CSE 332 Summer 2026 Lecture 11: Sorting 2

Michael Whitmeyer

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

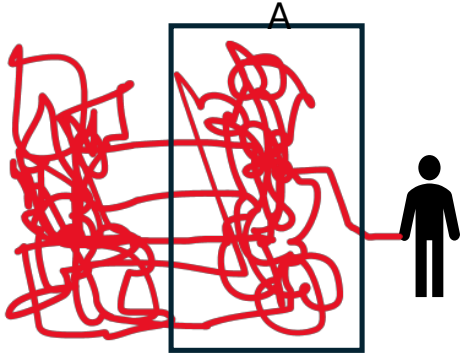
Properties To Consider

- 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 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

Merge Sort Pseudocode

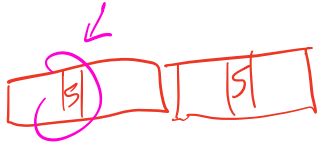
```
mergesort(L):
        ms_helper(L, 0, L.length())

mshelper(L, low, high):
        IF (low == high) RETURN // Base Case
        mid = (low+high)/2
        ms_helper(low, mid)
        ms_helper(mid+1, high)
        merge(L, low, mid, high)
```

L is length 1

Merge Sort Properties

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

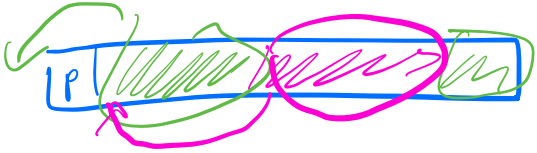
- Worst case running time
  - $\Theta(n \log n)$
- In place:
  - NO!
  - We need a second array to merge into
- Adaptive
  - NO!
  - Running time is the same regardless of original list's order
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - YES!
  - When merging, and there's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



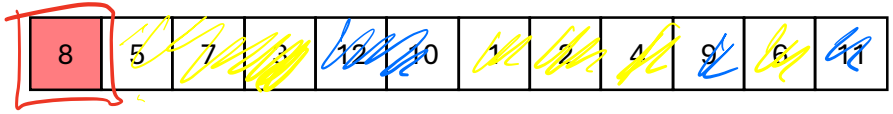
Idea: pick a pivot element, recursively sort two sublists around that element

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

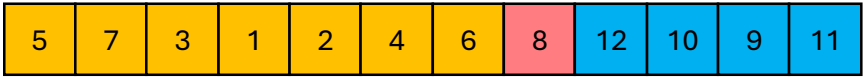
Partition (Divide step)

Given: a list, a pivot p

Start: unordered list

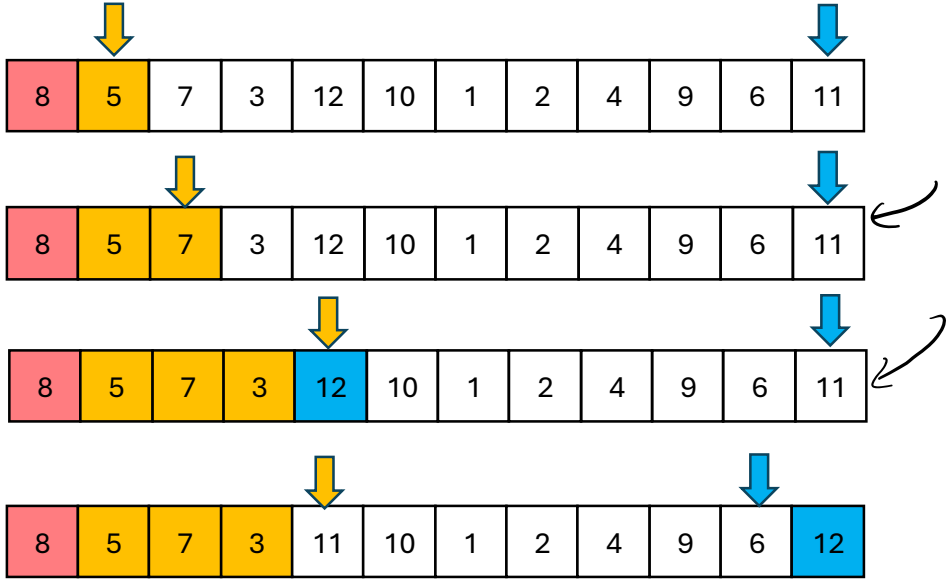


Goal: All elements $< p$ on left, all $> p$ on right



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

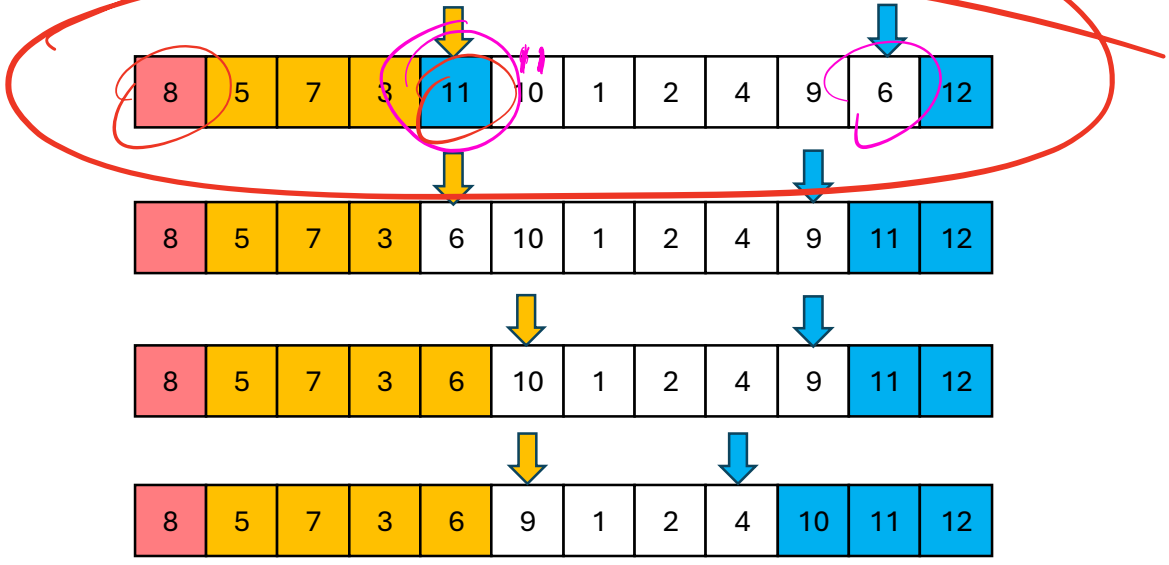


Partition Procedure (Slide 2 of 2)

IF Begin value $<$ p, move Begin right

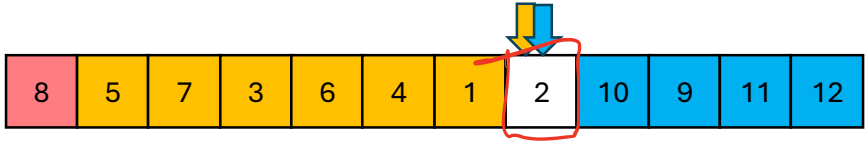
ELSE swap Begin value with End value, move End Left

Done when Begin = End



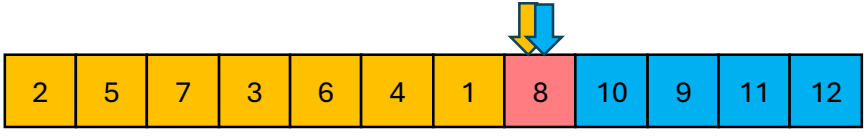
Partition – Begin Meets End (Case 1)

If Begin value $<$ p, move Begin right
Else swap Begin value with End value, move End Left
Done when Begin = End



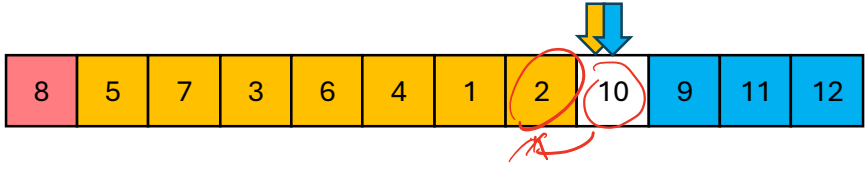
Case 1: meet at element $< p$

Swap p with pointer position (2 in this case)



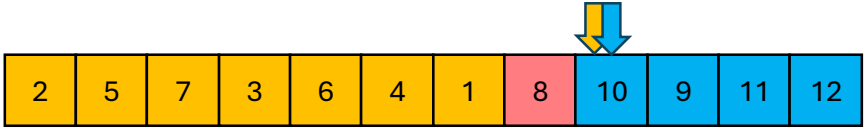
Partition – Begin Meets End (Case 2)

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 (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

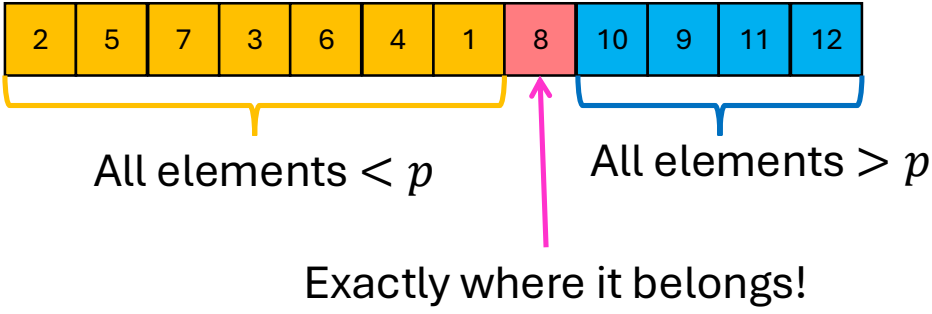
Run time? $O(n)$

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? $O(n)$

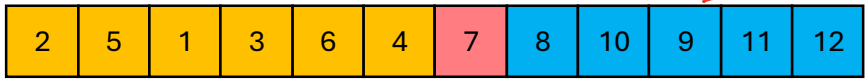
Conquer

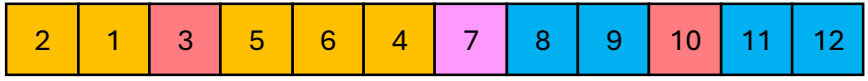


Recursively Quicksort™ Left and Right sublists

Quicksort Run Time (Best)

If the pivot is always the median:





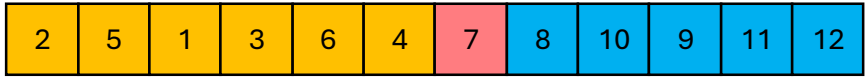
Then we divide in half each time

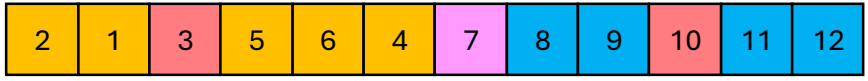
$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

$$= O(n \lg n)$$

Quicksort Run Time (Best)

If the pivot is always the median:





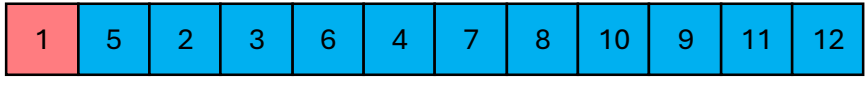
Then we divide in half each time

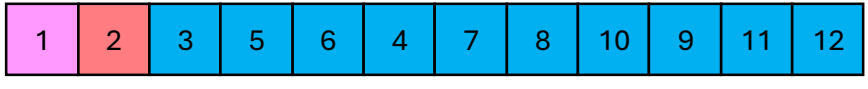
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

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

Quicksort Run Time (Worst)

If the pivot is always at the extreme:





Then we shorten by 1 each time

$$T(n) = n + T(n-1) = O(n^2)$$

$n + T(n-1)$

$+ n-1 + T(n-2)$

$+ n-2 + T(n-3)$

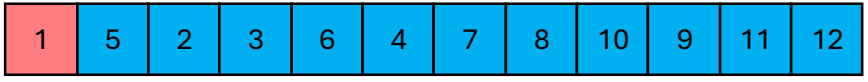
$n + n-1 + n-2 + \cdots + 1$

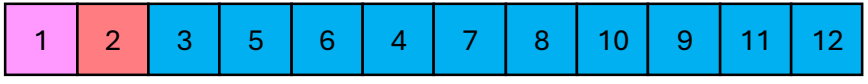
$= \sum_{i=1}^{n} i$

$= \frac{n(n+1)}{2}$

Quicksort Run Time (Worst)

If the pivot is always at the extreme:



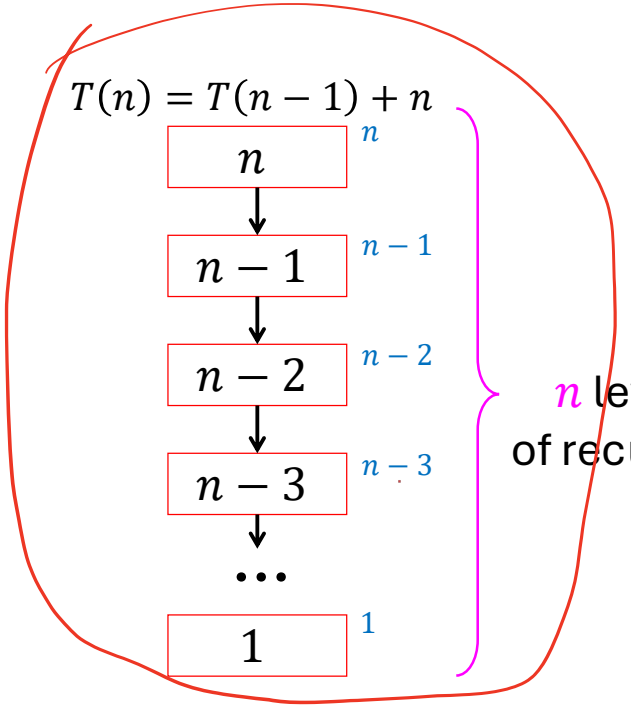


Then we shorten by 1 each time

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

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

Quicksort Worst Case Tree Method



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

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

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
  - Pick the middle of 3 random values as the pivot

$O(n)$

421

$T(n) = n + T(\frac{n}{4}) + T(\frac{3n}{4})$

Quick Sort Properties

- Worst case running time? $O(n^2)$
- In place? controversial
- Adaptive? NO!
- Online? no
- Stable? no.

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"
- Adaptive
  - NO!
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - NO!
  - Partition procedure may rearrange tied elements

$O(n \log n)$

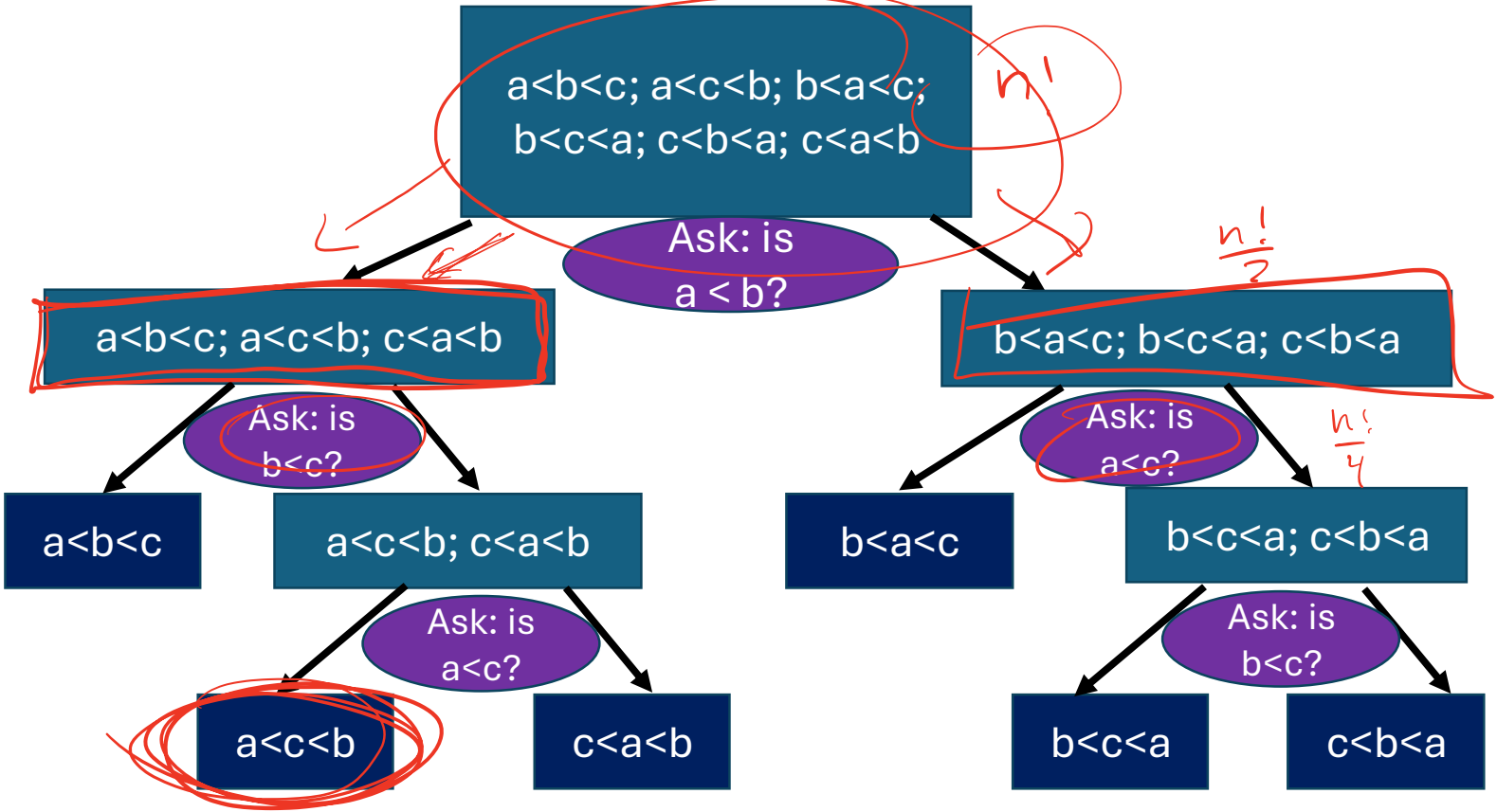
- We keep seeing a worst case $O(n \log n)$ runtime
  - Merge, heap, quick (with a good pivot)
- Can we do better?

Theorem: $\Omega(n \log n)$

- We keep seeing a worst case $O(n \log n)$ runtime
  - Merge, heap, quick (with a good pivot)
- Can we do better?
- **Theorem:** Any comparison-based Sorting algorithm requires $\Omega(n \log n)$ comparisons

Proving the Lower Bound

- **Theorem:** Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons
- Proof uses **"decision trees"**
  - A tree showing the decisions our code will make
  - Nodes of tree labelled with "$a < b$?"


a<b<c; a<c<b; b<a<c; b<c<a; c<b<a; c<a<b
n!
Ask: is a < b?
a<b<c; a<c<b; c<a<b
n!/2
b<a<c; b<c<a; c<b<a
Ask: is b<c?
Ask: is a<c?
n!/4
a<b<c
a<c<b; c<a<b
b<a<c
b<c<a; c<b<a
Ask: is a<c?
Ask: is b<c?
a<c<b
c<a<b
b<c<a
c<b<a

Proving the Lower Bound

- **Theorem:** Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons
- **Proof:**
  - **Worst case ~~runtime~~ =** # of comparisons = height/depth of tree
  - **# leaves = # permutations =** $n!$
  - **Depth lower bound =**

Key observation: at every internal node, $\exists$ child with $\geq \frac{1}{2}$ of the remaining possible permutations.

$$\log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{n/2}\right) \geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right)$$

$$= \Omega(n \log n)$$

Proving the Lower Bound

- **Theorem:** Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons
- **Proof:**
  - **Worst case runtime =** height of tree
  - **# leaves = # permutations =** $n!$
  - **Depth lower bound =** $log_2(n!)$

Calulation: $\log_2(n!) = \log_2(n) + \log_2(n-1) + \log_2(n-2) + \cdots + \log_2(1)$

$\geq \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2}\right) + \cdots + \log_2\left(\frac{n}{2}\right)$ (only $n/2$ copies)

$\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) = n/2(\log_2(n) - 1) = \Omega(n \log n)$

“Linear Time” Sorting Algorithms

integers are small.

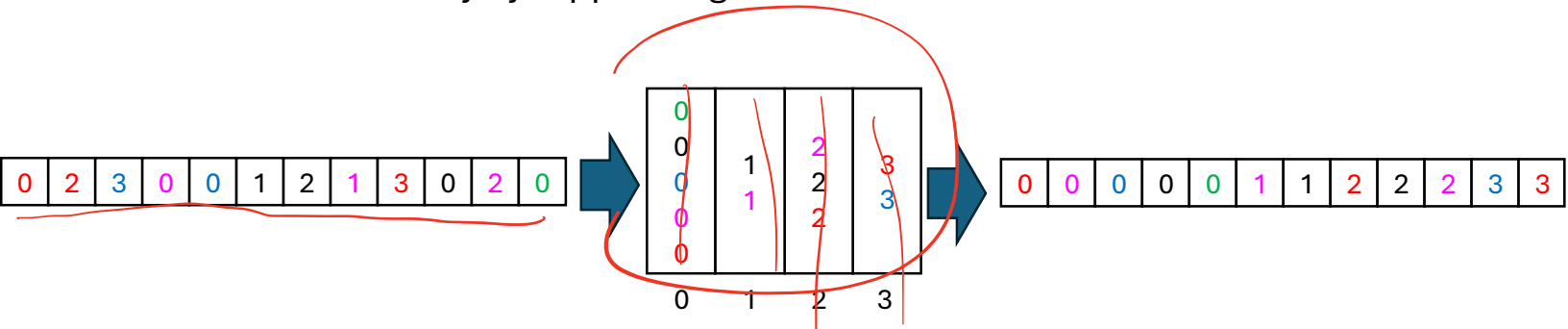
- Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  - Examples:
    - The list contains only positive integers less than k
    - The number of distinct values in the list is much smaller than the length of the list
- The running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

Sleep sort: make timers for each integer, when timer go off, add to array.

O(n)

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)
  - 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?
- Adaptive?
- Stable?

Properties of BucketSort

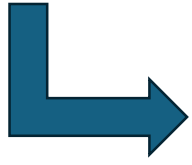
- In-Place?
  - No
- Adaptive?
  - No
- Stable?
  - Yes!

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

10	80	40	32	25	82	99	10	11	90	55	51	24	80	01	12
3	1	1	3	5	3	9	1	3	1	5	2	5	0	8	1
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



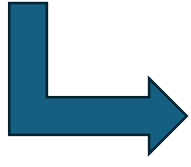
0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

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?