

```
METHOD Merge IMPORT array, leftIdx, midIdx, rightIdx EXPORT array
```

```
tempArr ← allocate array of length (rightIdx - leftIdx + 1)
```

```
ii ← leftIdx
```

← Index for the 'front' of left sub-array

```
jj ← midIdx + 1
```

← Index for the 'front' of right sub-array

```
kk = 0
```

← Index for next free element in tempArr

```
WHILE (ii <= midIdx) AND (jj <= rightIdx) DO ← Merge sub-arrays into tempArr
```

```
  IF (array[ii] <= array[jj]) THEN ← Use <= for a stable sort
```

```
    tempArr[kk] ← array[ii] ← Take from left sub-array
```

```
    ii ← ii + 1
```

```
  ELSE
```

```
    tempArr[kk] ← array[jj] ← Take from right sub-array
```

```
    jj ← jj + 1
```

```
  ENDIF
```

```
  kk ← kk + 1
```

```
ENDWHILE
```

<Merge() method continues on next slide>

MergeSort – Algorithm (3/3)

Merge() continued...

```
FOR ii ← ii TO midIdx DO
    tempArr[kk] ← array[ii]
    kk ← kk + 1
```

```
ENDFOR
```

```
FOR jj ← jj TO rightIdx DO
    tempArr[kk] ← array[jj]
    kk ← kk + 1
```

```
ENDFOR
```

```
FOR kk ← leftIdx TO rightIdx DO
    array[kk] ← tempArr[kk-leftIdx]
ENDFOR
```

← Flush remainder from left sub-array
NOTE: Goes to midIdx *inclusively*

← OR Flush remainder from right sub-array
NOTE: Goes to rightIdx *inclusively*

← Copy the now-sorted tempArr back to the actual array
← Use kk-leftIdx to align tempArr indexing to zero

MergeSort – Discussion

- MergeSort is **not an in-place sort**
 - The temp array for merging cannot be avoided without slowing MergeSort down
 - On the last merge, this temp array is of size N
 - If N is large, this can become a significant memory overhead
- It is a **stable sort**
 - ... if we're careful with the comparison
- Like Selection Sort, it is very consistent
 - Always halves the array, so is always $O(N \log N)$

QuickSort – Strategy

- Another recursive $O(N \log N)$ algorithm
 - Separate the data into two sub-arrays
 - This time, *partition* it such that a pivot element will divide the left from the right sub-array
 - Rather than simply halving it like MergeSort
 - Organise the data such that the left sub-array contains all values *smaller* than the pivot and the right sub-array contains all values *larger* than the pivot
 - Each sub-array is still unsorted, but the whole has some sorting
 - pivot element is in its correct position
 - Recursively partitioning each sub-array produces QuickSort

QuickSort – Pivot Selection

- The question is, which element should be the pivot?
 - It *must* be one of the elements in the array
 - We need a strategy for selecting a good pivot
- What makes a good pivot?
 - One that divides the array into **equal halves**
 - This will ensure that we only need to do the minimum number of splits to reach one-element sub-arrays
 - *i.e.*, only need $\log_2 N$ split
- What makes a bad pivot?
 - One that doesn't split the data at all! Leads to N splits

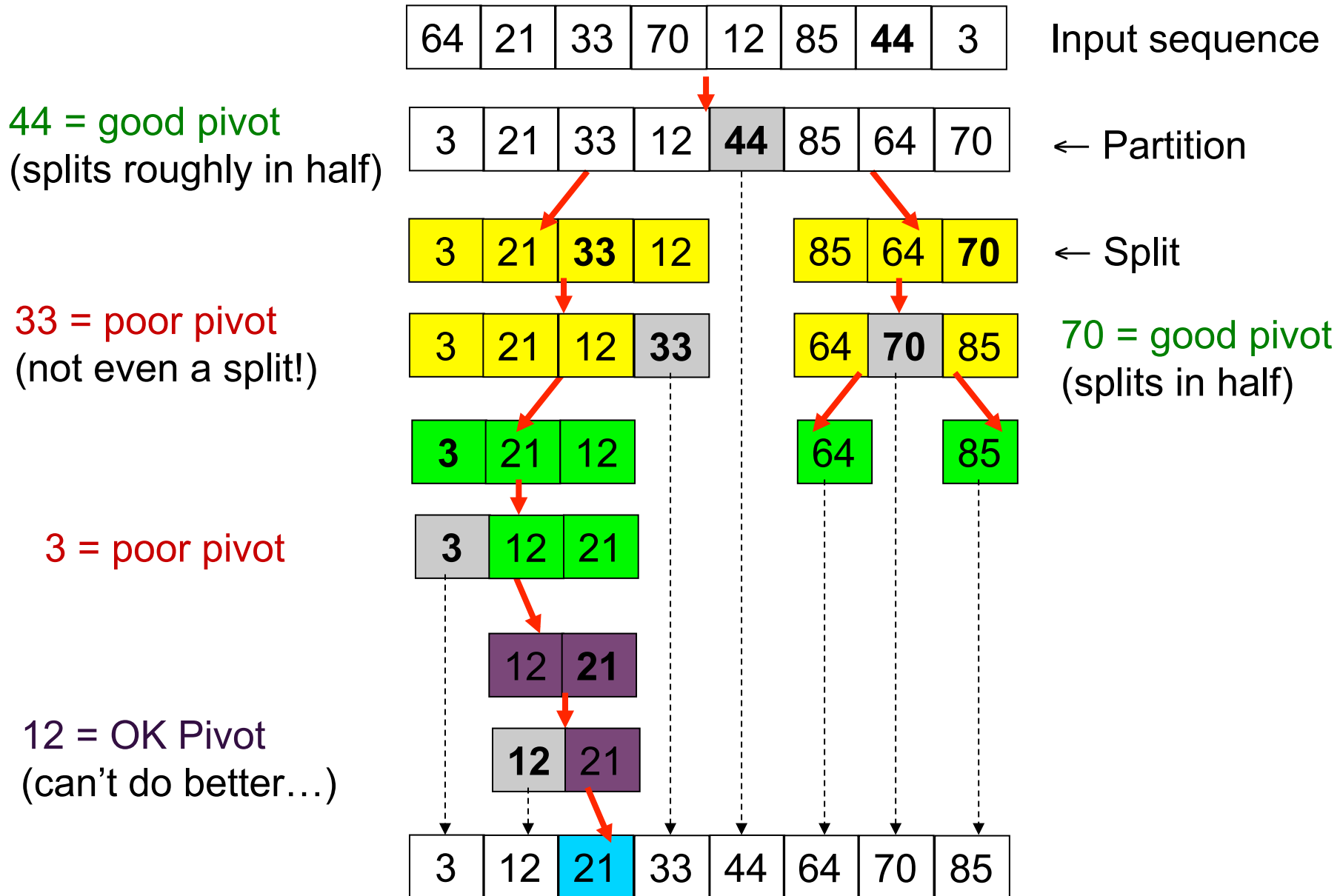
QuickSort – Pivot Selection

- Ideally, we'd choose the median value as the pivot at *every recursion step*
 - Median = central value after sorting (it is *not* the mean!)
 - That would split the array exactly in two every time
 - But the median itself is pretty hard (slow) to calculate
- An easy pivot is to choose the left-most element
 - **Problem**: if the array is already sorted, then there is **never a left sub-array**: no element is smaller than the left-most!
 - Similarly for the right-most
 - Leads to N splits: this makes QuickSort *much* slower
- We'll talk about pivot selection strategies a little later

QuickSort – Partitioning

- The concept of partitioning around the pivot is:
swap elements around until the pivot separates
the data into values $<$ and $>$ the pivot
 - There are several algorithms to achieve this
 - We'll look at a reasonably simple one that is also fast
 - However, it generally leads to an unstable sort too
 - Different algorithms exist to achieve a stable sort, but sacrifice speed or memory in the process
 - Since we must ensure all elements are split by the pivot, we must check all N elements in a partitioning run

QuickSort – Eg (Random Pivot = highlighted)



QuickSort – Time Complexity

- QuickSort complexity is a little difficult to evaluate
 - Depends heavily on how good the pivots are
 - Best case: every pivot splits the sub-arrays **exactly in half**
 - Results in an optimal $\log_2 N$ split levels
 - Every level involves about $N - (2^L)$ **compares** (L = level num), plus up to $N - (2^L)$ **swaps**
 - Since the pivots of the previous levels can be ignored
 - Approximately $O(N)$ per level (early levels are more, later are less)
 - Complexity: **$O(N \log N)$** – each level does $\sim O(N)$ compares

QuickSort – Time Complexity

- Worst case: every pivot creates **no split at all**
 - *i.e.*, the pivot is either the largest or smallest value
 - $N + N-1 + N-2 + \dots$ **compares** + a similar number of **swaps**
 - $= N(N+1)/2 + N(N+1)/2 \approx N^2$
 - Complexity: **$O(N^2)$**
- Average case: pivots split sub-arrays into $2/3$ and $1/3$ pieces
 - *ie*: half-way between best pivot and worst pivot cases
 - Results in $\log_{1.5} N$ split levels (Note the $1.5 = 3/2$: worse than \log_2)
 - Still $O(N)$ compares and swaps per level
 - Complexity: **$O(N \log N)$**
 - Differs from Best Case in that the log term is now not so good

QuickSort – Pivot Selection Strategies

- The pivot is the critical factor in QuickSort's speed
 - Poor pivots can lead to the worst case of $O(N^2)$
- There are many choices for pivots:
 - Left-most or right-most element
 - Middle element
 - Random element
 - Median-of-three

QuickSort – Pivot Selection Strategies

- **Left-most or right-most** are bad choices
 - If the array is sorted or reverse-ordered, left/right as the pivot will result in the worst case of $O(N^2)$
 - The problem is that *(semi)-sorted data is not unusual!*
- **Middle** element is a much better choice
 - Just as simple, and very unlikely to always be a poor pivot
 - For randomly-ordered data, it'll be an average pivot: sometimes good, sometimes bad, mostly OK
 - This is good enough for QuickSort to be fast with
 - For sorted/reversed arrays it will hit the **best case** since the middle element of a sorted list *is the median of that list*
 - There would have to be a very special ordering of the array for the middle element to cause $O(N^2)$ behaviour – highly unlikely

QuickSort – Pivot Selection Strategies

- **Random element** – select an element at random
 - Similar properties to middle element: average case will be common, worst case will be highly unlikely
 - But: random won't get a best case on sorted/reversed data
- **Median-of-three**
 - Choose three elements and take the median of those three
 - e.g., left-most, right-most and middle elements
 - e.g., random three elements
 - Reason: improves the chances of getting a good pivot
 - **Three** is a more representative sample of the array than **one**
 - At the very least, it guarantees a split with at least one element

QuickSort – Median-of-Three

- Choosing left, right and middle elements is probably the best strategy to take
 - Simple (thus fast)
 - Should get near-best-case sorting for semi-sorted data
- Why not do median-of-more-than-three?
 - If three is good, wouldn't (say) nine be better?
 - No: diminishing returns set in
 - The extra time to calculate the median of nine elements is **more than the improvement it will give to QuickSort's speed**

QuickSort – Algorithm (1/2)

```
METHOD QuickSort IMPORT array, leftIdx, rightIdx EXPORT array

IF (rightIdx > leftIdx) THEN      ← Check that the array is > one element in size
    pivotIdx ← (leftIdx+rightIdx) / 2  ← Pivot selection strategy: middle element
    newPivotIdx ← doPartitioning(array, leftIdx, rightIdx, pivotIdx)

    QuickSort(array, leftIdx, newPivotIdx-1)  ← Recurse: Sort left partition
    QuickSort(array, newPivotIdx+1, rightIdx) ← Recurse: Sort right partition
//ELSE
    // Base case: array is 1 element or smaller in size - already sorted
ENDIF

END
```

<doPartitioning() method on next slide>

QuickSort – Algorithm (2/2)

```
METHOD doPartitioning IMPORT array, leftIdx, rightIdx, pivIdx EXPORT newPivIdx
pivotVal ← array[pivIdx]
array[pivIdx] ← array[rightIdx]      ← Swap the pivotVal with the right-most element
array[rightIdx] ← pivotVal

// Find all values that are smaller than the pivot
// and transfer them to the left-hand-side of the array
currIdx ← leftIdx

FOR (ii ← leftIdx TO rightIdx-1)
    IF (array[ii] < pivotVal) THEN      ← Find the next value that should go on the left
        temp ← array[ii]              ← Put this value to the left-hand-side
        array[ii] ← array[currIdx]
        array[currIdx] ← temp
        currIdx ← currIdx + 1
    ENDIF
ENDFOR

newPivIdx ← currIdx
array[rightIdx] ← array[newPivIdx]    ← Put the pivot into its rightful place (the value at
array[newPivIdx] ← pivotVal           [newPivIdx] is >= pivotVal, so it can be put to the end)
```

QuickSort – Discussion

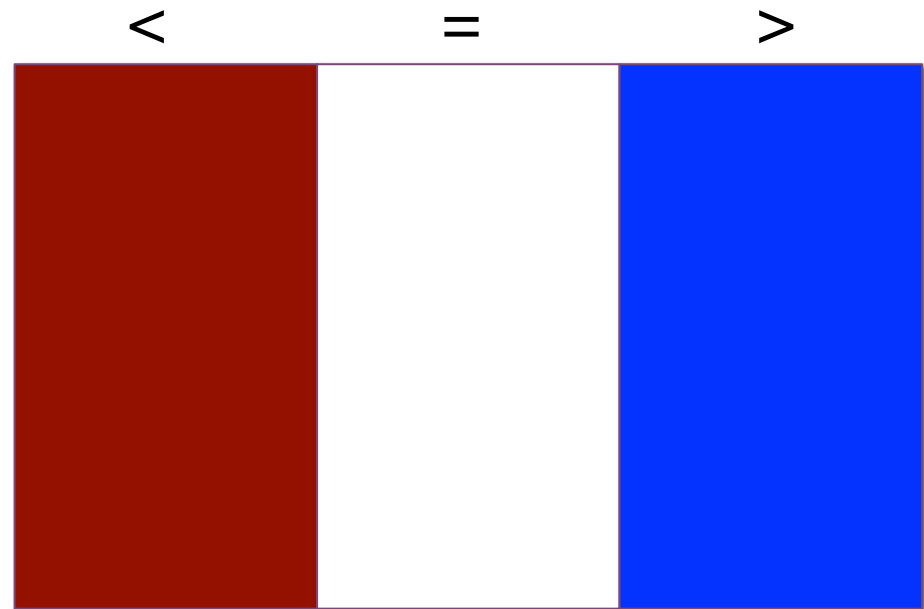
- QuickSort tends to be very fast
 - Pivots are put in their ultimately-sorted positions early
 - ... so you don't need to check them ever again after the split
 - Even if a few bad pivots are chosen, as long as splits are occurring QuickSort's average case is easily achieved
- ...can be complicated to implement
 - Choosing a good pivot needs some effort
 - Partitioning the array ready for a split isn't simple to understand (although the code itself isn't too complex)
 - ...and fast partitioning methods create unstable sorts
- ...and a bad implementation can degrade to $O(N^2)$

QuickSort – Discussion

- There are special cases where it is difficult to avoid QuickSort's $O(N^2)$, even with a good pivot strategy
 - This can occur in arrays that have only a few unique values but these values are duplicated many times
 - e.g., 1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,3,3,3,2,2,2,2,2,2,2,4,4,4,4,4,4
 - An extreme example can illustrate the problem best:
 - e.g., 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
 - The *optimal* pivot is 1, but it still leads to $O(N^2)$, because the pivot is unavoidably the largest (or smallest) for every split!
 - Less extreme examples can still cause poor performance since initially the splits are OK, but soon you hit sub-arrays that contain values that are all the same

QuickSort – 3-way Algorithm

- Variation of code to deal with duplicates
- Partitions into three categories: $<$ $=$ $>$



MergeSort

- ✓ Consistently $O(N \log N)$: best, average, worst cases
- ✓ Stable sort
- ✓ Simpler than QuickSort (but still pretty complicated)
- ✗ Not in-place: requires temp array of size N for merge
- ✗ Recursive, so can *theoretically* cause stack overflow
 - ✓ However, it guarantees $\log_2 N$ levels, so this unlikely
 - e.g., $\log_2(N=1,000,000,000) \rightarrow \text{max recursive call depth} = 30$
 - Shows that $\log_2(N)$ scales *exceptionally* well

QuickSort

- ☑ In general, the fastest sorting algorithm around
- ☑ In-place sort
 - Only need a single temp variable for swaps
- ☒ Unstable sort
- ☒ Fairly complicated to implement well
- ☒ Recursive, so can cause a stack overflow
 - And unlike MergeSort, it **cannot guarantee $\log_2 N$ levels**
 - Could be N levels in the worst case; $\log_2 N$ only in the best case
- ☒ $O(N^2)$ worst case
 - But a good impl. is unlikely to be worse than $O(N \log N)$

Summary of Sorting Algorithms

	Pros	Cons
Bubble Sort	<ul style="list-style-type: none">• Simple to implement• Fast if already sorted• In-place, stable sort	<ul style="list-style-type: none">• Generally poor speed – too many swaps• In practice, only ever comes close to best case performance if data is <i>already</i> sorted
Insertion Sort	<ul style="list-style-type: none">• Works relatively fast (vs other $O(N^2)$) with a variety of semi-sorted data• In-place, stable sort	<ul style="list-style-type: none">• Not particularly simple to implement• Slow on reversed / near-reversed data
Selection Sort	<ul style="list-style-type: none">• Simple to implement• Minimal work per pass (only one swap)• In-place	<ul style="list-style-type: none">• Best, avg and worst cases are all identical -ie: takes <u>no</u> advantage of semi-sorted data• Unstable sort
MergeSort	<ul style="list-style-type: none">• Consistently very fast• Stable sort	<ul style="list-style-type: none">• Not in-place: needs a temp array (this doubles the memory space required)• Fairly complex to implement
QuickSort	<ul style="list-style-type: none">• Typically the fastest algorithm for most data sets (if implemented well)• In-place sort	<ul style="list-style-type: none">• $O(N^2)$ worst case• Complicated: many factors to consider for achieving a good implementation• Recursive: stack overflow possible• Unstable sort

Summary of Sorting: Big-O

	Best Case	Average Case	Worst Case
Bubble Sort	$O(N)^\dagger$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
MergeSort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
QuickSort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$

[†] In practice this case really only occurs if the data is already sorted

Next Week

- Advanced Trees