CSE 373

NOVEMBER 15TH – NON-COMPARISON SORTS

ASSORTED MINUTIAE

- P3 part 1 due 11:30

ASSORTED MINUTIAE

- **P3 part 1 due 11:30**
  - Make sure partners names are in code and on canvas for submissions
  - 50% awarded back on part 2 submissions next Wednesday
  - EC added, due with part 3

ASSORTED MINUTIAE

- **P3 part 1 due 11:30**
  - Make sure partners names are in code and on canvas for submissions
  - 50% awarded back on part 2 submissions next Wednesday
  - EC added, due with part 3
- **Last Written Assignment**
  - Out Nov 29, Due December 6 **No late days**
  - Extra credit opportunity

ASSORTED MINUTIAE

- Unless you’ve talked with me and set up a meeting, midterm grades are final

ASSORTED MINUTIAE

- **Unless you've talked with me and set up a meeting, midterm grades are final**
  - 25% for lower exam score
  - 35% for higher exam score

ASSORTED MINUTIAE

- **Unless you've talked with me and set up a meeting, midterm grades are final**
  - 25% for lower exam score
  - 35% for higher exam score
- **Final Exam**
  - December 12th, 2:30 – 4:20
  - Twice the time, more critical thought

RECAP FROM MONDAY

- Partitioning and Merging
- N log N – Lower bound for comparison sorts
- Cutoffs
- Section07 solutions are posted and show how to show work

COUNTING COMPARISONS

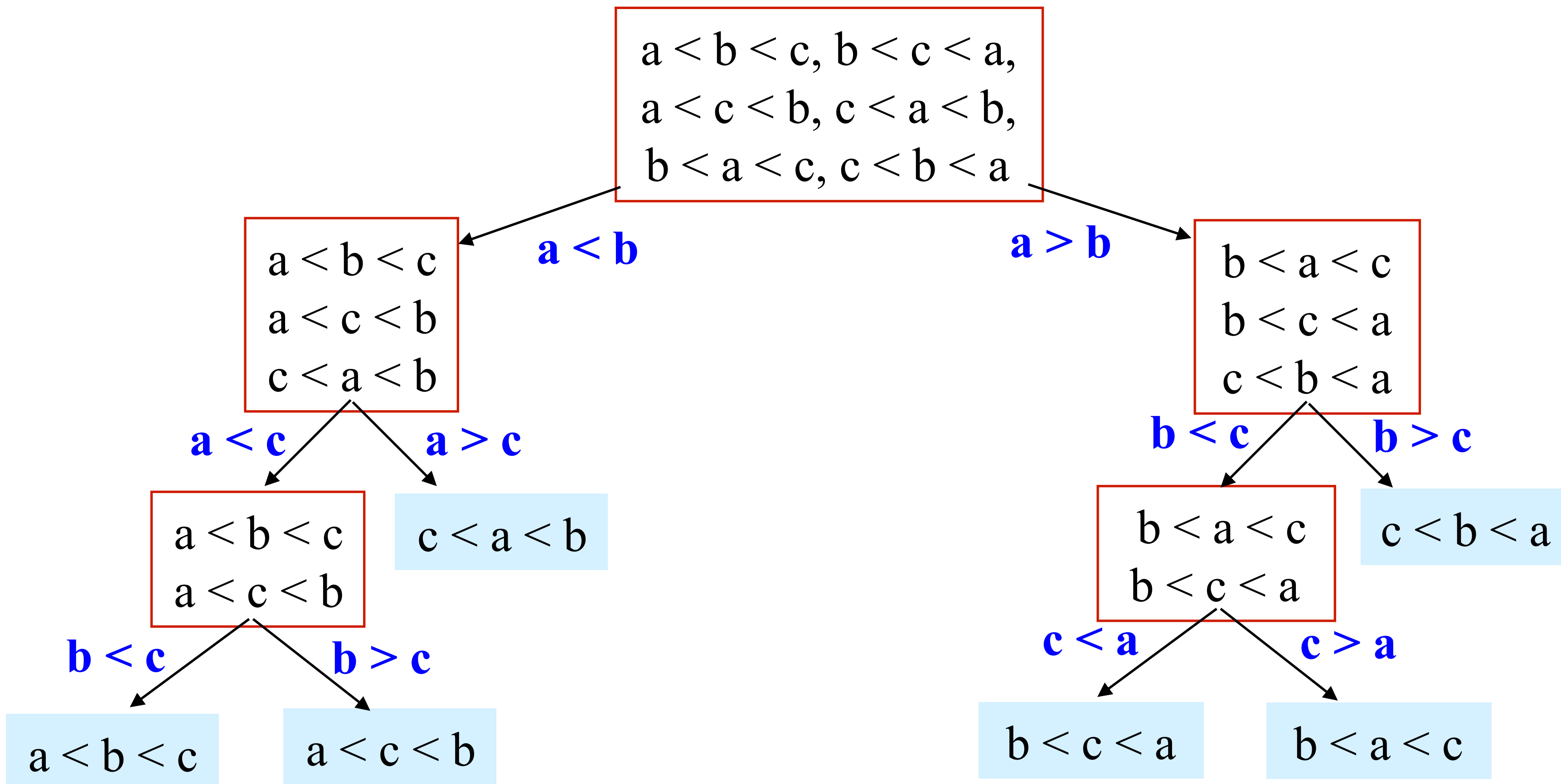
No matter what the algorithm is, it cannot make progress without doing comparisons

- **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings

Can represent this process as a *decision tree*

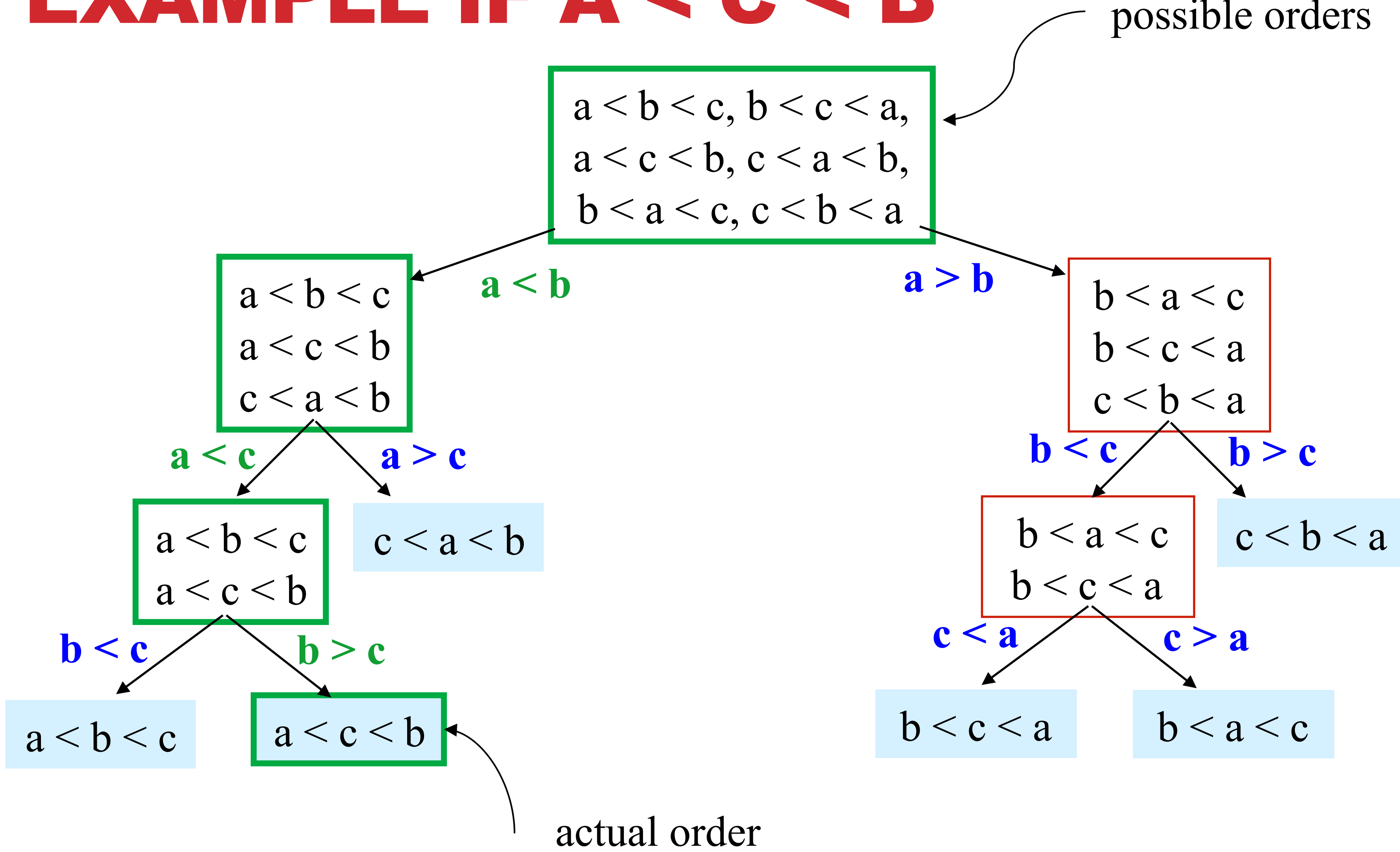
- Nodes contain "set of remaining possibilities"
- Edges are "answers from a comparison"
- The algorithm does not actually build the tree; it's what our *proof* uses to represent "the most the algorithm could know so far" as the algorithm progresses

DECISION TREE FOR N = 3



- The leaves contain all the possible orderings of a, b, c

EXAMPLE IF A < C < B



DECISION TREE

A binary tree because each comparison has 2 outcomes (we're comparing 2 elements at a time)

Because any data is possible, any algorithm needs to ask enough questions to produce all orderings.

The facts we can get from that:

1. Each ordering is a different leaf (only one is correct)
2. Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree. Worst number of comparisons is the longest path from root-to-leaf in the decision tree for input size n
3. There is no worst-case running time better than the height of a tree with *<num possible orderings>* leaves

POSSIBLE ORDERINGS

Assume we have *n* elements to sort. How many *permutations* of the elements (possible orderings)?

- For simplicity, assume none are equal (no duplicates)

Example, *n*=3

$$a[0]<a[1]<a[2] \qquad a[0]<a[2]<a[1]$$
$$a[1]<a[0]<a[2]$$
$$a[1]<a[2]<a[0] \qquad a[2]<a[0]<a[1]$$
$$a[2]<a[1]<a[0]$$

In general, *n* choices for least element, *n*-1 for next, *n*-2 for next, …

- *n*(*n*-1)(*n*-2)…(2)(1) = ***n*!** possible orderings

That means with n! possible leaves, best height for tree is log(n!), given that best case tree splits leaves in half at each branch

RUNTIME

That proves runtime is at least Ω(`log` (*n!*)). Can we write that more clearly?

$$
\begin{aligned}
\lg(n!) &= \lg(n(n-1)(n-2)\ldots 1) && \text{[Def. of } n!\text{]} \\
&= \lg(n) + \lg(n-1) + \ldots \lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2} - 1\right) + \ldots \lg(1) && \text{[Prop. of Logs]} \\
&\geq \lg(n) + \lg(n-1) + \ldots + \lg\left(\frac{n}{2}\right) \\
&\geq \left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) \\
&= \left(\frac{n}{2}\right)(\lg n - \lg 2) \\
&= \frac{n \lg n}{2} - \frac{n}{2} \\
&\in \Omega(n \lg(n))
\end{aligned}
$$

Nice! Any sorting algorithm must do *at best* (1/2)*(*n*`log` *n* – *n)* comparisons: Ω(*n*`log` *n*)

SORTING

- “Slow” sorts

SORTING

- **“Slow” sorts**
  - Insertion
  - Selection

SORTING

- **“Slow” sorts**
  - Insertion
  - Selection
- **“Fast” sorts**

SORTING

- **“Slow” sorts**
  - Insertion
  - Selection
- **“Fast” sorts**
  - Quick
  - Merge
  - Heap

SORTING

- **“Slow” sorts**
  - Insertion
  - Selection
- **“Fast” sorts**
  - Quick
  - Merge
  - Heap
- **These are all comparison sorts, can’t do better than O(n log n)**

SORTING

- **Non-comparison sorts**

SORTING

- **Non-comparison sorts**
  - If we know something about the data, we don't strictly need to compare objects to each other

SORTING

- **Non-comparison sorts**
  - If we know something about the data, we don't strictly need to compare objects to each other
  - If there are only a few possible values and we know what they are, we can just sort by identifying the value

SORTING

- **Non-comparison sorts**
  - If we know something about the data, we don't strictly need to compare objects to each other
  - If there are only a few possible values and we know what they are, we can just sort by identifying the value
  - If the data are strings and ints of finite length, then we can take advantage of their sorted order.

SORTING

- **Two sorting techniques we use to this end**

SORTING

- **Two sorting techniques we use to this end**
  - Bucket sort

SORTING

- **Two sorting techniques we use to this end**
  - Bucket sort
  - Radix sort

SORTING

- **Two sorting techniques we use to this end**
  - Bucket sort
  - Radix sort
- **If the data is sufficiently structured, we can get O(n) runtimes**

BUCKETSORT

If all values to be sorted are known to be integers between 1 and *K* (or any small range):

- Create an array of size *K*
- Put each element in its proper bucket (a.k.a. bin)
- *If* data is only integers, no need to store more than a *count* of how times that bucket has been used

Output result via linear pass through array of buckets

`count` array	
1	3
2	1
3	2
4	2
5	3

- Example:

  K=5

  input (5,1,3,4,3,2,1,1,5,4,5)

  output: 1,1,1,2,3,3,4,4,5,5,5

ANALYZING BUCKET SORT

Overall: $O(n+K)$

- Linear in n, but also linear in K

Good when K is smaller (or not much larger) than n

- We don't spend time doing comparisons of duplicates

Bad when K is much larger than n

- Wasted space; wasted time during linear $O(K)$ pass

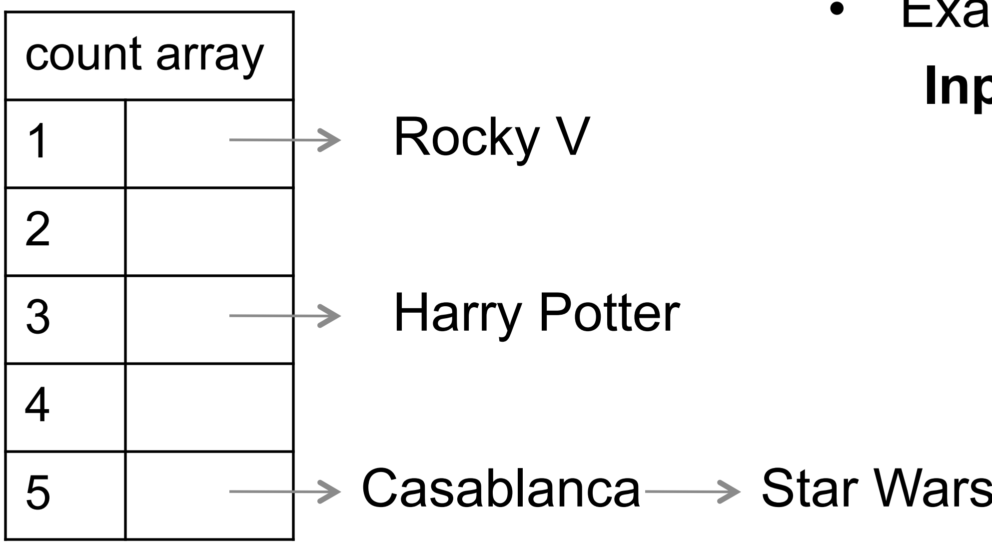
For data in addition to integer keys, use list at each bucket

BUCKET SORT

Most real lists aren’t just keys; we have data

Each bucket is a list (say, linked list)

To add to a bucket, insert in *O*(1) (at beginning, or keep pointer to last element)



- Example: Movie ratings; scale 1-5

  Input:

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Rocky V

- Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- Easy to keep ‘stable’; Casablanca still before Star Wars

RADIX SORT

Radix = “the base of a number system”

- Examples will use base 10 because we are used to that
- In implementations use larger numbers
  - For example, for ASCII strings, might use 128

Idea:

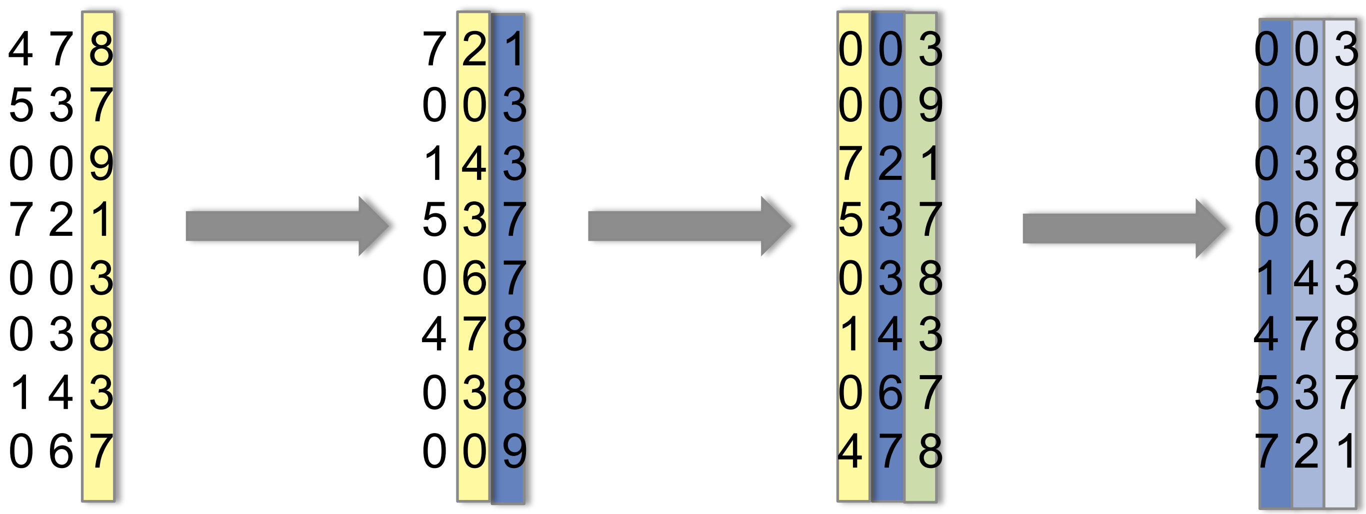
- Bucket sort on one digit at a time
  - Number of buckets = radix
  - Starting with *least* significant digit
  - Keeping sort *stable*
- Do one pass per digit
- **Invariant**: After k passes (digits), the last k digits are sorted

RADIX SORT EXAMPLE

Radix = 10

Input: 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow



ANALYSIS

Input size: n

Number of buckets = Radix: B

Number of passes = "Digits": P

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $O(P(B+n))$

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
    - Run-time proportional to: 15*(52 + n)
    - This is less than n log n only if n > 33,000
    - Of course, cross-over point depends on constant factors of the implementations

SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small n

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small n

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \texttt{ log } n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
  - Often fastest, but depends on costs of comparisons/copies

SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small n

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies

$\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons

SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small *n*

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n$ `log` $n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies

Ω $(n$ `log` $n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts

- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!

SORTING TAKEAWAYS

Simple $O(n^2)$ sorts can be fastest for small n

- Selection sort, Insertion sort (latter linear for mostly-sorted)
- Good for "below a cut-off" to help divide-and-conquer sorts

$O(n \log n)$ sorts

- Heap sort, in-place but not stable nor parallelizable
- Merge sort, not in place but stable and works as external sort
- Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies

$\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons

Non-comparison sorts

- Bucket sort good for small number of possible key values
- Radix sort uses fewer buckets and more phases

Best way to sort? It depends!