

Non-Comparison Sorting Algorithms

CSE 332 Summer 2021

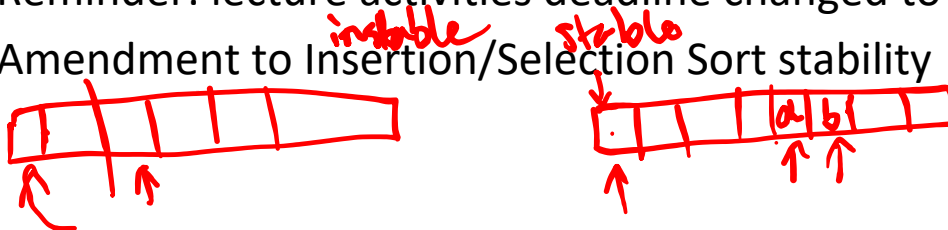
Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- ❖ Midterm
- ❖ P1 Grades out. Regrade Requests
- ❖ Ex7: videos tonight!
 - Reflection q's cannot be edited after submissions came in, so a new reflection assignment will be posted. ***Disregard the reflections section in the Exercise 7 gradescope assignment. A new one will be posted!***
- ❖ Office Hours today pushed to 11:10 so I can release midterm
- ❖ Reminder: lecture activities deadline changed to 7am
- ❖ Amendment to Insertion/Selection Sort stability



Lecture Outline

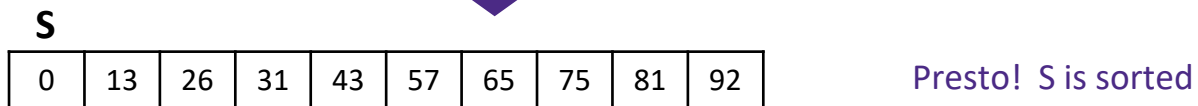
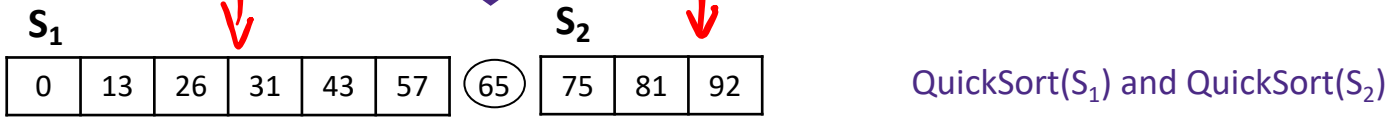
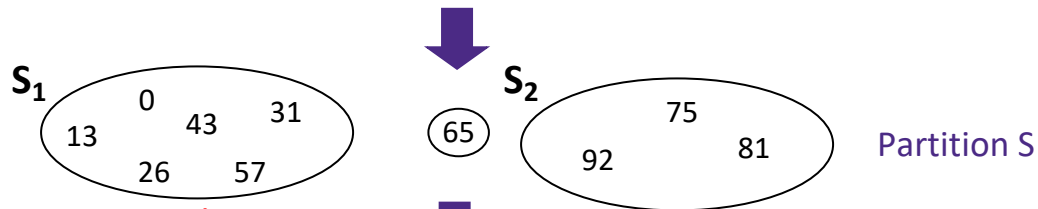
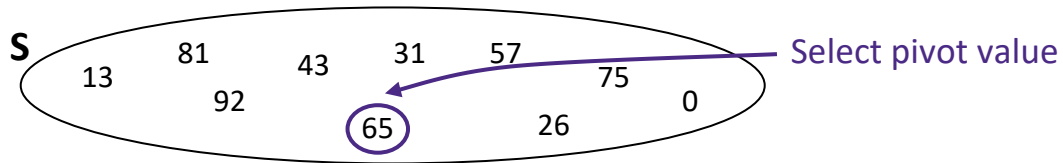
- ❖ **QuickSort**
 - Comparison with MergeSort
- ❖ Comparison Sort Lower Bound
- ❖ Non-Comparison Sorts
 - BucketSort
 - RadixSort
- ❖ Sorting Wrapup

QuickSort Steps

1. Pick the pivot value(s)
 - Any choice is correct; data will end up sorted
 - For efficiency, these value(s) ought to approximate the median
2. Partition all the values into:
 - a. The values less than the pivot(s)
 - b. The pivot(s)
 - c. The values greater than the pivot(s)
 - d. .. In linear time? In-place? Stably?
3. Recursively QuickSort(A) and QuickSort(C)

 TA-DA!

QuickSort Intuition: Set Partitioning



QuickSort Steps

1. Pick the pivot value(s)

- Any choice is correct; data will end up sorted
- For efficiency, these value(s) ought to approximate the median

→ 2. Partition all the values into:

- a. The values less than the pivot(s)
- b. The pivot(s)
- c. The values greater than the pivot(s)
- d. ... In linear time? In-place? Stably?

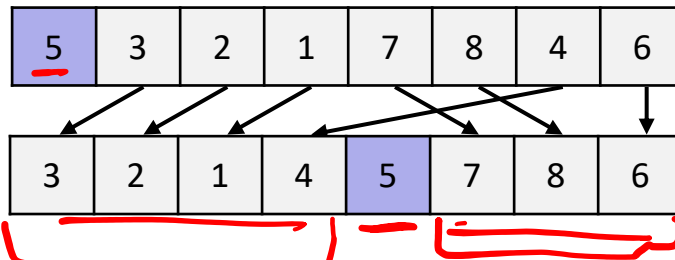
3. Recursively QuickSort(A) and QuickSort(C)

✨TA-DA!✨

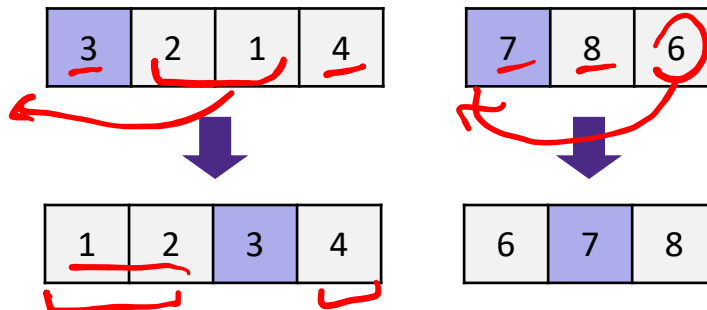
Recursive Call (1 of 3)

Note: for the remainder of this section, our pivot-selection algorithm is “first item in the subarray”

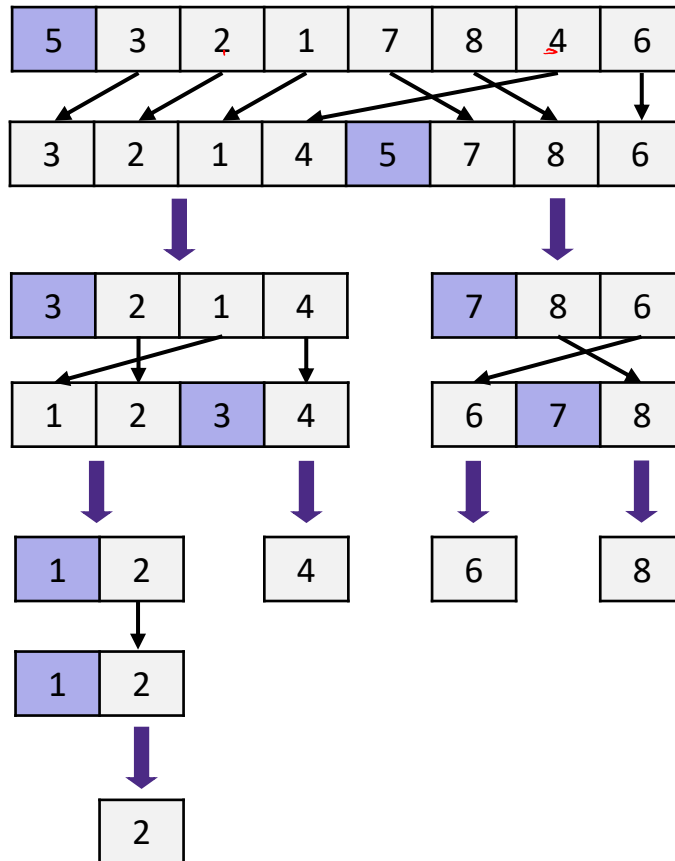
- ❖ After partitioning on 5:
 - 5 is in its “correct place” (ie, where it’d be if the array were sorted)



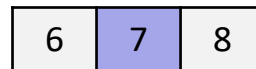
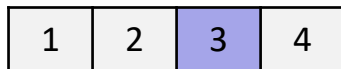
- Can now sort two halves separately (eg, through recursive use of partitioning)



Recursive Call (2 of 3)



Recursive Call (3 of 3)



QuickSort Steps

1. Pick the pivot value(s)

- Any choice is correct; data will end up sorted
- For efficiency, these value(s) ought to approximate the median

2. Partition all the values into:

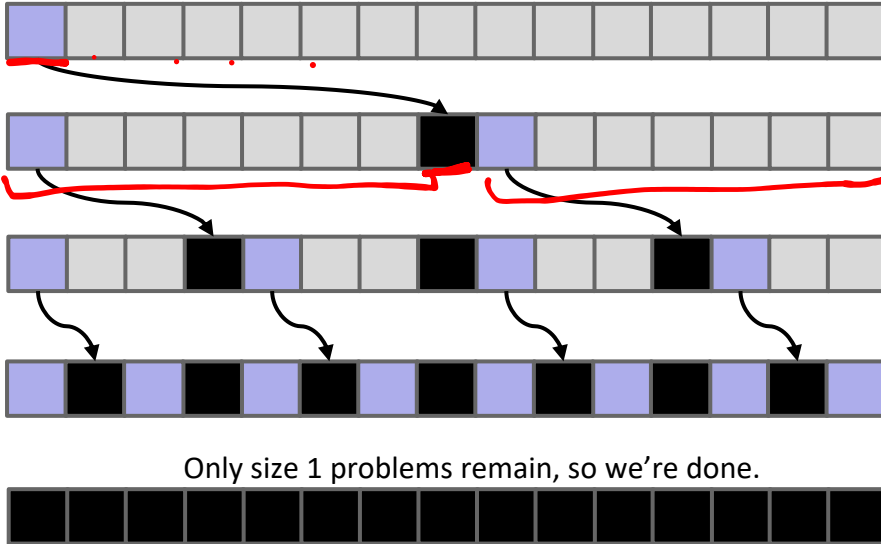
- a. The values less than the pivot(s)
- b. The pivot(s)
- c. The values greater than the pivot(s)
- d. ... In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

✨TA-DA!✨

Pivot Selection: Pivot is the Median

$$2T(n/2) \Rightarrow \log n$$



$$T(0) = T(1) = c_1$$

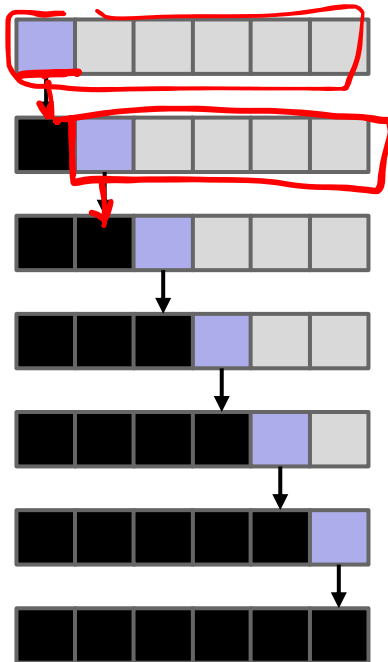
$$T(n) = 2T(n/2) + c_2 n$$

(partition is linear-time)

Same recurrence as MergeSort:

$$O(n \log n)$$

Pivot Selection: Pivot is the Min/Max



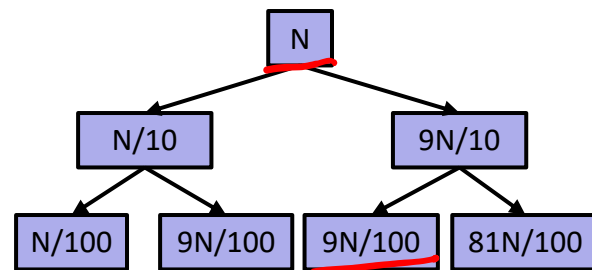
$$T(0) = T(1) = c_1$$

$$T(n) = \underline{T(n-1)} + c_2n$$

Basically same recurrence as
SelectionSort: $O(n^2)$

Pivot Selection: Pivot is Random

- ❖ Suppose pivot always ends up *at least 10% from either edge*



- ❖ Work at each level: $O(N)$ and Runtime is $O(NH)$
 - Height is approximately $\log_{10/9} N = O(\log N)$
- ❖ Runtime: $O(N \log N)$
 - See proof in text

Pivot Selection Dictates Runtime!

❖ If pivot lands “somewhere good”, Quicksort is $\Theta(N \log N)$

❖ However, the very rare $\Theta(N^2)$ cases do happen in practice 🙄

→ **Bad ordering:** Array already in (almost-)sorted order and pivot is first or last index

▪ **Bad elements:** Array with all duplicates



❖ Three philosophies for avoiding worst-case behavior:

1. **Randomness:** pick a random pivot; shuffle before sorting

- Elegant, but (pseudo)random number generation can be slow

2. **Smarter Pivot Selection:** calculate or approximate the median

- Median-of-3: median of `arr[l0]`, `arr[hi-1]`, `arr[(hi+l0)/2]`

3. **Introspection:** switch to safer sort if recursion goes too deep

QuickSort Steps

1. Pick the pivot value(s)
 - Any choice is correct; data will end up sorted
 - For efficiency, these value(s) ought to approximate the median

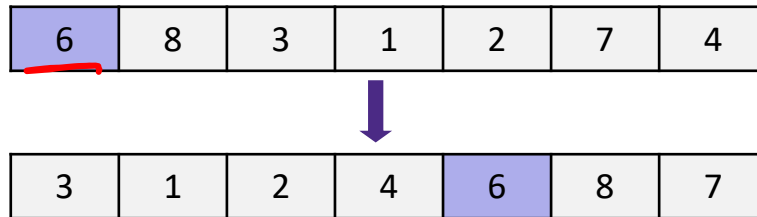
2. Partition all the values into:
 - a. The values less than the pivot(s)
 - b. The pivot(s)
 - c. The values greater than the pivot(s)
 - d. ... In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

✨ TA-DA! ✨

Partitioning: Problem Statement

- Given an array of elements and the 0th value as the pivot, write pseudocode that partitions the array



- Constraints:

- Must complete in $O(N \log N)$ time, but ideally $\Theta(N)$
- Must use $O(N)$ space, but ideally $\Theta(1)$
- May use any data structure (eg, BSTs, stacks/queues, etc)
 - Ideally, preserves the elements' relative ordering ("stable")

- Conceptually simple, but hardest part to code up correctly!

Partitioning: Option 1: Three-Pass

❖ Overview:

- Copy “less than”s, then copy pivot(s), finally copy “greater-than”s

❖ Stable! 😊

❖ Uses extra space 😞

❖ Constants aren't great; very slow 😞

Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array

5	550	10	4	10	9	330
---	-----	----	---	----	---	-----

Stable Three-Pass Partition

- **Allocate an extra array to store output**
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array

5	550	10	4	10	9	330
---	-----	----	---	----	---	-----

--	--	--	--	--	--	--

Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array

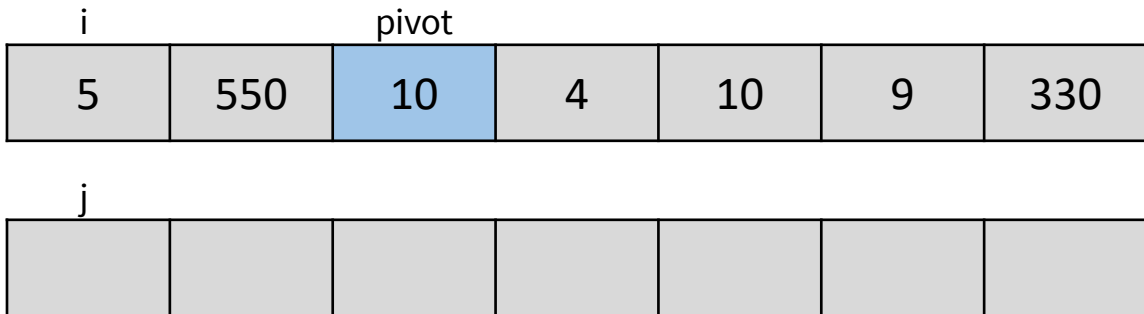
5	550	10	4	10	9	330
---	-----	----	---	----	---	-----

For demo purposes, we are using 10 as our pivot. However, there are many pivot-selection algorithms, including picking the 0th element.

--	--	--	--	--	--	--

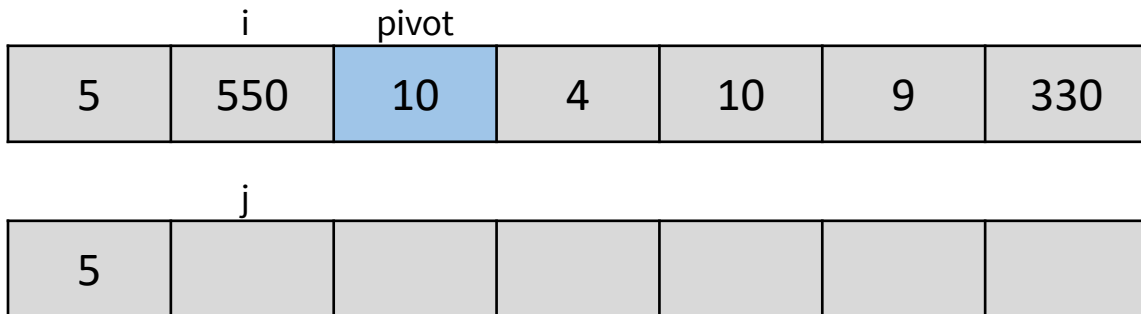
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



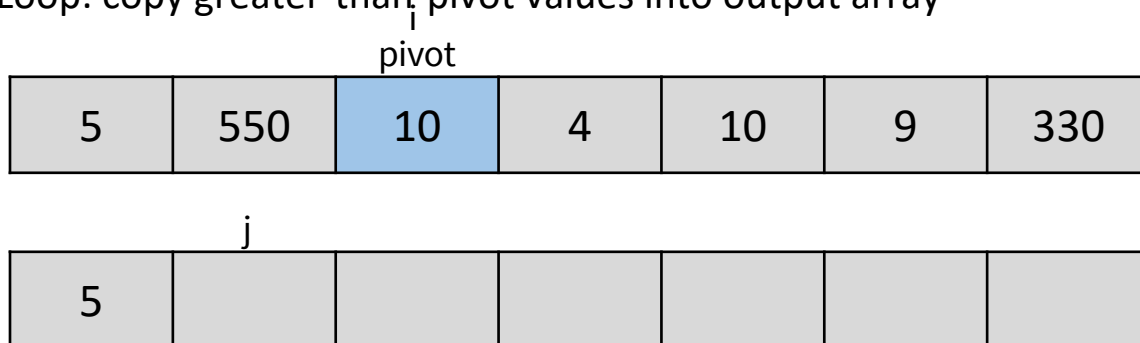
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



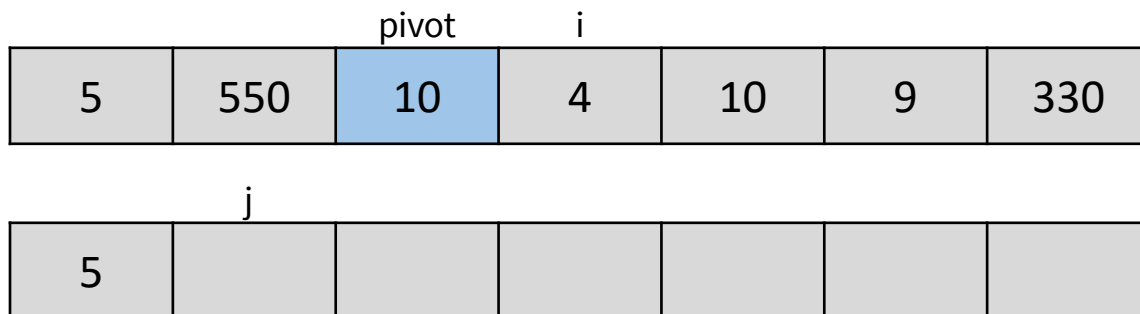
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



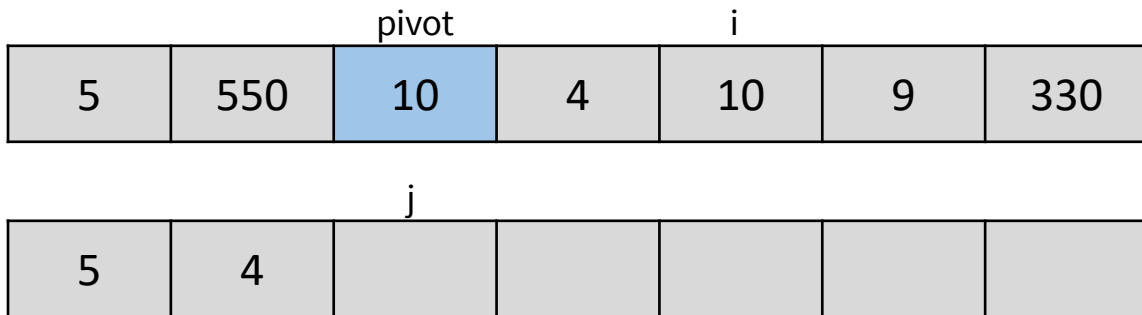
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



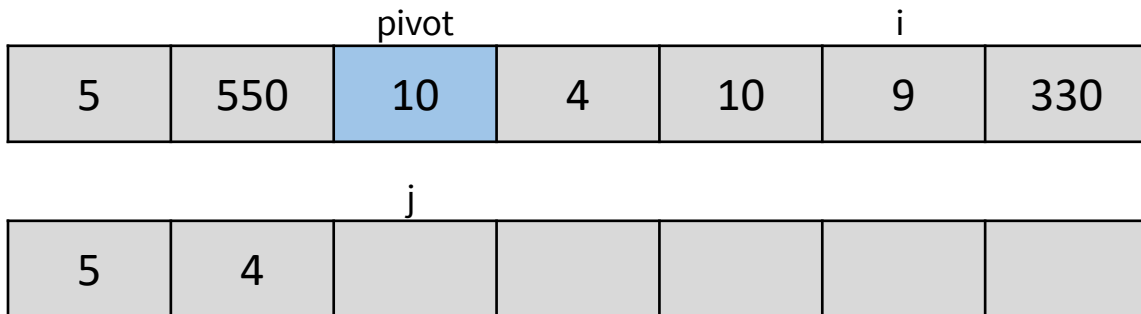
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



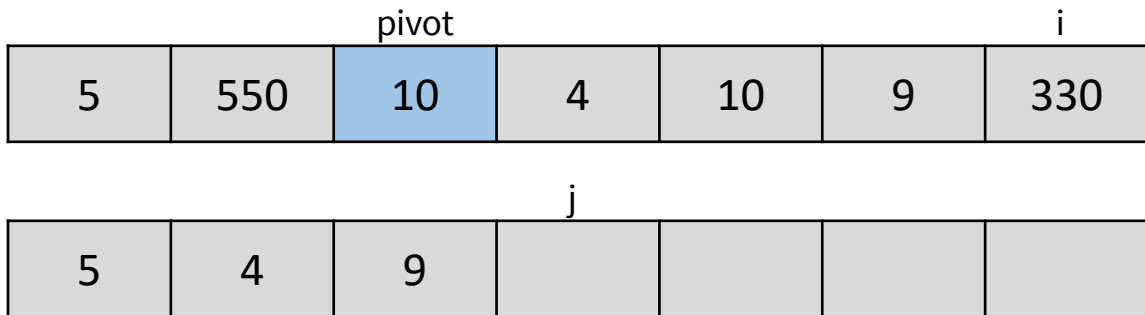
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



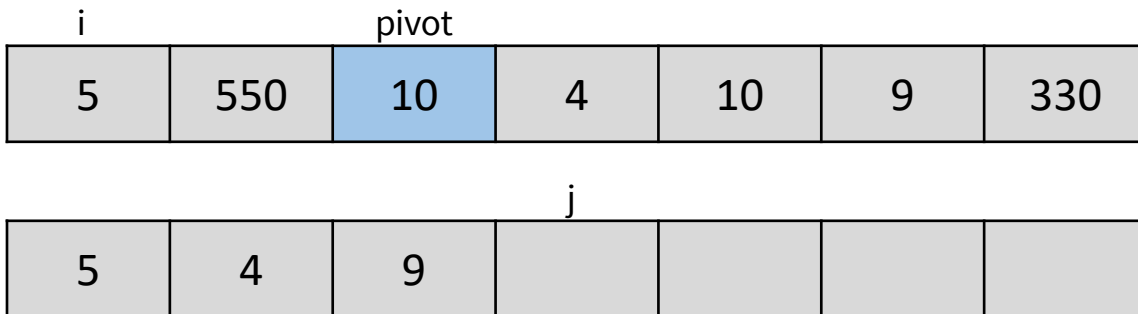
Stable Three-Pass Partition

- Allocate an extra array to store output
- **Loop: copy less-than-pivot values into output array**
- Loop: copy pivot value(s) into output array
- Loop: copy greater-than-pivot values into output array



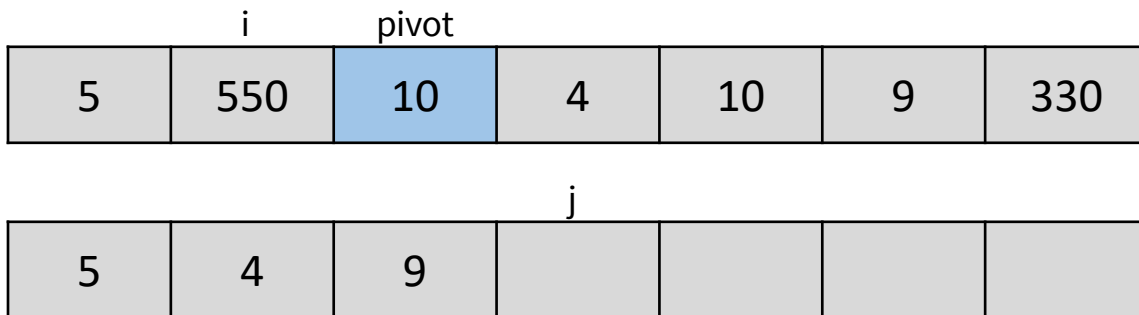
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



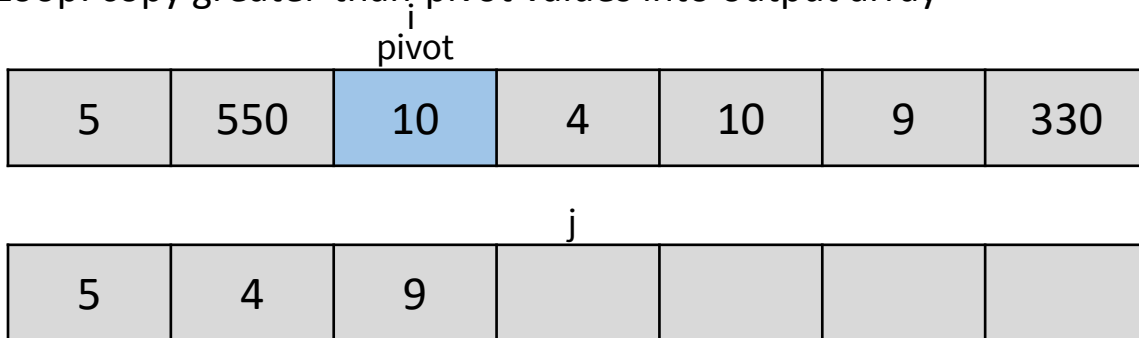
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



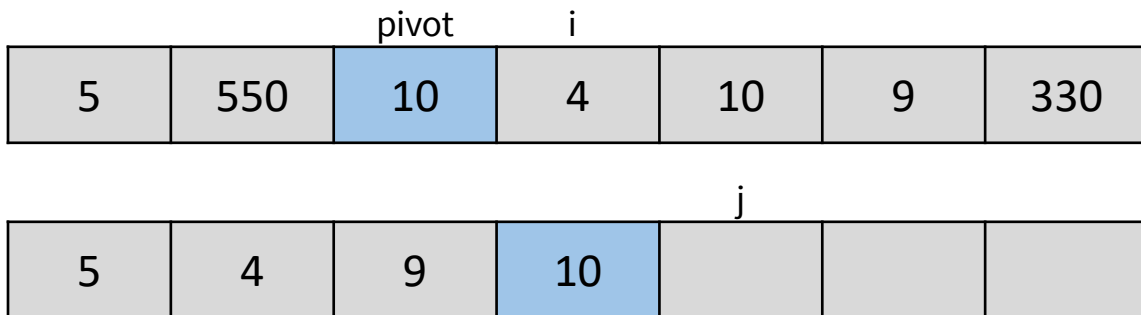
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



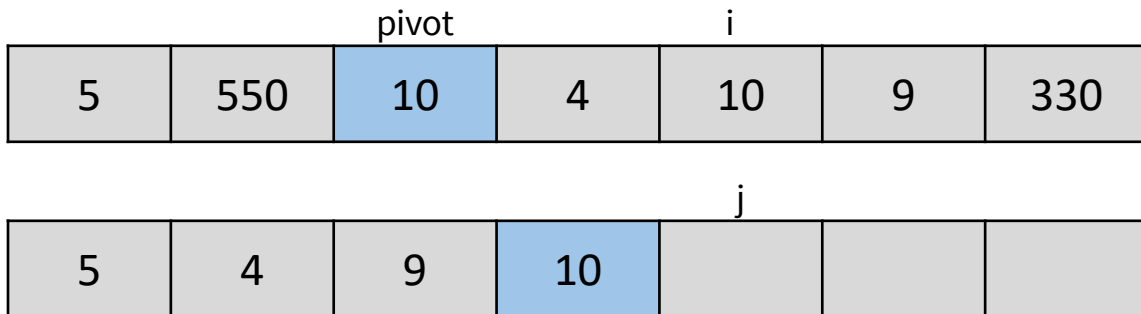
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



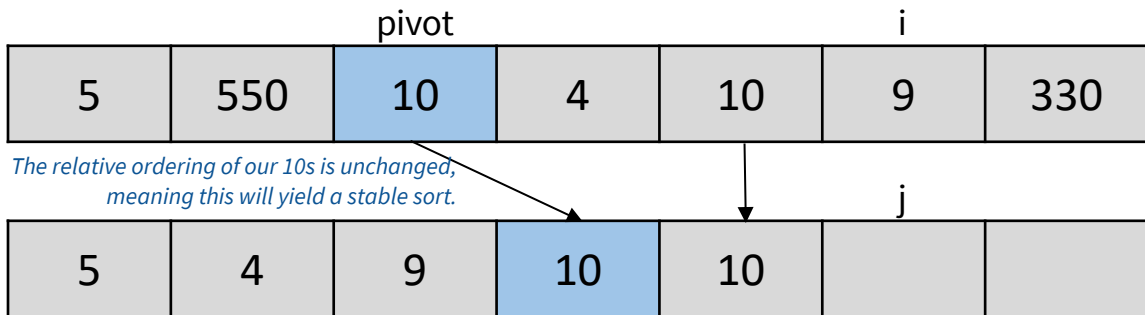
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



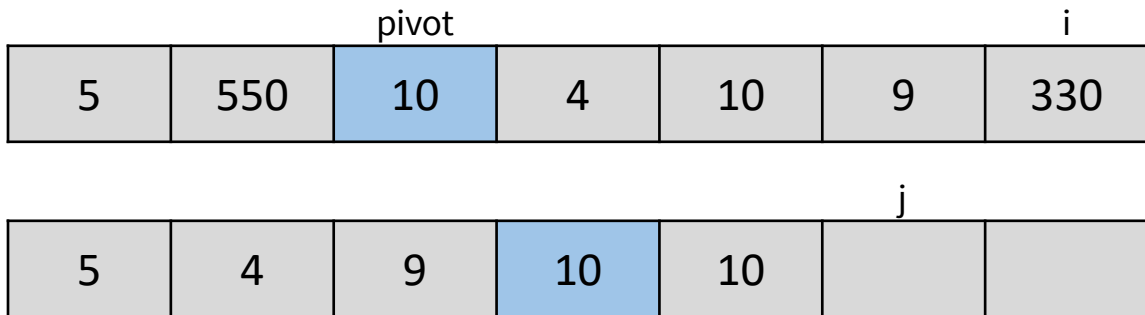
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



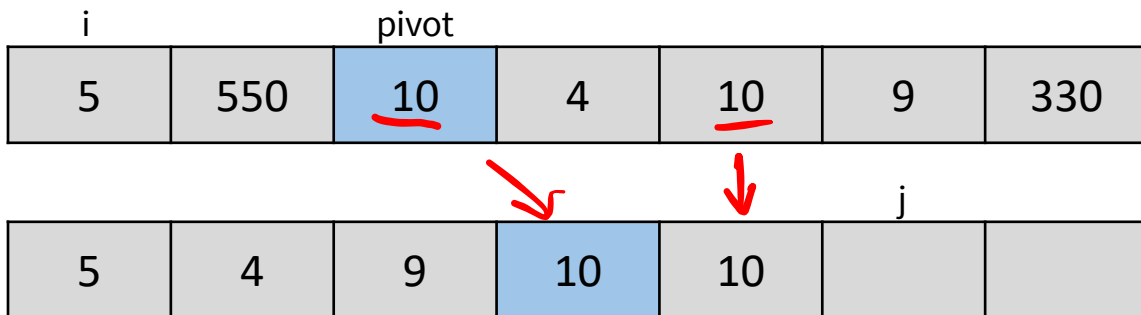
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- **Loop: copy pivot value(s) into output array**
- Loop: copy greater-than-pivot values into output array



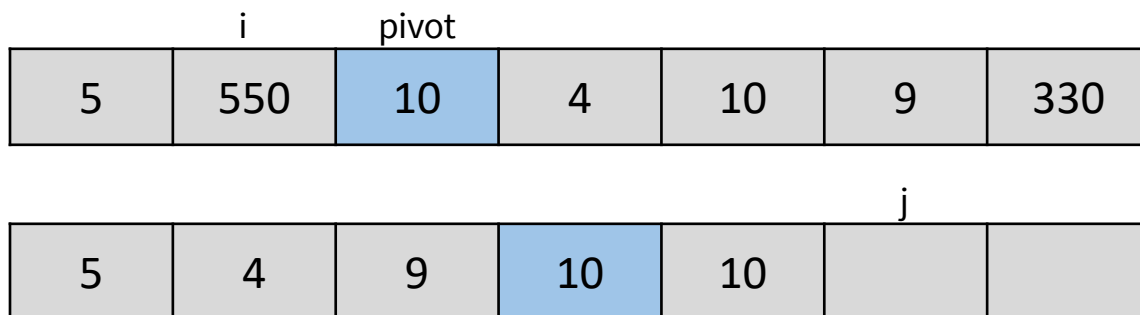
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



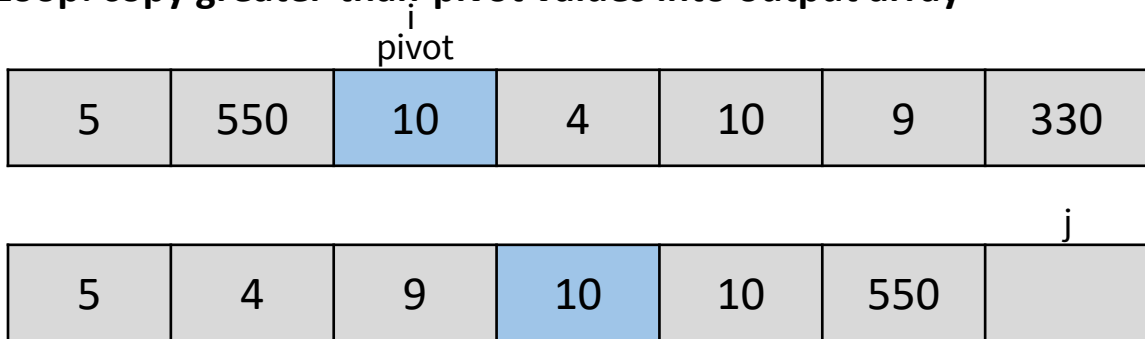
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



Stable Three-Pass Partition

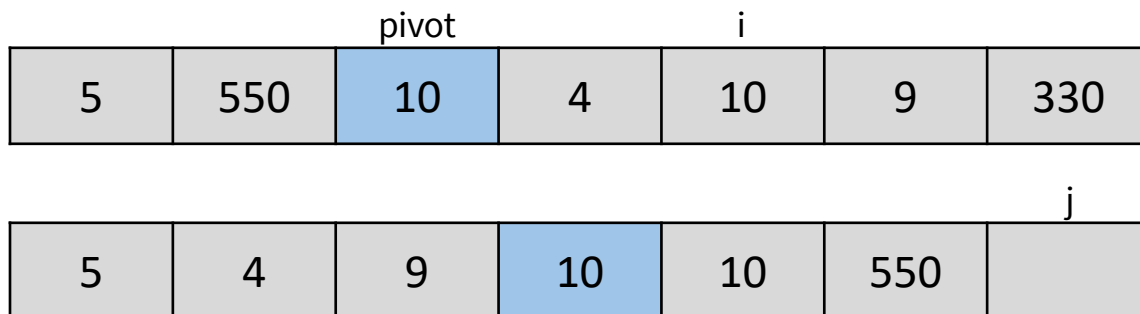
- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**

		pivot	i			
5	550	10	4	10	9	330

						j
5	4	9	10	10	550	

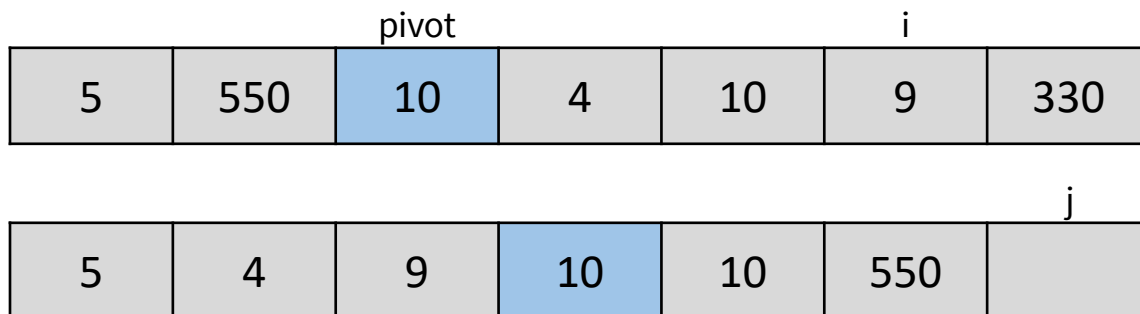
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



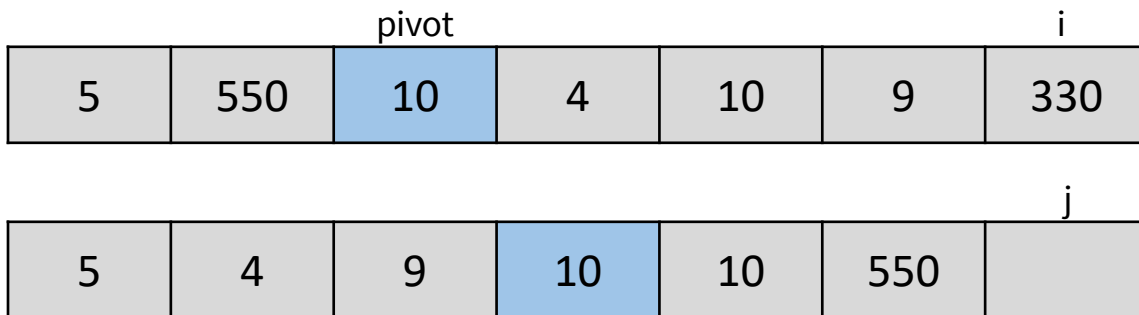
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



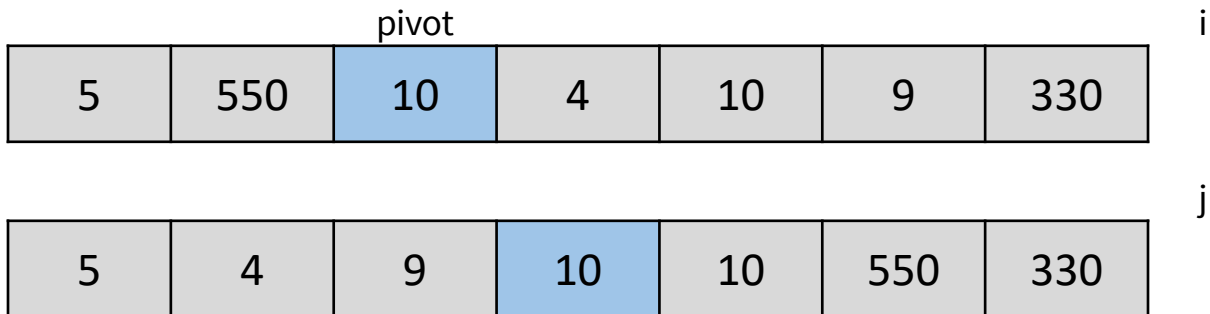
Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



Stable Three-Pass Partition

- Allocate an extra array to store output
- Loop: copy less-than-pivot values into output array
- Loop: copy pivot value(s) into output array
- **Loop: copy greater-than-pivot values into output array**



Done partitioning!

Partitioning: Option 2: Hoare Partitioning

- ❖ As published in Hoare's original QuickSort paper!
- ❖ Intuition:
 - L loves small items (i.e., $<$ pivot) and R loves large items (i.e., $>$ pivot)
 - Walk towards each other, swapping anything they don't like

❖ Algorithm:

1. Swap pivot with arr[lo] ("move it out of the way")
2. Start i at $lo+1$, and j at $hi-1$
3. Move j rightward until we hit value $<$ pivot ("belongs on left")
4. Move i leftward until we hit value $>$ pivot ("belongs on right")
5. Swap $arr[i]$ and $arr[j]$
6. When they meet, swap $arr[0]$ and $arr[i]$ ("put pivot in correct place")

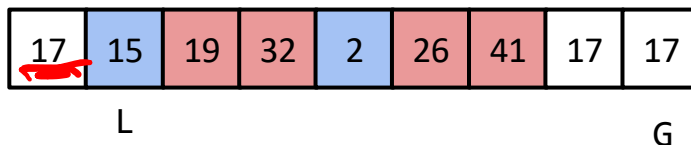
```
while (i < j)
    if (arr[j] > pivot) j--;
    else if (arr[i] <= pivot) i++;
    else swap(arr[i], arr[j])
```

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

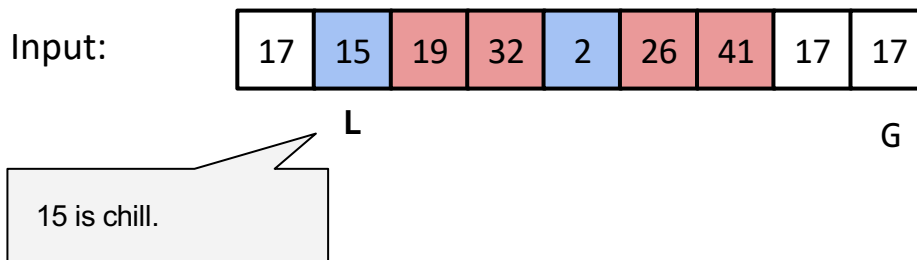
Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

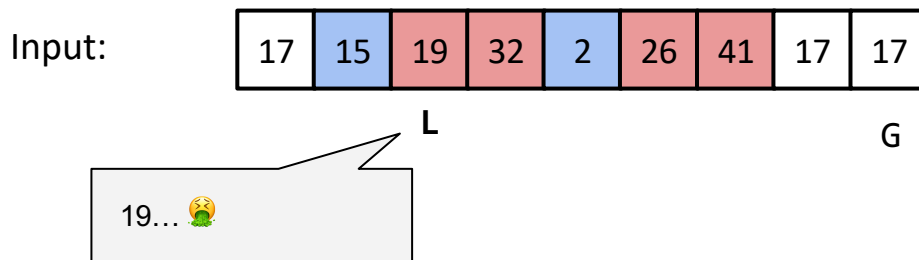
- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other**, stopping on a hated item.
 - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

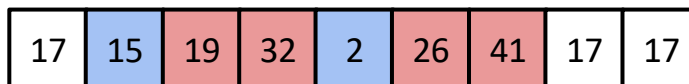


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap.

Input:



L

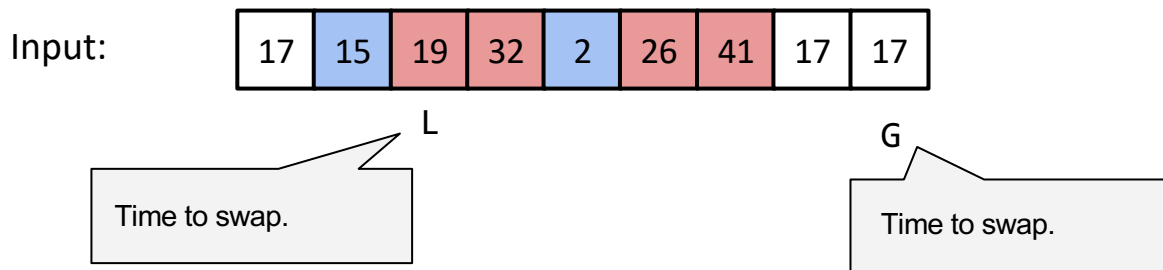
G

17 sucks

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - **When both pointers have stopped, swap and move pointers by one.**
- When pointers cross, you are done.

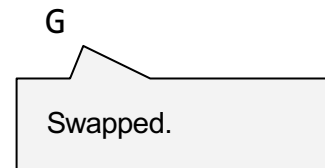
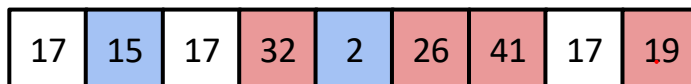


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - **When both pointers have stopped, swap and move pointers by one.**
- When pointers cross, you are done.

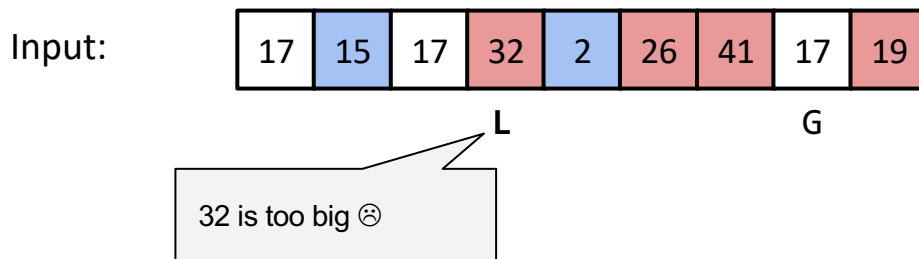
Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

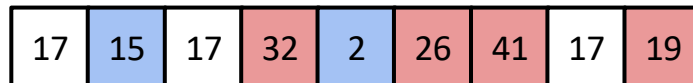


Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

Input:



L

G

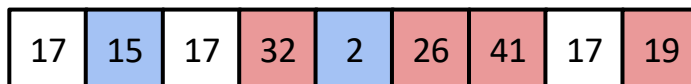
Too small

Hoare Partitioning

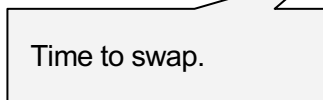
Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - **When both pointers have stopped, swap and move pointers by one.**
 - When pointers cross, you are done walking.
- Swap pivot with G.

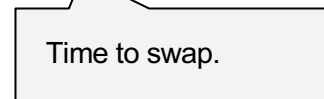
Input:



L



G



Hoare Partitioning

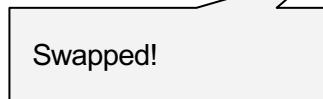
Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - **When both pointers have stopped, swap and move pointers by one.**
 - When pointers cross, you are done walking.
- Swap pivot with G.

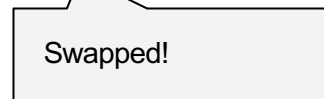
Input:



L



G



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- Swap pivot with G.

Input:



L

G

2 is cool.

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- Swap pivot with G.

Input:



L G

26 thicc. No thx

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- Swap pivot with G.

Input:



L

G

41 you good

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- Swap pivot with G.

Input:



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- **Walk pointers towards each other, stopping on a hated item.**
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- Swap pivot with G.

Input:



G L

2 what u doin here 🤔
also hi L!!

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - When both pointers have stopped, swap and move pointers by one.
 - **When pointers cross, you are done walking.**
- Swap pivot with G.

Input:



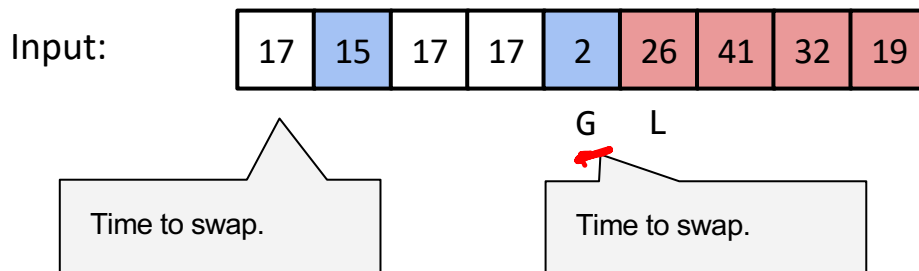
G L

2 what u doin here 🤔
also hi L!!

Hoare Partitioning

Create L and G pointers at left and right ends.

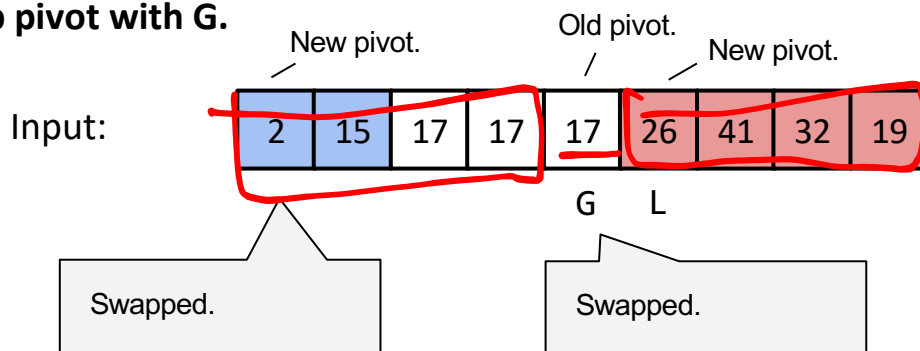
- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- **Swap pivot with G.**



Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
 - When both pointers have stopped, swap and move pointers by one.
 - When pointers cross, you are done walking.
- **Swap pivot with G.**



- ❖ Partition the following array using Hoare's partitioning algorithm
 - The pivot, 5, has already been moved to the front
 - Sort only by the numbers (eg, 1); the extra letter (eg, 1a) is to help you determine stability

5	<u>1a</u>	6	9	3	7	2	<u>4a</u>	<u>4b</u>	<u>1b</u>
---	-----------	---	---	---	---	---	-----------	-----------	-----------

Swap pivot with `arr[lo]` ("move it out of the way")

Start `i` at `lo+1`, and `j` at `hi-1`

Move `j` rightward until we hit value `< pivot` ("belongs on left")

Move `i` leftward until we hit value `> pivot` ("belongs on right")

Swap `arr[i]` and `arr[j]`

When they meet, swap `arr[lo]` and `arr[i]` ("put pivot in correct place")

```
while (i < j)
  if (arr[j] > pivot) j--;
  else if (arr[i] <= pivot) i++;
  else swap(arr[i], arr[j])
```

Notes on Hoare Partitioning

- Unstable 😞
- Good constants: single-pass and in-place 😊

Partitioning: Option 3: Three-Way

- ❖ Pick *two* pivots
 - Same intuition as median-of-three: it's hard to pick multiple bad pivots simultaneously
- ❖ Like Hoare Partitioning, use two pointers walking to the middle
 - But split array into three pieces, not two
 - Good constants: single-pass and in-place; $\log_3 N$ vs $\log_2 N$ 😊
 - Still an unstable sort 😞
- ❖ Used in Java's `Arrays.sort()`, Python's unstable sort, etc
 - Basically the de-facto partition algorithm circa 2020
- ❖ Won't be going through this version

QuickSort Steps

1. Pick the pivot value(s)
 - Any choice is correct; data will end up sorted
 - For efficiency, these value(s) ought to approximate the median

2. Partition all the values into:
 - a. The values less than the pivot(s)
 - b. The pivot(s)
 - c. The values greater than the pivot(s)
 - d. ... In linear time? In-place? Stably?

3. Recursively QuickSort(A) and QuickSort(C)

✨**TA-DA!**✨

QuickSort: End-to-end Example (1 of 3)

- Pick pivot (we'll use median-of-3)

8	1	4	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

- Partition (we'll use Hoare Partitioning)
 - Move pivot to the beginning position

6	1	4	9	0	3	5	2	7	8
---	---	---	---	---	---	---	---	---	---

- Let $lo = 1$ and $hi = 9$; loop until we find "swappable" values

6	1	4	9	0	3	5	2	7	8
---	---	---	---	---	---	---	---	---	---



6	1	4	9	0	3	5	2	7	8
---	---	---	---	---	---	---	---	---	---

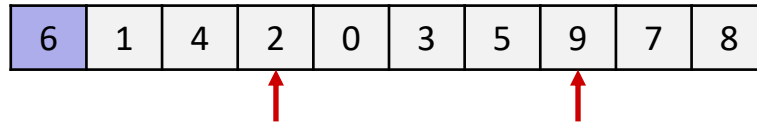


6	1	4	9	0	3	5	2	7	8
---	---	---	---	---	---	---	---	---	---

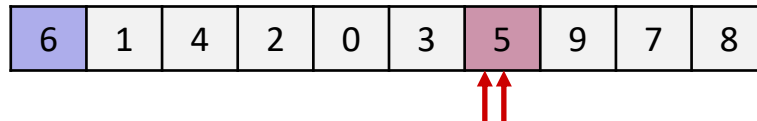
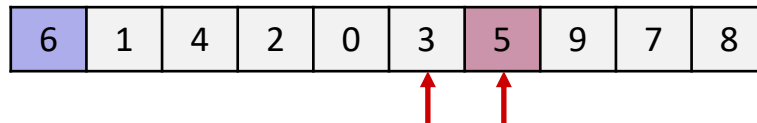
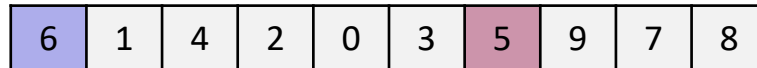


QuickSort: End-to-end Example (2 of 3)

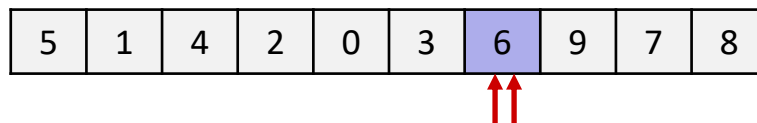
- Swap $l_o = 3$ and $h_i = 7$



- Keep looping

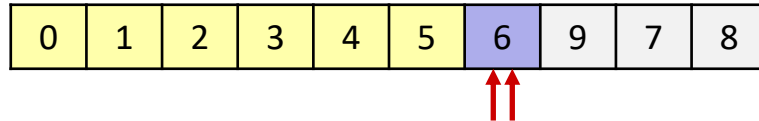


- Done! Swap pivot into position

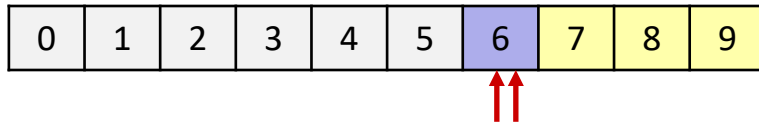


QuickSort: End-to-end Example (3 of 3)

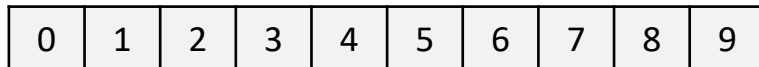
3. Recursively sort left (0 to $lo-1 = 5$)



4. Recursively sort right ($hi+1 = 7$ to `arr.length`)

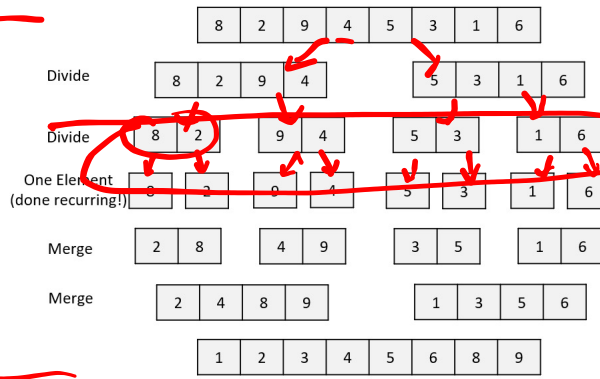


5. Sorted!



QuickSort Optimization: Cutoffs (1 of 2)

- ❖ For small n , recursion tends to cost more than a quadratic sort
 - Remember: asymptotic complexity applies to large n
 - ➔ ■ Recursive calls add overhead (which “isn’t worth it” for small n)
- ❖ Recursive calls for small n are the most common (“leaf calls”)
 - Calls for small n are the vast majority of the recursive calls!
 - Image below is for mergesort, but shows this well



QuickSort Optimization: Cutoffs (2 of 2)

→ So, switch algorithms for subproblems below a cutoff size

- Eg, Java 12 uses InsertionSort for primitive types when $n < 47$

```
void quickSort(int[] arr, int lo, int hi) {  
    if (hi - lo < CUTOFF)  
        insertionSort(arr, lo, hi);  
    else  
        ...  
}
```

- ❖ Switching algorithms after a cutoff is a common technique!
 - E.g. *parallel* algorithms switch to *sequential* after a certain cutoff
 - E.g. MergeSort also uses cutoffs to switch to InsertionSort
- ❖ Does not affect asymptotic complexity, just the constants

QuickSort vs MergeSort (1 of 2)

Different algorithms, same problem

❖ MergeSort:

■ Execution

- Does its work “on the way up”
 - i.e., in the merge, after the recursive call returns

- Uses its auxiliary space very effectively:
 - Works well on linked lists
 - Linear merges minimize disk accesses

■ Time: always $O(n \log n)$

❖ QuickSort:

■ Execution:

- Does its work “on the way down”
 - i.e., in the partition, before the recursive call
- Doesn't need auxiliary space

■ Runtime: $O(n \log n)$ in best and randomized cases 😊

- But $O(n^2)$ worst-case 😞

QuickSort vs MergeSort (2 of 2)

❖ Asymptotic Runtime:

- ➔ QuickSort is $O(n \log n)$ in best and randomized cases, but $O(n^2)$ worst-case
 - MergeSort is always $O(n \log n)$

❖ Constants Matter!

- QuickSort does fewer copies and more comparisons, so it depends on the relative cost of these two operations
- Typically, cost of copies is higher so QuickSort really is the “quickest”

Lecture Outline

- ❖ QuickSort
 - Comparison with MergeSort

- ❖ **Comparison Sort Lower Bound**

- ❖ Non-Comparison Sorts
 - BucketSort
 - RadixSort

- ❖ Sorting Wrapup

A Different View of Sorting

❖ Assume we have n elements, none are equal (ie, no duplicates)

• **$n!$ permutations** (possible orderings) of the elements. For $n=3$

$a < b < c$ $a < c < b$ $b < a < c$ $b < c < a$ $c < a < b$ $c < b < a$

❖ Assume an “OptimalSort”

■ Instead of describing how it **works**, we’ll describe what it **knows and when it knows it**

■ Starts “knowing nothing”; “anything is possible”

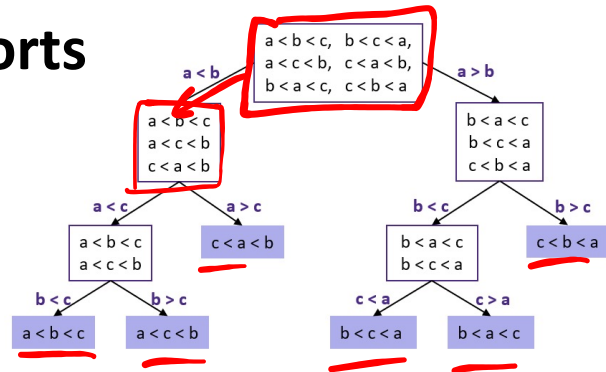
■ Each binary: $a < b$ or $b < a$ comparison gains information, eliminating some possibilities

• Each comparison eliminates (at most) half of remaining possibilities

■ In the end, narrows down to a single possibility

Representing Comparison Sorts

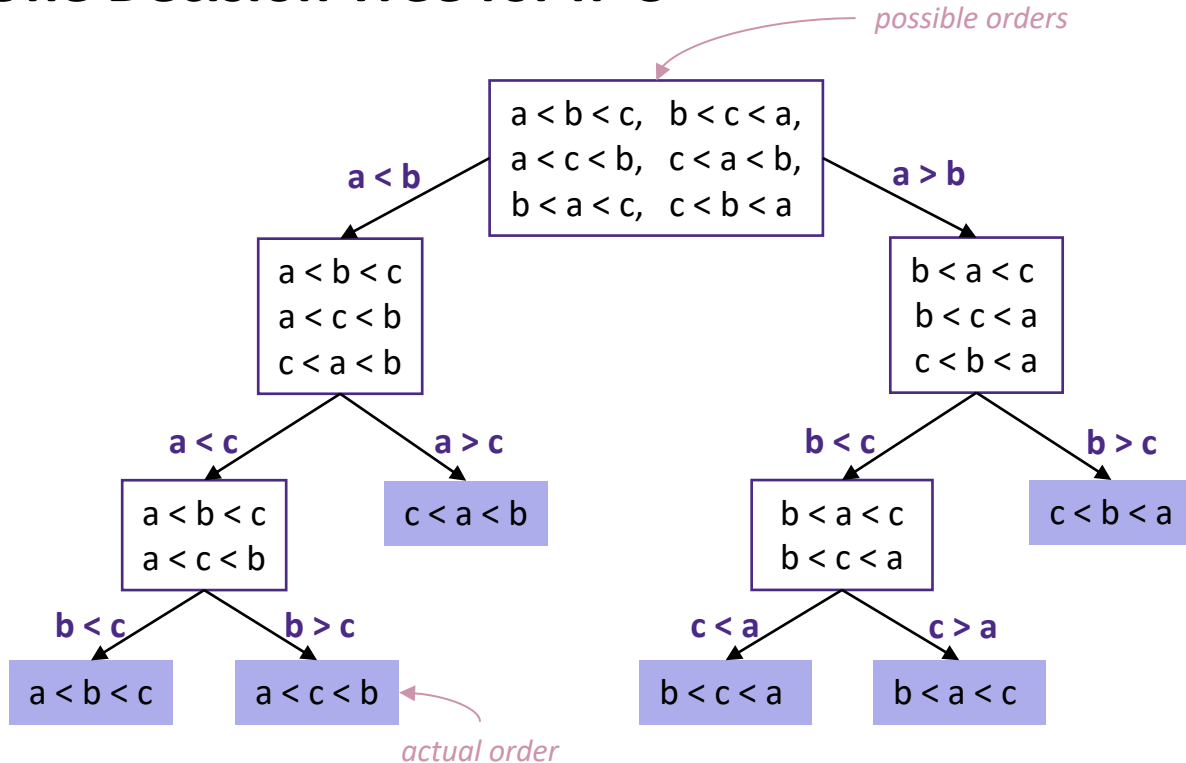
- ❖ Let's represent these binary comparisons as a binary tree!



- ❖ Called a *Decision Tree*
 - Nodes contain “set of remaining possible orderings”
 - The root contains all possible orderings; anything is possible
 - The leaves contain exactly one specific ordering
 - Edges are “answers from a comparison”

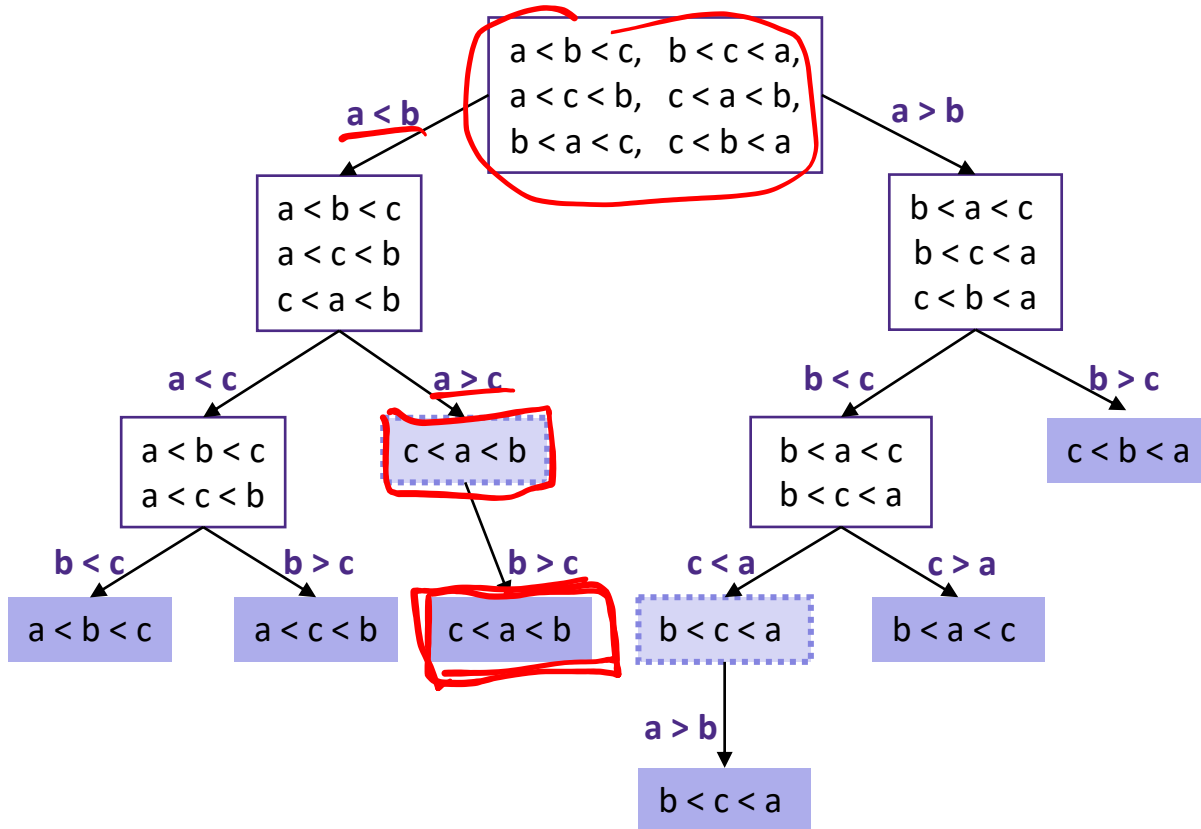
We are not actually building the tree; it's what our proof uses to represent “the most the algorithm could know so far”

One Decision Tree for $n=3$



- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree

Another Decision Tree for n=3



What the Decision Tree Tells Us

- ❖ Because any order is possible, any algorithm needs to ask enough questions to produce all $n!$ leaves (ie, orderings)
 - Each answer/ordering may lead to a different leaf
 - *So the binary tree must be big enough to have $n!$ leaves*

- ❖ Running *any* algorithm on *any* input will at best correspond to a root-to-leaf path in *some* decision tree with $n!$ leaves
 - Path length is the number of comparison operations needed
 - ~~*So no algorithm can have worst-case running time better than the height of a tree with $n!$ leaves*~~
 - Because the worst-case number-of-comparisons for an algorithm is an input that yields to a longest path in algorithm's decision tree

How did we get $n!$?

This is a 312 question. We have n variables and we wish to know how many different ways they can be ordered. In other words, want the number of permutations of size n . A permutation is the choice of r things from a set of n things without replacement and where the order matters:

$${}^n P_r = \frac{(n!)}{(n-r)!}$$

In our case, $r = n$ because we want the choice of all n things from the set of n things, so: ${}^n P_n = \frac{(n!)}{(n-n)!} = \frac{(n!)}{0!} = \frac{n!}{1} = n!$

Lower Bound on Height (1 of 2)

- ❖ A binary tree of height h has at most how many leaves?

$$L \leq \underline{2^h}$$

- ❖ A binary tree with L leaves has height at least:

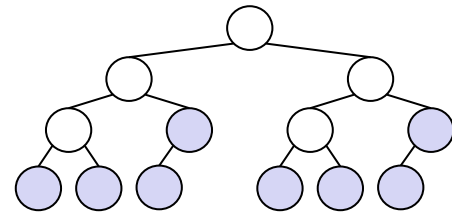
$$h \geq \underline{\log_2 L}$$

- ❖ The decision tree has how many leaves: $\underline{n!}$

- ❖ So the decision tree has height:

$$\cancel{h} \geq \underline{\log_2 n!}$$

Lower Bound on Height (2 of 2)



❖ The height of a binary tree with L leaves is at least $\log_2 L$.

❖ So the height of our decision tree, h :

$$h \geq \log_2 (n!)$$

$$= \log_2 (n \cdot (n-1) \cdot (n-2) \cdots (2) \cdot (1))$$

$$= \log_2 n + \log_2 (n-1) + \dots + \log_2 1$$

$$\geq \log_2 n + \log_2 (n-1) + \dots + \log_2 (n/2)$$

$$\geq (n/2) \log_2 (n/2)$$

$$= (n/2)(\log_2 n - \log_2 2)$$

$$= (1/2)n \log_2 n - (1/2)n$$

$$\in \Omega(n \log n)$$

property of binary trees

definition of factorial

property of logarithms

keep first $n/2$ terms

each of the $n/2$ terms is $\geq \log_2 (n/2)$

property of logarithms

arithmetic

So... No comparison based sort can be faster than $n \log n$.

Lecture Outline

- ❖ QuickSort
 - Comparison with MergeSort

- ❖ Comparison Sort Lower Bound

- ❖ **Non-Comparison Sorts**
 - **BucketSort**
 - RadixSort

- ❖ Sorting Wrapup

BucketSort (a.k.a. BinSort)

- ❖ 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 **bucket (a.k.a. bin)**
 - If data is only integers, can store *count* of how many times that bucket has been used
- ❖ Output result via linear pass through array of buckets

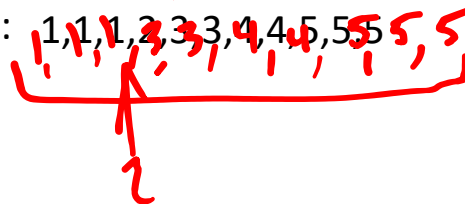
count array	
1	3
2	4
3	2
4	2
5	3

- 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 BucketSort

[1, 100000000]

- ❖ Overall $O(n+K)$
 - Linear in n , but also linear in K
 - $\Omega(n \log n)$ doesn't apply because **this is not a comparison sort**
- ❖ Good when range, K , is smaller (or not much larger) than n
 - We don't spend time doing lots of comparisons of duplicates!
- ❖ Bad when K is much larger than n
 - Wasted space; wasted time during final linear $O(K)$ pass

BucketSort with Data

- ❖ Most real lists aren't just #'s; we have data too
 - Make each bucket is a list (say, linked list)
 - To add to a bucket, place at end $O(1)$ (keep pointer to last element)

- Example: movie ratings (1=bad, ... 5=excellent)

- Input=

- 5: Pulp Fiction

- 3: Harry Potter movies

- 1: Star Wars I

- 5: Star Wars V

- Output= Star Wars I, Harry Potter movies, Pulp Fiction, Star Wars V

count array	
1	Star Wars I
2	
3	Harry Potter
4	
5	Pulp Fiction → Star Wars V

Lecture Outline

- ❖ QuickSort
 - Comparison with MergeSort
- ❖ Comparison Sort Lower Bound
- ❖ Non-Comparison Sorts
 - BucketSort
 - **RadixSort**
- ❖ Sorting Wrapup

RadixSort

- ❖ Radix = “the base of a number system”
 - Examples will use 10 because we are used to that
 - Implementations may 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, sort with Bucket Sort
 - Keeping sort stable
 - Do one pass per digit
- ❖ **Invariant:** After k passes, the last k digits are sorted
- ❖ Aside: Origins go back to the 1890 U.S. census

1's place

RadixSort: Example (1 of 6)

0	1	2	3	4	5	6	7	8	9

Input: 333
143
591
65
332
491

First pass:

1. BucketSort by ones digit
2. Iterate thru and collect into a list
 - List is sorted by first digit

RadixSort: Example (2 of 6)

0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				

Input:

333
143
591
65
332
491

First pass:

1. BucketSort by ones digit
2. Iterate thru and collect into a list
 - List is sorted by first digit

Order is now:

591
491
332
333
143
65

0	1	2	3	4	5	6	7	8	9
	5 <u>9</u> 1 491	332	333 143		<u>6</u> 5				

Input:

333
143
591
65
332
491

Second pass:

1. BucketSort by tens digit, stably

Order is now:



0	1	2	3	4	5	6	7	8	9
	591 491	332	333 143		65				
0	1	2	3	4	5	6	7	8	9
			<u>332</u> <u>333</u>	<u>143</u>		<u>65</u>			<u>591</u> <u>491</u>

Input: 333
143
591
65
332
491

Second pass:

1. BucketSort by tens digit, stably

Notice: if we chop off the 100's place, these are now sorted

Order is now:

332
333
143
65
591
491

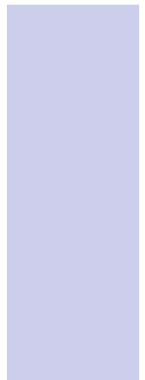
0	1	2	3	4	5	6	7	8	9
			332 333	143		65			591 491
0	1	2	3	4	5	6	7	8	9

Input: 333
143
591
65
332
491

Third pass:

1. BucketSort by hundreds digit, stably

Order is now:



0	1	2	3	4	5	6	7	8	9
			332 333	143		65			591 491
↓ 0	↓ 1		↓ 3						↓ 9
65	143		→ 332 → 333						491 591

Input:

333
143
591
65
332
491

Third pass:

1. BucketSort by hundreds digit, stably

🎉 *Only 3 digits; we're done!* 🎉

Order is now:

65
 143
 332
 333
 491
 591

Analysis of Radix Sort

❖ Performance depends on:

■ Input size: n

■ Number of buckets = Radix: K

- e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62

■ Number of passes = “Digits”: P

- e.g. Ages of people: 3; Phone #: 10; Person’s name: ?

❖ Work per pass is 1 BucketSort:

$$\underline{O(n+k)}$$

■ *Each pass is a BucketSort!*

❖ Total work is

$$\underline{O(P \cdot (n+k))} \rightarrow O(P \cdot n)$$

■ *We do ‘P’ passes, each of which is a BucketSort!*

Comparison to Comparison Sorts

- ❖ Compared to comparison sorts, radix sorts are sometimes a win, but often not

- ❖ Example: Strings of English letters up to length 15

- Approximate run-time: $15^*(52 + n)$

- This is less than $n \log n$ only if $n > 33,000$

- Of course, cross-over point depends on constant factors of the implementations plus P and B

- And radix sort can have poor locality properties

- ❖ Not really practical for many classes of keys

- Strings: Lots of buckets

cat
dot

Lecture Outline

- ❖ QuickSort
 - Comparison with MergeSort
- ❖ Comparison Sort Lower Bound
- ❖ Non-Comparison Sorts
 - BucketSort
 - RadixSort
- ❖ **Sorting Wrapup**

Features of Sorting Algorithms

❖ In-place

- Sorted items occupy the same space as the original items. (No copying required, only $O(1)$ extra space if any.)

❖ Stable

- Items in input with the same value end up in the same order as when they began.

1 "a", 1 "b"

Sorting: Summary (1 of 3)

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

- SelectionSort, InsertionSort:

→ The latter is linear for mostly-sorted!

- Good for “below a cut-off” to help divide-and-conquer sorts

→ “Fancy” $O(n \log n)$ sorts

- HeapSort: not parallelizable

- MergeSort: works as external sort

- QuickSort: $O(n^2)$ in worst-case; cost of comparisons/copies often makes it fastest

Sorting : Summary (2 of 3)

- ❖ $\Omega(n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- ❖ Non-comparison sorts
 - Bucket sort good for small number of key values
 - Radix sort uses fewer buckets and more phases
- ❖ Best way to sort? It depends!

Sorting: Summary (3 of 3)



	Best-Case	Worst-Case	Randomized Case	In-Place?	Stable?	Notes
InsertionSort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$	Yes	Yes	Fastest for small or partially-sorted input
SelectionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Yes	No	
In-Place HeapSort	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Yes	No	Slow in practice
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	No	Yes	Fastest stable sort
QuickSort <i>(1st-element pivot + 3-pass partition)</i>	$\Theta(N \log N)$	$\Theta(N^2)$	$\Theta(N \log N)$	No	Yes	$\geq 2x$ slower than MergeSort
QuickSort <i>(Median-of-three pivot + Hoare partition + cutoffs)</i>	$\Omega(N)$	$O(N^2)$	$\Theta(N \log N)$	Yes	No	Fastest comparison sort
BucketSort	$\Theta(N+K)$	$\Theta(N+K)$	$\Theta(N+K)$	No	Yes	
RadixSort	$\Theta(P(K+N))$	$\Theta(P(K+N))$	$\Theta(P(K+N))$	No	Yes	