CSE 332 Autumn 2026
Lecture 13: Sorting, pt. 2

Ruth Anderson

http://www.cs.uw.edu/332

Announcements

- “Meet the staff” activity – Come meet us in office hours!
  - Will keep this open thru end of week 6 (Nov 6), but will be de-prioritized in OH
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 4 AVL Trees: Due Friday Oct 30
- Exercise 5 Hashing: Due Friday Nov 6
- Exercise 6 Sorting: coming later this week, due Nov 13
- Midterm: 5-6pm Monday November 9
  - Review during both lecture sections on Fri Nov 6
  - No lecture on Mon Nov 9
  - Sample exams posted! See exams page.

Properties To Consider

- Worst case running time
- In place:
  - We only need to use the pre-existing array to do sorting
  - Constant extra space (only some additional variables needed)
- Adaptive
  - The running time improves as the given list is closer to being sorted
  - It should be linear time for a pre-sorted list, and nearly linear time if the list is nearly sorted
- Online
  - We can start sorting before we have the entire list.
- Stable
  - “Tied” elements keep their original order

Divide And Conquer Sorting

- Divide and Conquer:
  - Recursive algorithm design technique
  - Solve a large problem by breaking it up into smaller versions of the same problem

Divide and Conquer

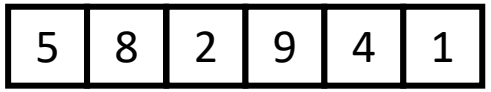
- **Base Case**:
  - If the problem is “small” then solve directly and return
- **Divide**:
  - Break the problem into subproblem(s), each smaller instances
- **Conquer**:
  - Solve subproblem(s) recursively
- **Combine**:
  - Use solutions to subproblems to solve original problem

Divide and Conquer Template Pseudocode

```
def my_DandC(problem) {
   // Base Case
   if (problem.size() <= small_value) {
      return solve(problem);  // directly solve (e.g., brute force)
   }
   // Divide
   List subproblems = divide(problem);

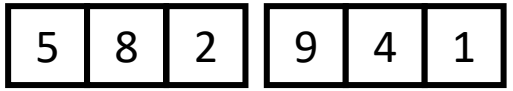
   // Conquer
   solutions = new List();
   for (sub : subproblems) {
      subsolution = my_DandC(sub);
      solutions.add(subsolution);
   }
   // Combine
   return combine(solutions);
}
```

Merge Sort

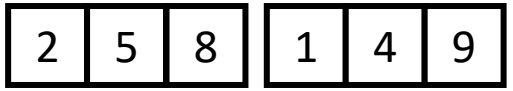




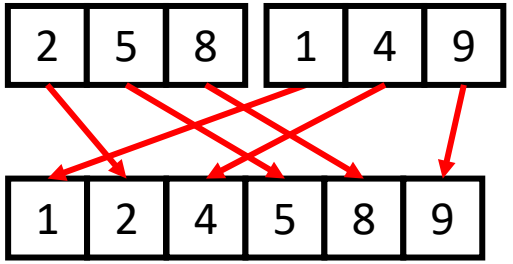
- **Base Case:**
  - If the list is of length 1 or 0, it's already sorted, so just return it



- **Divide:**
  - Split the list into two "sublists" of (roughly) equal length



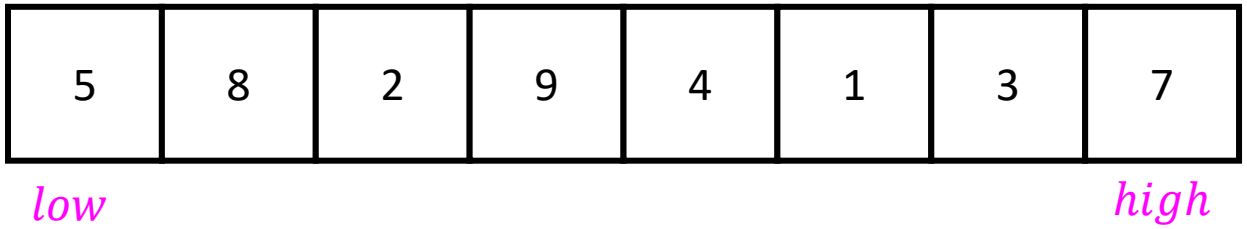
- **Conquer:**
  - Sort both lists recursively



- **Combine:**
  - **Merge** sorted sublists into one sorted list

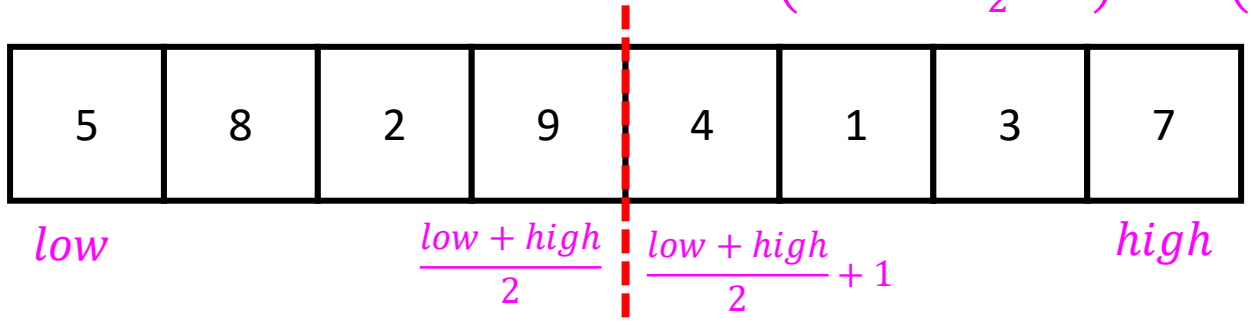
Merge Sort In Action!

Sort between indices low and $high$

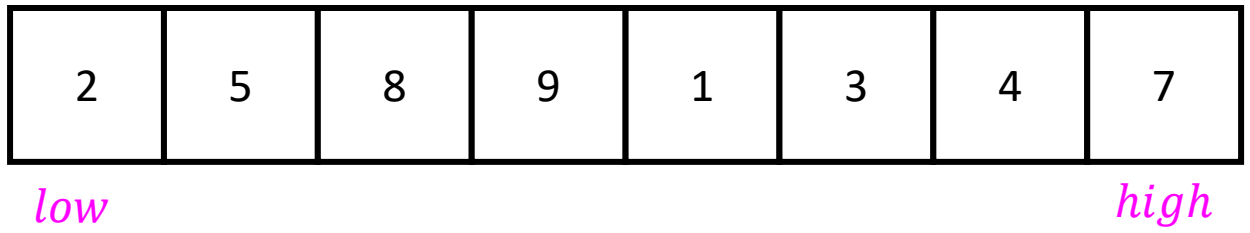


Base Case: if low == $high$ then that range is already sorted!

Divide and Conquer: Otherwise call mergesort on ranges $\left(low, \frac{low+high}{2}\right)$ and $\left(\frac{low+high}{2}+1, high\right)$



After Recursion:

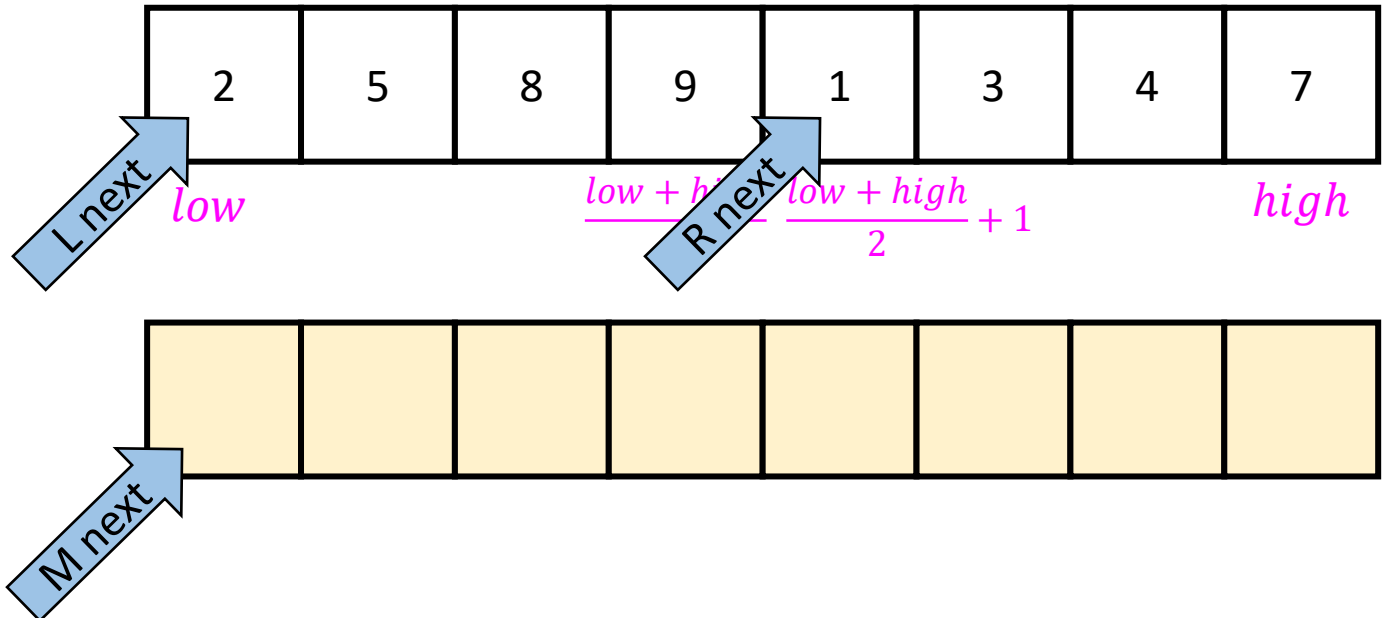


Merge (the combine part)

Create a new array to merge into, and 3 pointers/indices:

- L_next: the smallest "unmerged" thing on the left
- R_next: the smallest "unmerged" thing on the right
- M_next: where the next smallest thing goes in the merged array

One-by-one: put the smallest of L_next and R_next into M_next, then advance both M_next and whichever of L/R was used.



Merge Sort Pseudocode

```
void mergesort(myArray) {
        ms_helper(myArray, 0, myArray.length());
}
void ms_helper(myArray, low, high) {
        if (low == high) {return;}  // Base Case
        mid = (low+high)/2;
        ms_helper(low, mid);
        ms_helper(mid+1, high);
        merge(myArray, low, mid, high);
}
```

Merge Pseudocode

```
void merge(myArray, low, mid, high) {
        merged = new int[high-low+1]; // or whatever type is in myArray
        l_next = low;
        r_next = mid + 1;
        m_next = 0;
        while (l_next <= mid && r_next <= high) {
                if (myArray[l_next] <= myArray[r_next]) {
                        merged[m_next++] = myArray[l_next++];
                }
                else {
                        merged[m_next++] = myArray[r_next++];
                }
        }
        while (l_next <= mid) { merged[m_next++] = myArray[l_next++]; }
        while (r_next <= high) {  merged[m_next++] = myArray[r_next++]; }
        for (i = 0; i <= merged.length; i++) { myArray[i+low] = merged[i];}
}
```

Analyzing Merge Sort

1. Identify time required to Divide and Combine
2. Identify all subproblems and their sizes
3. Use recurrence relation to express recursive running time
4. Solve and express running time asymptotically

- **Divide**: 0 comparisons
- **Conquer**: recursively sort two lists of size $\frac{n}{2}$
- **Combine**: n comparisons
- **Recurrence:**

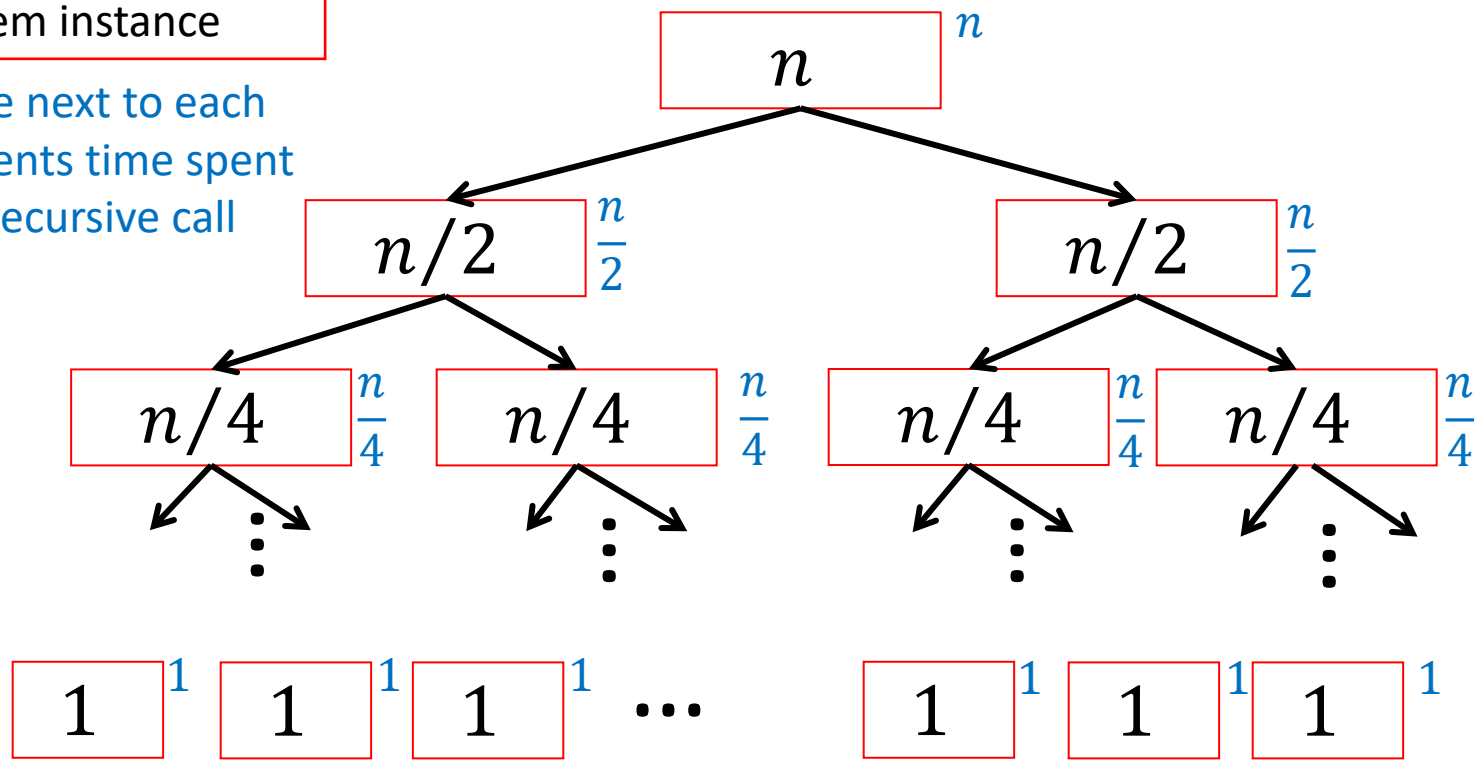
$$T(n) = 0 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Tree Method $T(n) = 2T\left(\frac{n}{2}\right) + n$

Red box represents a problem instance

Blue value next to each box represents time spent at that recursive call



$\Rightarrow n$ work per level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=0}^{\log_2 n} n = \Theta(n \log n)$$

Merge Sort Properties

- Worst case running time
  - $\Theta(n \log n)$
- In place:
  - NO!
  - We need a second array to merge into
- Adaptive
  - NO!
  - Running time is the same regardless of original list's order
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - YES!
  - When merging, and there's a tie, choose the element from the left

Quicksort Vs. Mergesort

- Like Mergesort:
  - Divide and conquer
  - $O(n \log n)$ run time (kind of…)
- Unlike Mergesort:
  - Divide step is the “hard” part
  - *Typically* faster than Mergesort

Quicksort Overview

Idea: pick a pivot element from the array, recursively sort two sublists around that element

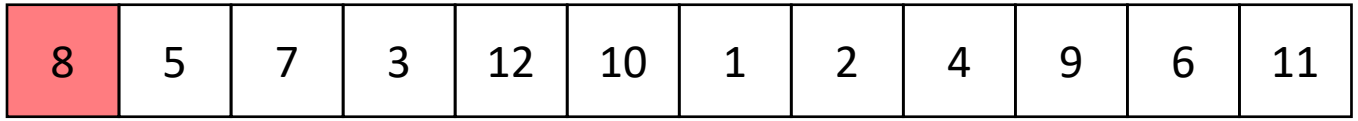
- Divide: select pivot element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing to do!

Partition (Divide step)

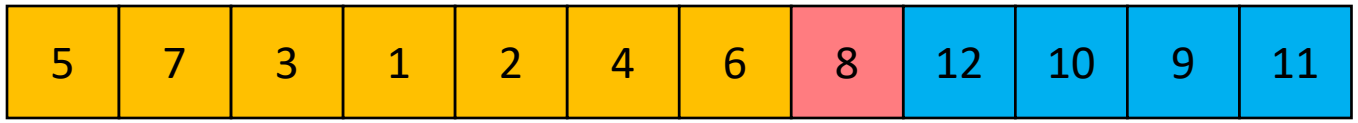
Given: a list, a pivot p

- (We'll talk later about how to select the pivot)
- Move the pivot to the first location in the array

Start: unordered list



Goal: All elements $< p$ on left, all $> p$ on right

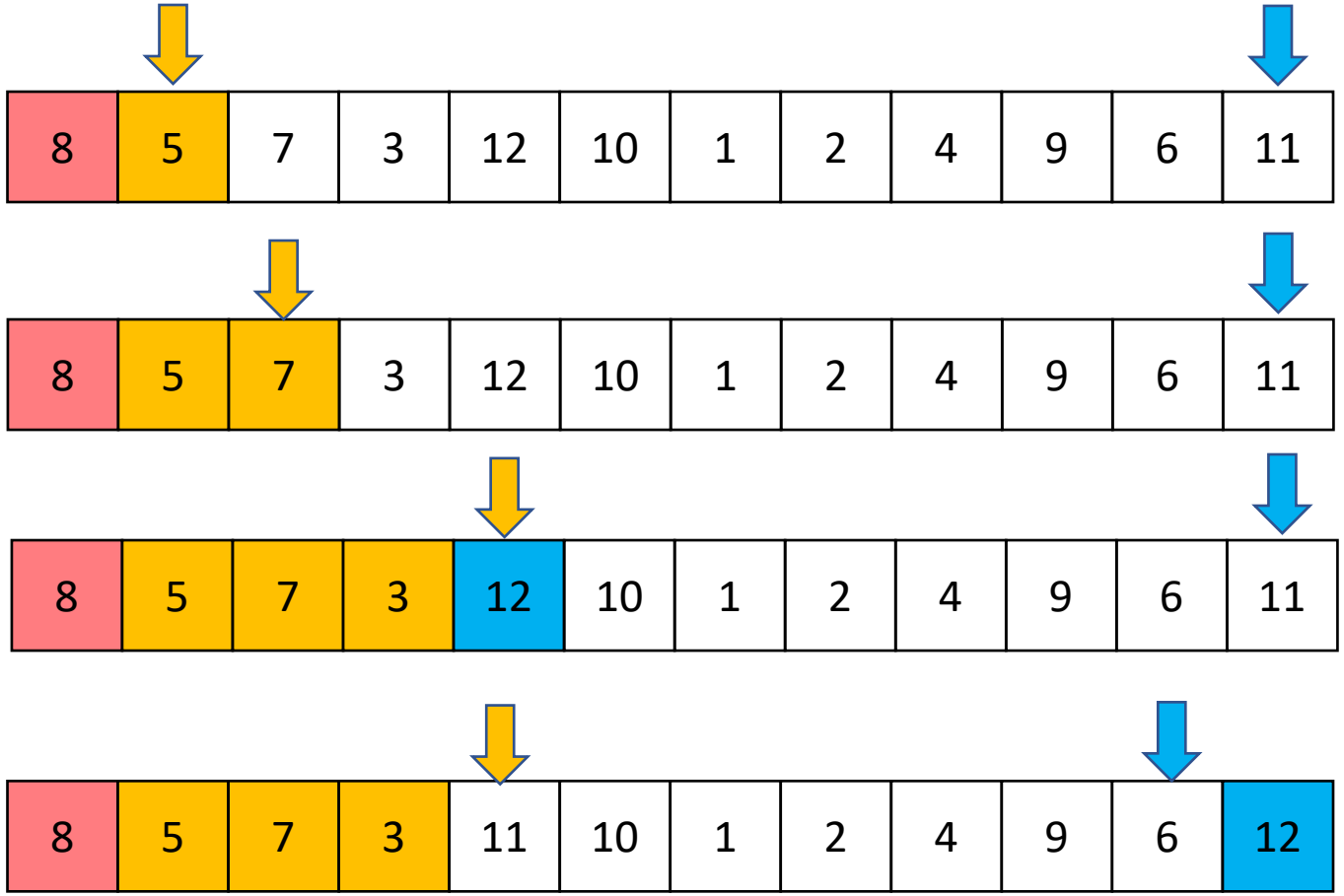


Partition Procedure (Slide 1 of 2)

If Begin value $<$ p, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End

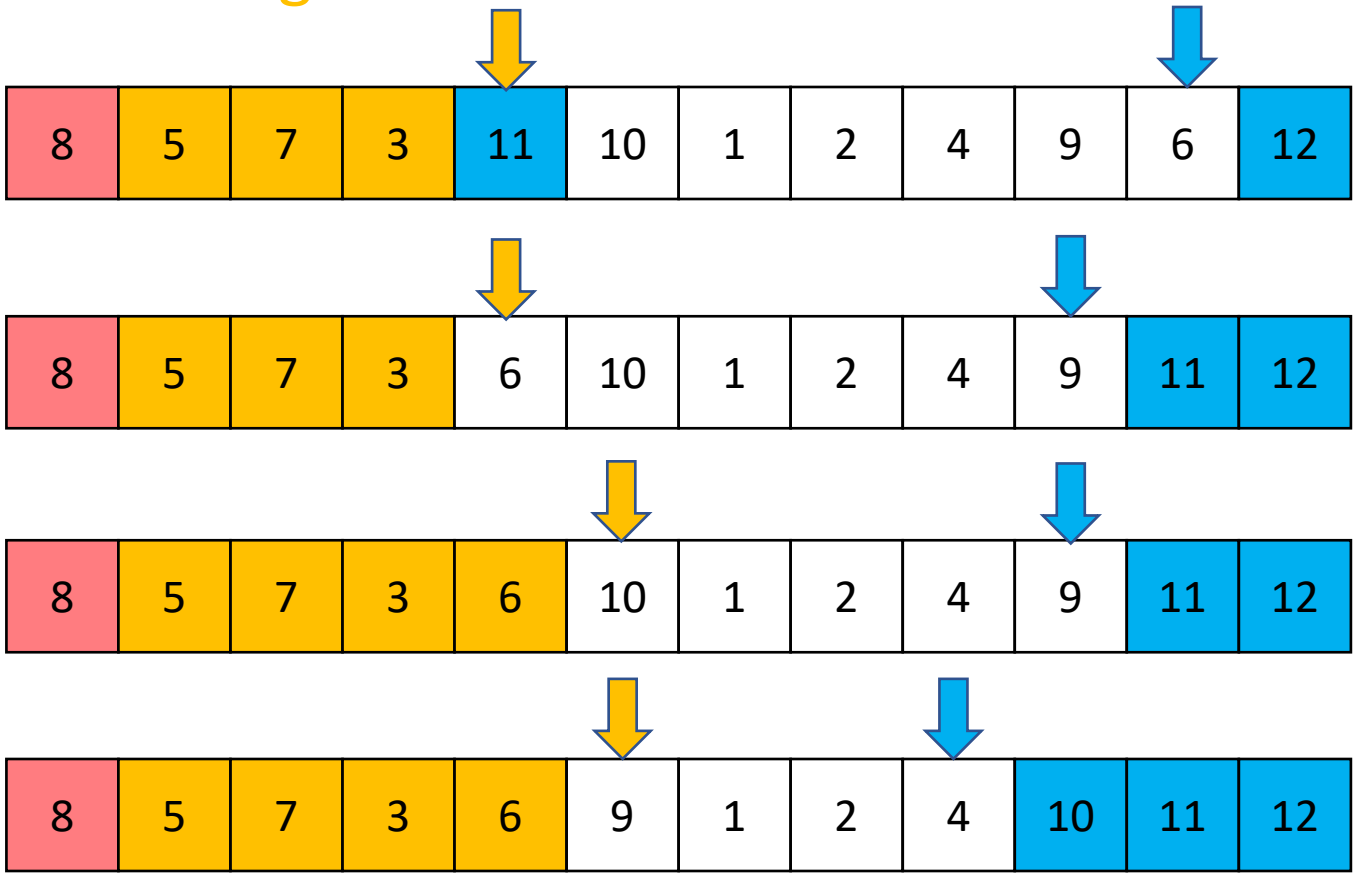


Partition Procedure (Slide 2 of 2)

If Begin value $<$ p, move Begin right

Else swap Begin value with End value, move End Left

Done when Begin = End

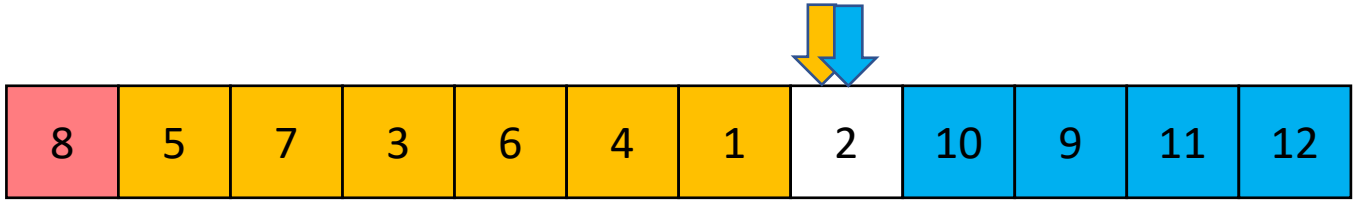


Partition – Begin Meets End (Case 1)

If Begin value $<$ p, move Begin right

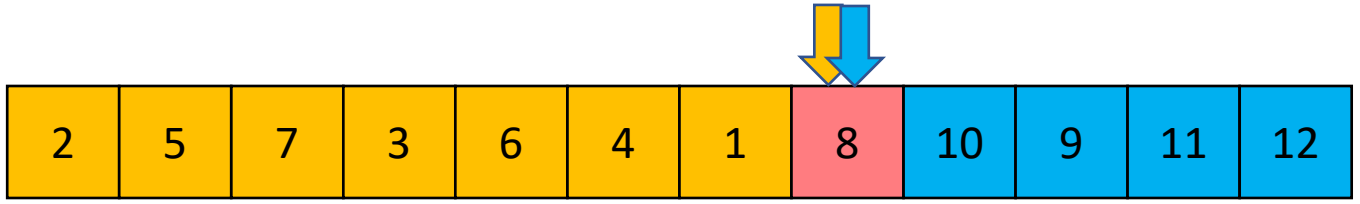
Else swap Begin value with End value, move End Left

Done when Begin = End



Case 1: meet at element $< p$

Swap p with pointer position (2 in this case)

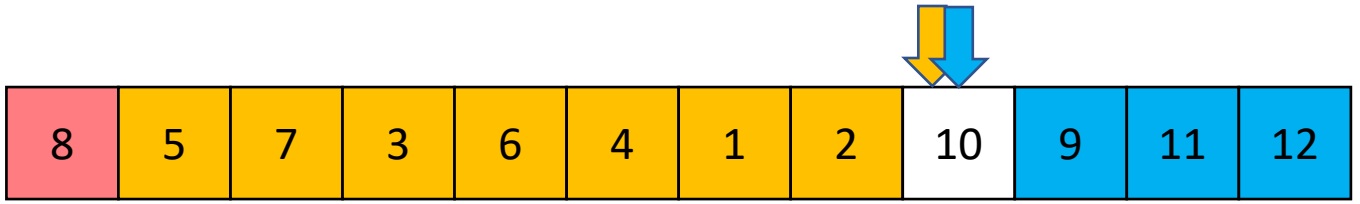


Partition – Begin Meets End (Case 2)

If Begin value $<$ p, move Begin right

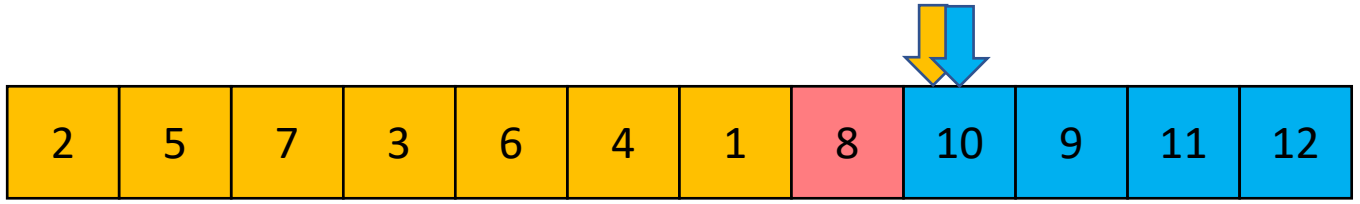
Else swap Begin value with End value, move End Left

Done when Begin = End



Case 2: meet at element $> p$

Swap p with value to the left of pointer (2 in this case)



Partition Summary

1. Put p at beginning of list
2. Put a pointer (Begin) just after p, and a pointer (End) at the end of the list
3. While Begin < End:
    1. If Begin value $< p$, move Begin right
    2. Else swap Begin value with End value, move End Left
4. If pointers meet at element $< p$: Swap p with pointer position
5. Else If pointers meet at element $> p$: Swap p with value to the left

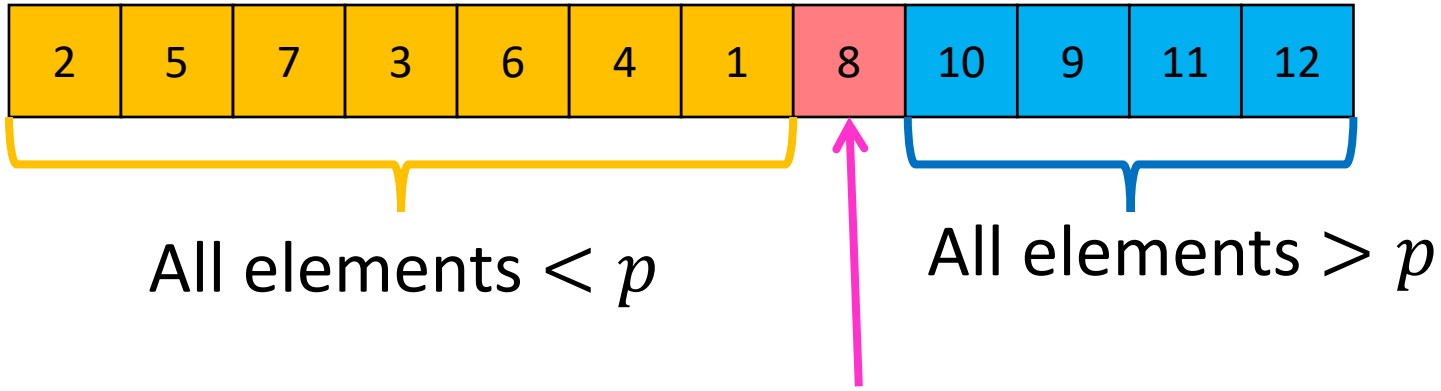
Run time?

Partition Summary (Answers)

1. Put p at beginning of list
2. Put a pointer (Begin) just after p, and a pointer (End) at the end of the list
3. While Begin < End:
    1. If Begin value $< p$, move Begin right
    2. Else swap Begin value with End value, move End Left
4. If pointers meet at element $< p$: Swap p with pointer position
5. Else If pointers meet at element $> p$: Swap p with value to the left

Run time? $O(n)$

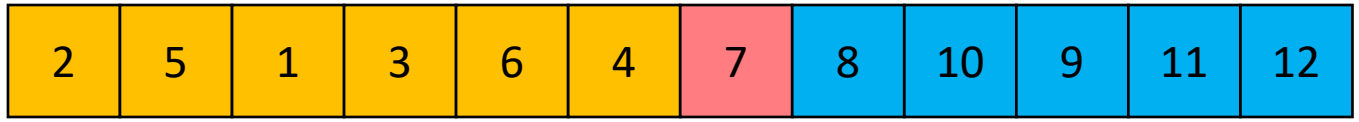
Conquer

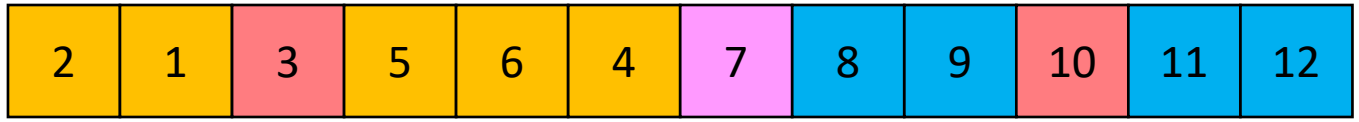


Recursively sort Left and Right sublists

Quicksort Run Time (Best)

If the pivot is always the median:





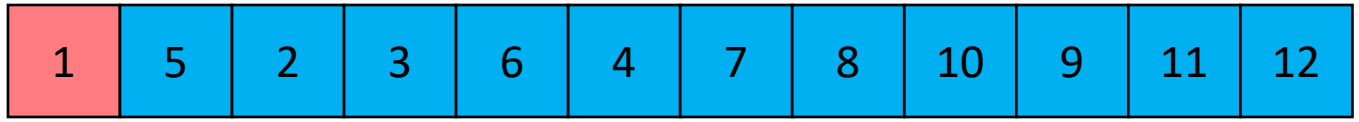
Then we divide in half each time

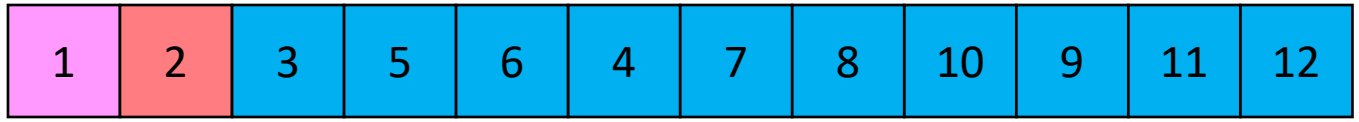
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T(n) = O(n \log n)$$

Quicksort Run Time (Worst)

If the pivot is always at the extreme:





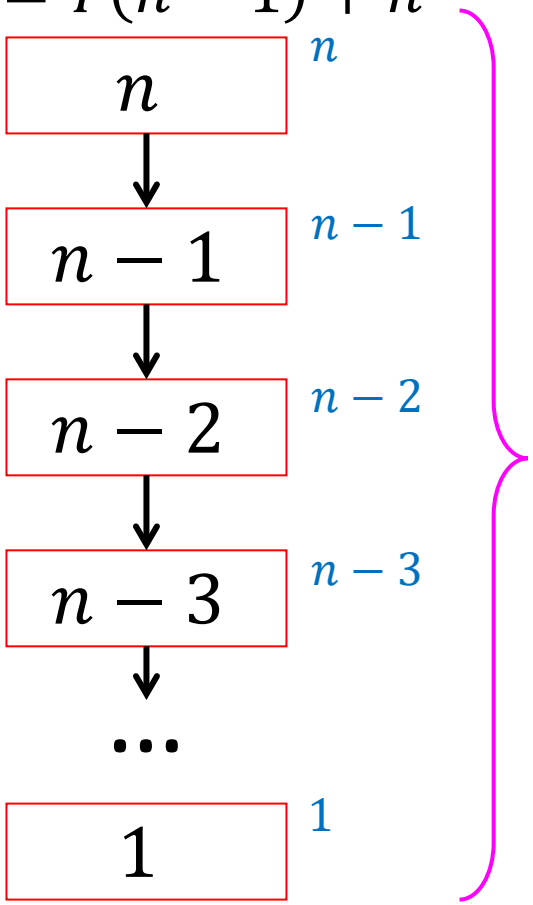
Then we shorten by 1 each time

$$T(n) = T(n-1) + n$$

$$T(n) = O(n^2)$$

Quicksort Worst Case Tree Method

$T(n) = T(n-1) + n$



n levels of recursion

$$T(n) = \sum_{i=0}^{n-1} n - i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= \Theta(n^2)$$

Picking a Good Pivot

- What makes a good Pivot?
  - Roughly even split between left and right
  - Ideally: median
- There are ways to find the median in linear time, but it's complicated and slow and you're better off using mergesort
- In Practice:
  - Pick a "random" value as a pivot
  - Median of Three: Pick the middle of 3 "random" values as the pivot
    - E.g. Compare the elements in the: first, last and middle locations of the array; Pick the middle of these 3 values. Can be done in constant time!

Quick Sort Properties

- Worst case running time
  - $\Theta(n^2)$, but "almost always" $\Theta(n \log n)$ in practice
  - Better constants than merge sort
- In place:
  - Kinda…
  - We swap within the given array, but do so using recursion, so we need space for the stack frames
  - Different textbooks disagree on whether call-stack space "counts" towards your space use
- Adaptive
  - NO!
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - NO!
  - Swapping during Partition procedure may rearrange tied elements

How fast can we sort?

- Insertion Sort:
- Heap Sort:
- Merge Sort:
- Quick Sort:

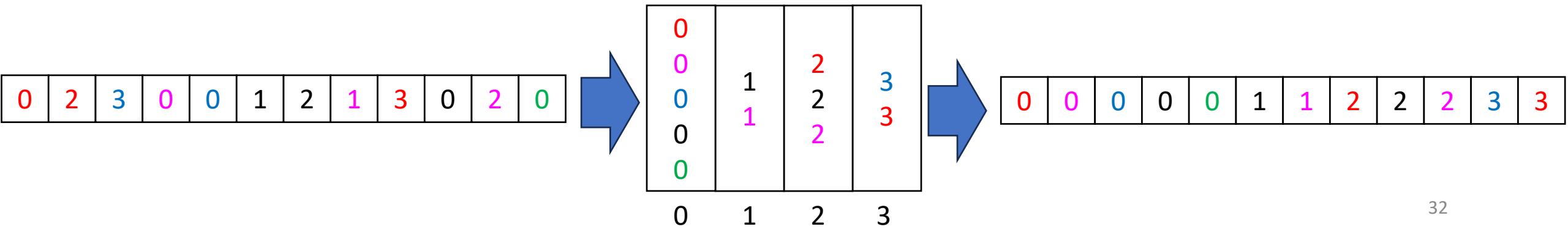
- These are all **comparison-based** sorts. All you need is the ability to compare elements.
- No comparison-based sort can have a worst case running time faster than $\Theta(n \log n)$ (See Weiss for proof)

“Linear Time” Sorting Algorithms

- Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  - Examples:
    - The list contains only positive integers less than k
    - The number of distinct values in the list is much smaller than the length of the list
- The running time expression will always have a term other than the list’s length to account for this assumption
  - Examples:
    - Running time might be $\Theta(k + n)$ or $\Theta(k * n)$ where k is the range/count of values

BucketSort

- Assumes the array contains integers between 0 and $k - 1$ (or some other small range)
- Idea:
  - Use each value as an index into an array of size k
  - Add the item into the “bucket” at that index (e.g. linked list queue)
  - Get sorted array by “appending” all the buckets



BucketSort Running Time

- Create array of k buckets
  - Either $\Theta(k)$ or $\Theta(1)$ depending on some things…
- Insert all n things into buckets
  - $\Theta(n)$
- Empty buckets into an array
  - $\Theta(n + k)$
- Overall:
  - $\Theta(n + k)$
- When is this better than mergesort?

Properties of BucketSort

- In-Place?
  - No
- Adaptive?
  - No
- Stable?
  - Yes!

~~Bucket Sort this list?~~ 191, 124, 423

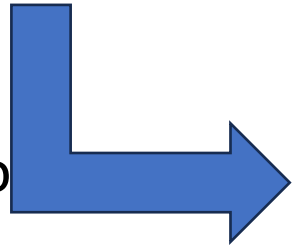
- Instead: do 3 stable sorting passes, first on 1s digit, then 10s digit, then 100s digit

RadixSort

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

103	801	401	323	255	823	999	101	113	901	555	512	245	800	018	121
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Place each element into a "bucket" according to its 1's place



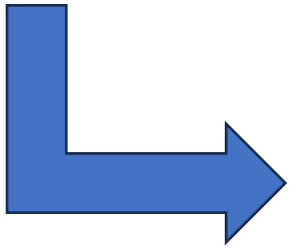
0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

RadixSort – Re-Bucket by 10s

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

Place each element into a "bucket" according to its 10's place



0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

RadixSort – Re-Bucket by 100s

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

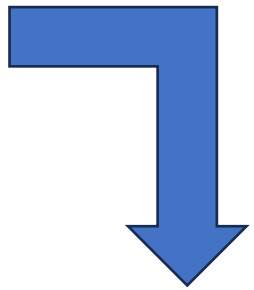
Place each element into a "bucket" according to its 100's place

0	1	2	3	4	5	6	7	8	9
018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999

RadixSort – Empty Buckets into Output Array

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

0	1	2	3	4	5	6	7	8	9
018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999



Convert back into an array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
018	101	103	113	121	245	255	323	401	512	555	800	801	823	901	999

RadixSort Running Time

- Suppose largest value is m
- Choose a radix (base of representation) b
- BucketSort all n things using b buckets
  - $\Theta(n + b)$
- Repeat once per each digit
  - $\log_b m$ iterations
- Overall:
  - $\Theta(n \log_b m + b \log_b m)$
- In practice, you can select the value of b to optimize running time
- When is this better than mergesort?

Sorting Summary

- Insertion Sort $\Theta(n^2)$
  - can be fastest for small *n*
  - good for "below a cut-off" to help divide-and-conquer sorts
- $\Theta(n\ log\ n)$ sorts
  - Heap sort: in-place but not stable
  - Merge sort: not in place but stable
  - Quick sort: "in place" but not stable and $\Theta(n^2)$ in worst-case
    - often fastest
- $\Omega(n\ log\ n)$ is worst-case lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort: good for small number of key values
  - Radix sort: uses fewer buckets and more passes
- Best way to sort? It depends!