

CSE 332

**JULY 21ST – SORTING & INTRO TO
PARALLELISM**

MIDTERM RECAP

- **Several students lost points because they were unable to finish**

MIDTERM RECAP

- **Several students lost points because they were unable to finish**
 - Want exams to be test of knowledge, not necessarily about test taking

MIDTERM RECAP

- **Several students lost points because they were unable to finish**
 - Want exams to be test of knowledge, not necessarily about test taking
 - By Saturday night, email me if you'd like to redo a problem.

MIDTERM RECAP

- **Several students lost points because they were unable to finish**
 - Want exams to be test of knowledge, not necessarily about test taking
 - By Saturday night, email me if you'd like to redo a problem.
 - On Monday, in the late afternoon, you'll get a replacement design decision question and you'll have 24 hours to complete a very thorough evaluation.

MIDTERM RECAP

- **Several students lost points because they were unable to finish**
 - For one token, you may replace an one exam question with the score you receive on the assignment

MIDTERM RECAP

- **Several students lost points because they were unable to finish**
 - For one token, you may replace an one exam question with the score you receive on the assignment

SORTING

- **Important definitions**
 - In-place: Requires only $O(1)$ extra memory
 - **usually means the array is mutated**
 - Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
 - Sorting by first name and then last name will give you **last then first** with a stable sort.
 - The most recent sort will always be the primary

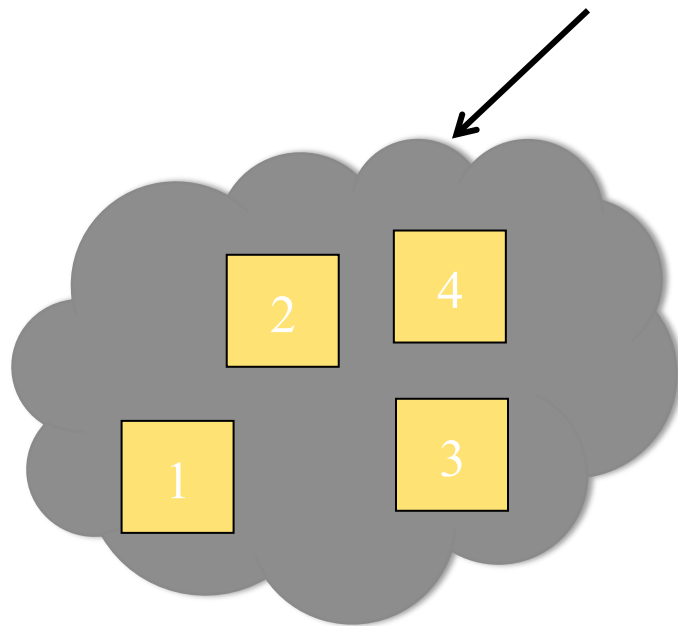
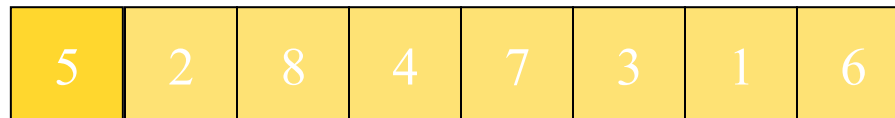
SORTING

- **Important definitions**

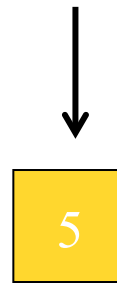
- **Interruptable:** the algorithm can run only until the first k elements are in sorted order
- **Comparison sort:** utilizes comparisons between elements to produce the final sorted order.
 - Bogo sort is not a comparison sort
 - Comparison sorts are $\Omega(n \log n)$, they cannot do better than this

QUICK SORT

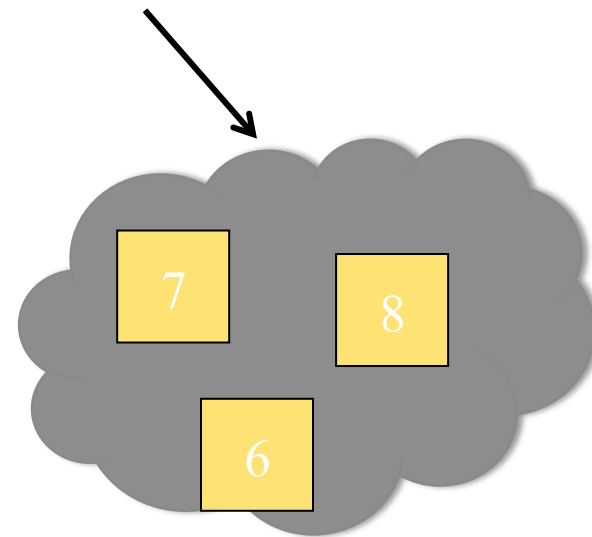
Divide: Split array around a 'pivot'



numbers \leq
pivot



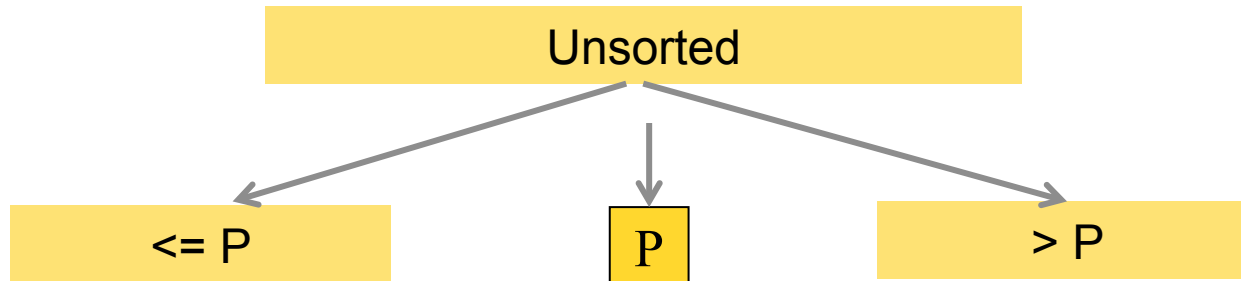
pivo
t



numbers $>$ pivot

QUICK SORT

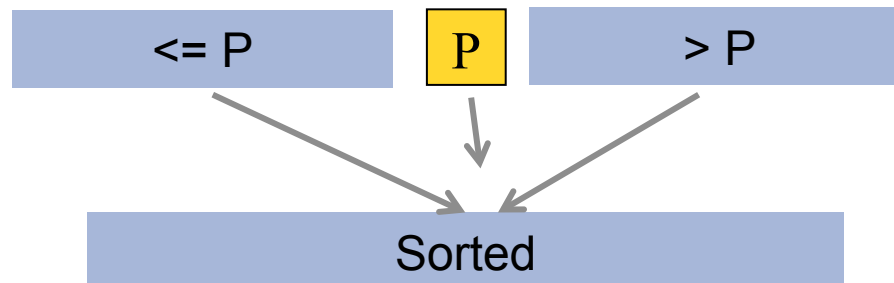
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot



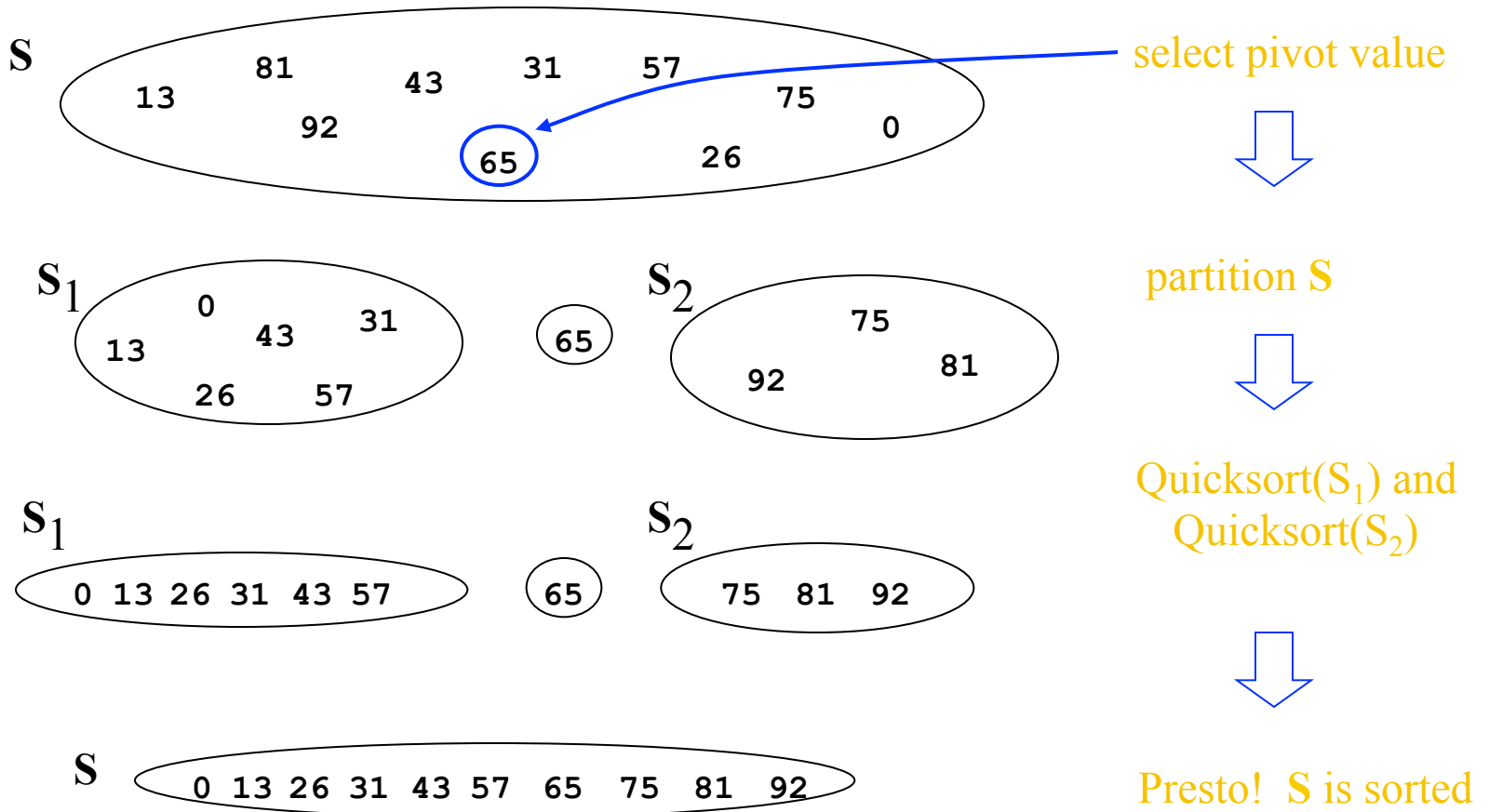
QUICK SORT

PSEUDOCODE

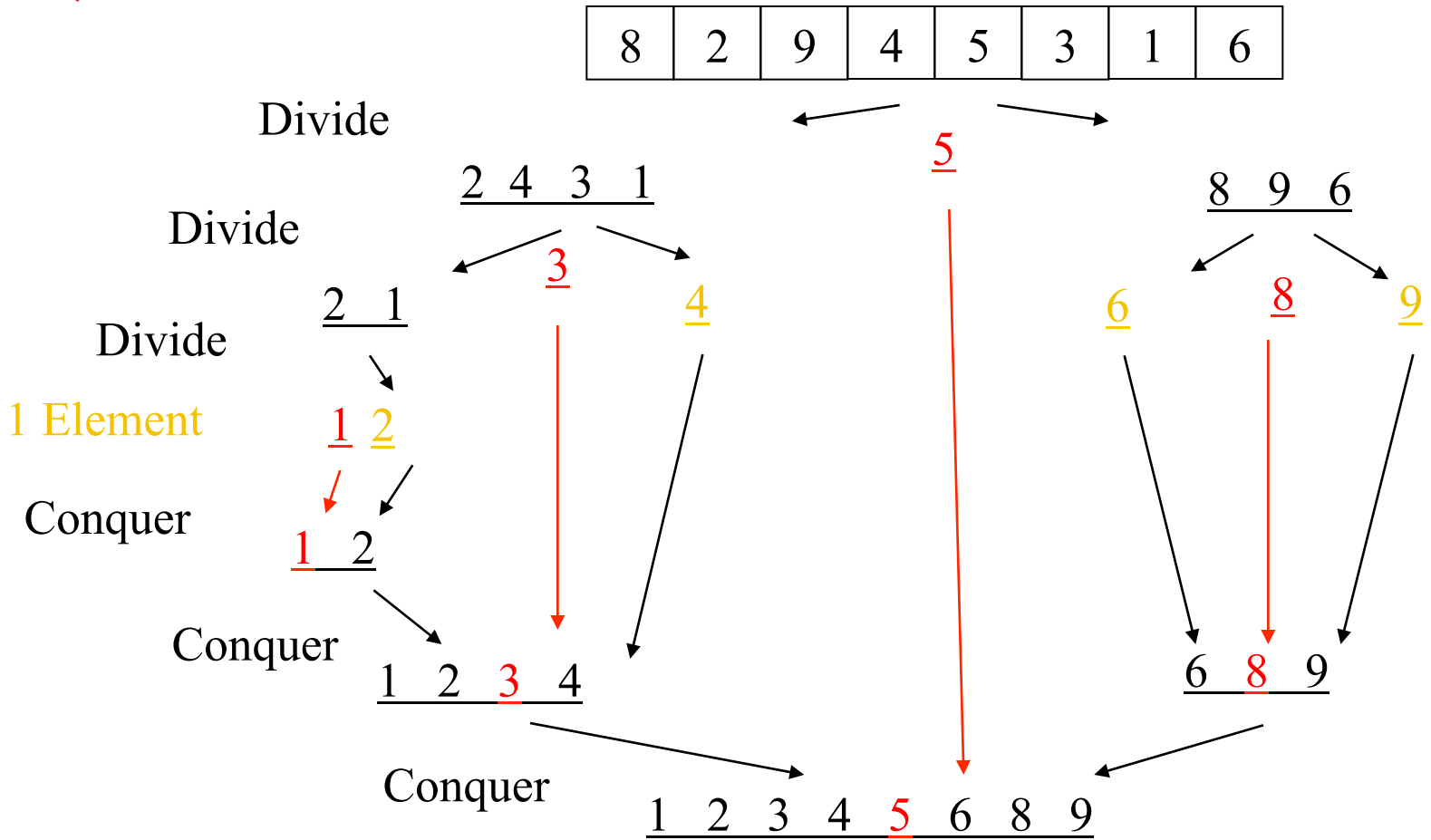
Core idea: Pick some item from the array and call it the pivot. Put all items smaller in the pivot into one group and all items larger in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

```
quicksort(input) {
  if (input.length < 2) {
    return input;
  } else {
    pivot = getPivot(input);
    smallerHalf = sort(getSmaller(pivot, input));
    largerHalf = sort(getBigger(pivot, input));
    return smallerHalf + pivot + largerHalf;
  }
}
```

QUICKSORT



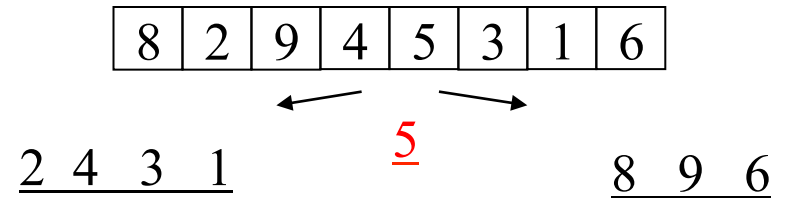
QUICKSORT



PIVOTS

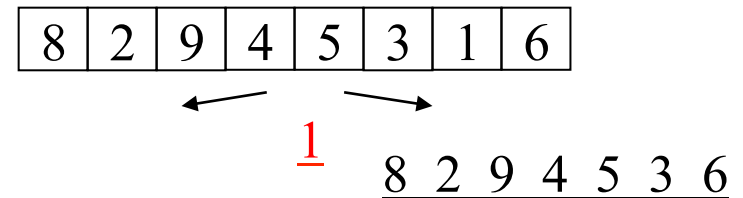
Best pivot?

- Median
- Halve each time



Worst pivot?

- Greatest/least element
- Problem of size $n - 1$
- $O(n^2)$



POTENTIAL PIVOT RULES

While sorting `arr` from `lo` (inclusive) to `hi` (**exclusive**)...

Pick `arr[lo]` or `arr[hi-1]`

- Fast, but worst-case occurs with mostly sorted input

Pick random element in the range

- Does as well as any technique, but (pseudo)random number generation can be slow
- Still probably the most elegant approach

Median of 3, e.g., `arr[lo]`, `arr[hi-1]`, `arr[(hi+lo)/2]`

- Common heuristic that tends to work well

PARTITIONING

Conceptually simple, but hardest part to code up correctly

- After picking pivot, need to partition in linear time in place

One approach (there are slightly fancier ones):

1. Swap pivot with `arr[lo]`
2. Use two counters `i` and `j`, starting at `lo+1` and `hi-1`
3. `while (i < j)`
 - `if (arr[j] > pivot) j--`
 - `else if (arr[i] < pivot) i++`
 - `else swap arr[i] with arr[j]`
4. Swap pivot with `arr[i]` *

***skip step 4 if pivot ends up being least element**

EXAMPLE

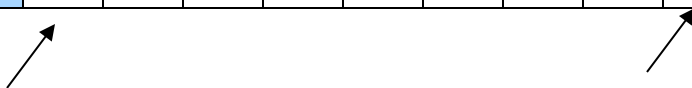
Step one: pick pivot as median of 3

- $l_o = 0, h_i = 10$

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

- Step two: move pivot to the l_o position

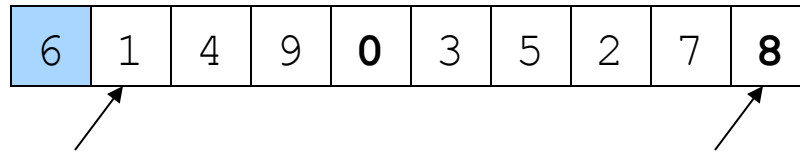
0	1	2	3	4	5	6	7	8	9
6	1	4	9	0	3	5	2	7	8



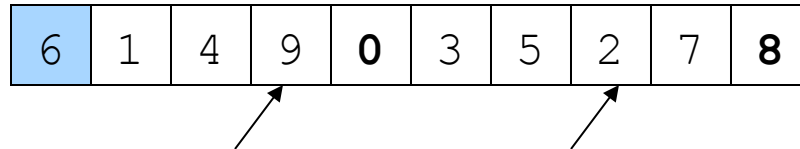
EXAMPLE

Often have more than one swap during partition – this is a short example

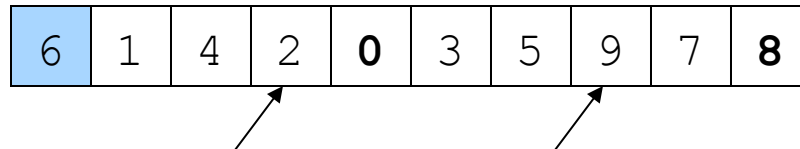
Now partition in place



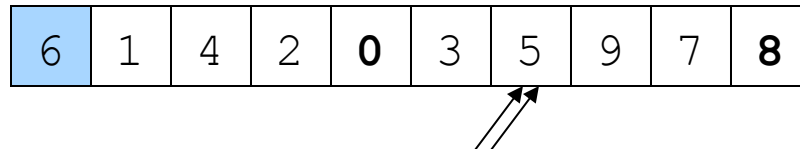
Move cursors



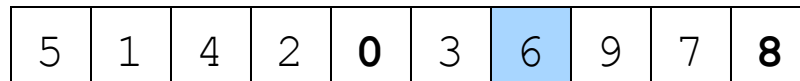
Swap



Move cursors

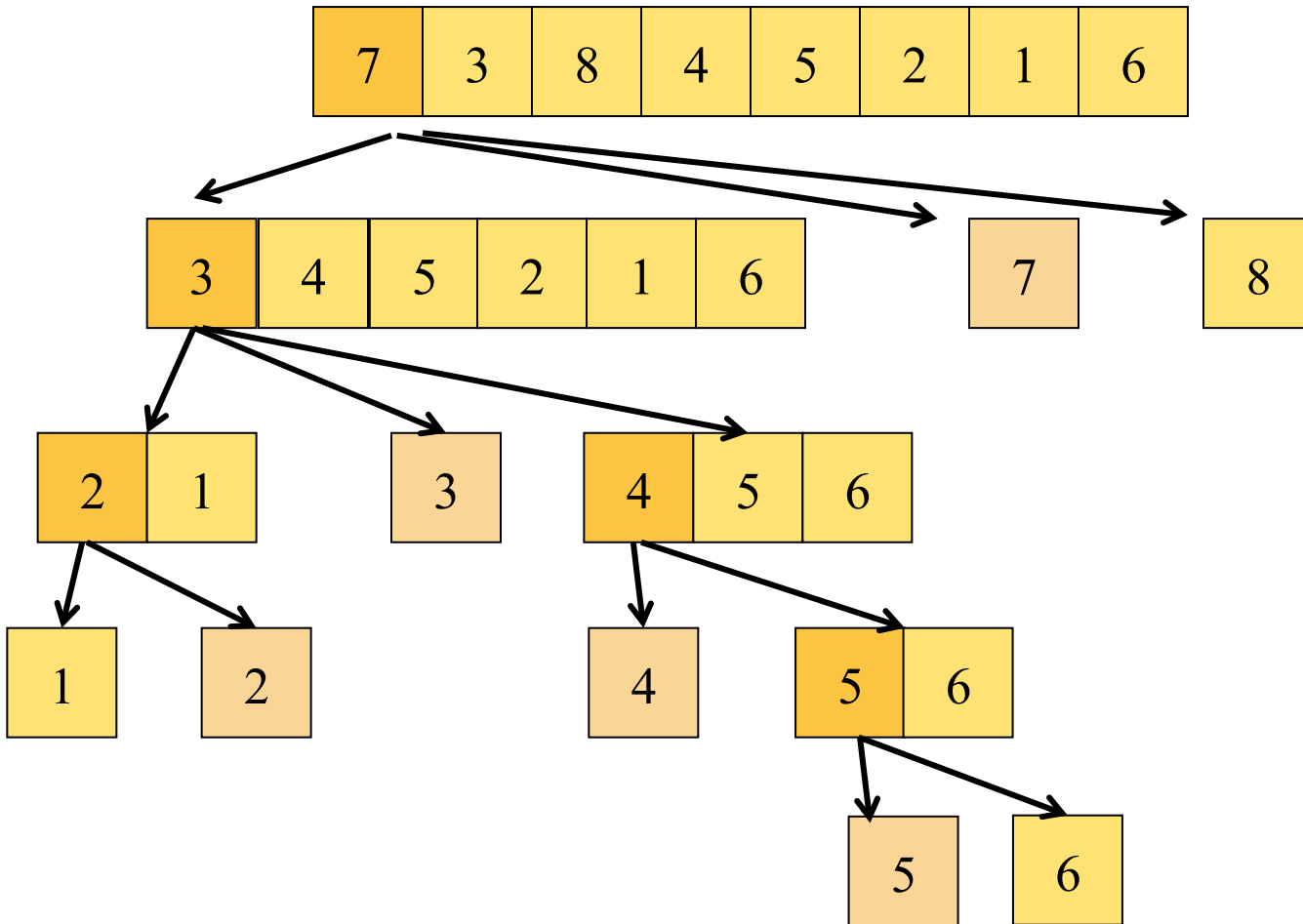


Move pivot



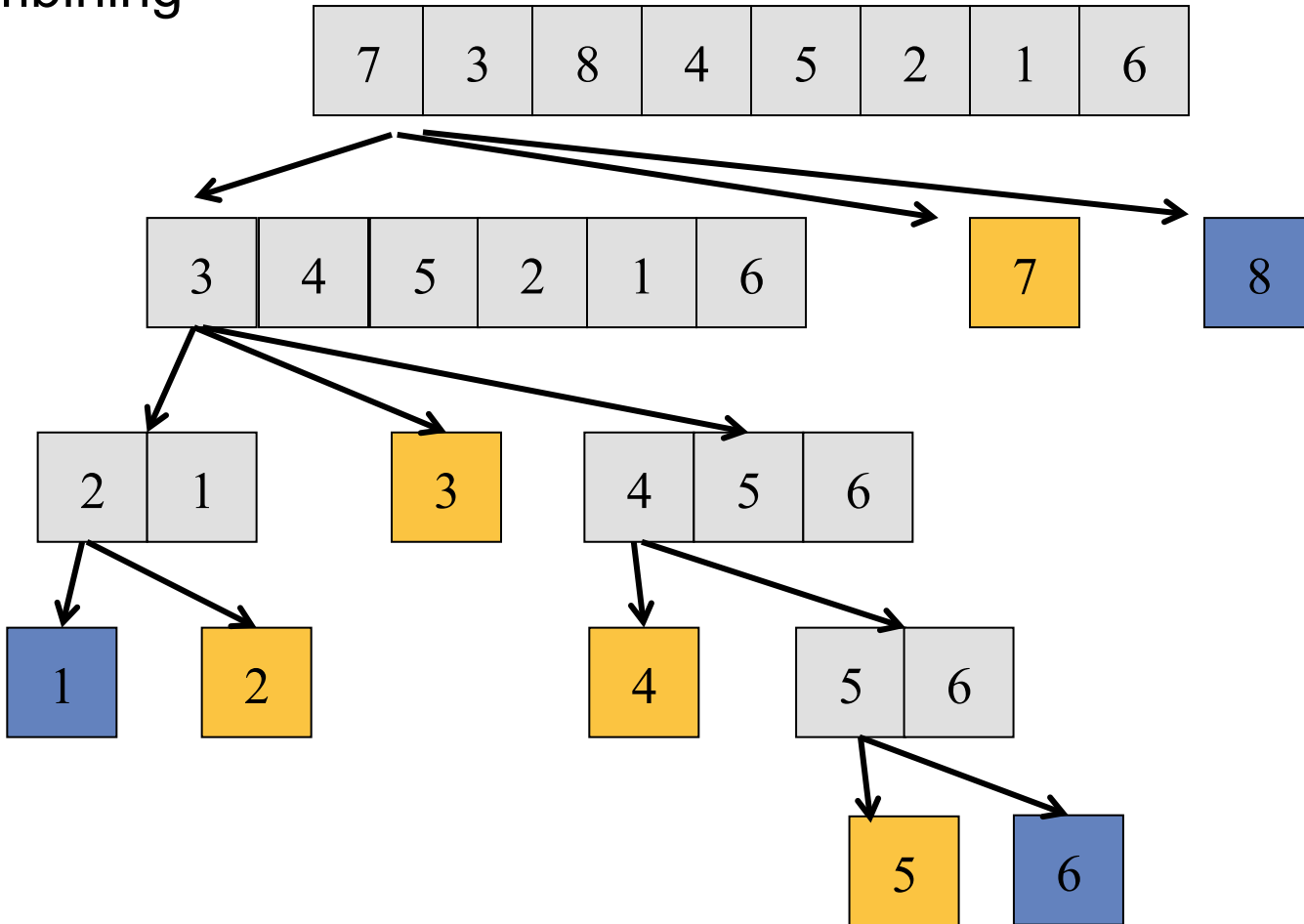
QUICK SORT EXAMPLE: DIVIDE

Pivot rule: pick the element at index 0



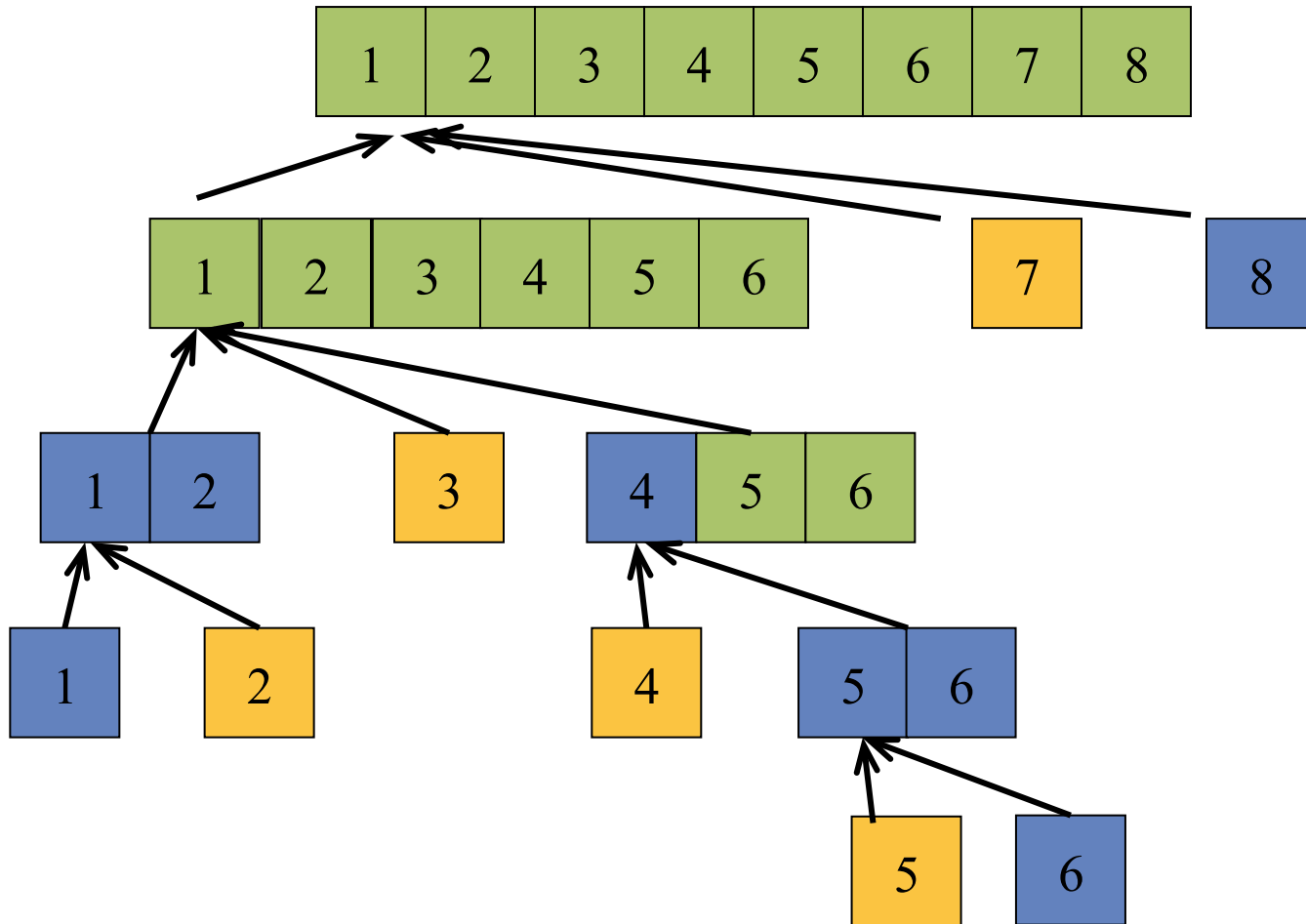
QUICK SORT EXAMPLE: COMBINE

Combine: this is the order of the elements we'll care about when combining



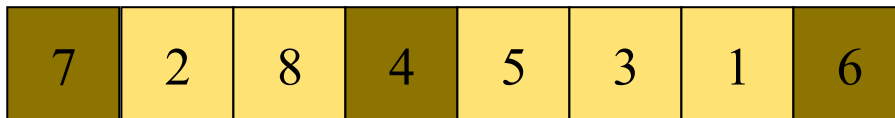
QUICK SORT EXAMPLE: COMBINE

Combine: put left partition < pivot < right partition



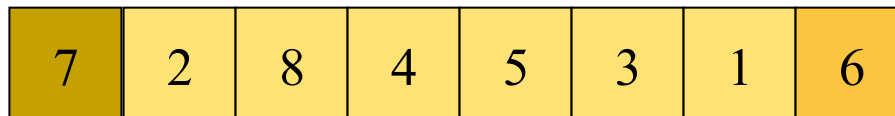
MEDIAN PIVOT EXAMPLE

Pick the median of first, middle, and last

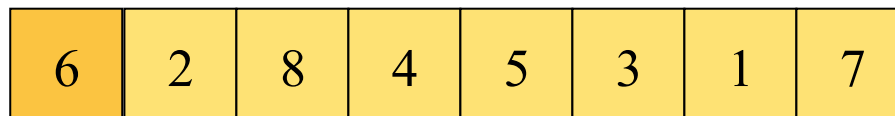


Median = 6

Swap the median with the first value



Pivot is now at index 0, and we're ready to go



PARTITIONING

Conceptually simple, but hardest part to code up correctly

- After picking pivot, need to partition in linear time in place

One approach (there are slightly fancier ones):

1. Put pivot in index `lo`
2. Use two pointers `i` and `j`, starting at `lo+1` and `hi-1`
3. `while (i < j)`
 - `if (arr[j] > pivot) j--`
 - `else if (arr[i] < pivot) i++`
 - `else swap arr[i] with arr[j]`
4. Swap pivot with `arr[i]` *

***skip step 4 if pivot ends up being least element**

EXAMPLE

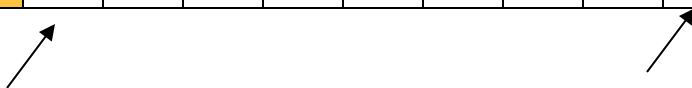
Step one: pick pivot as median of 3

- $l_o = 0, h_i = 10$

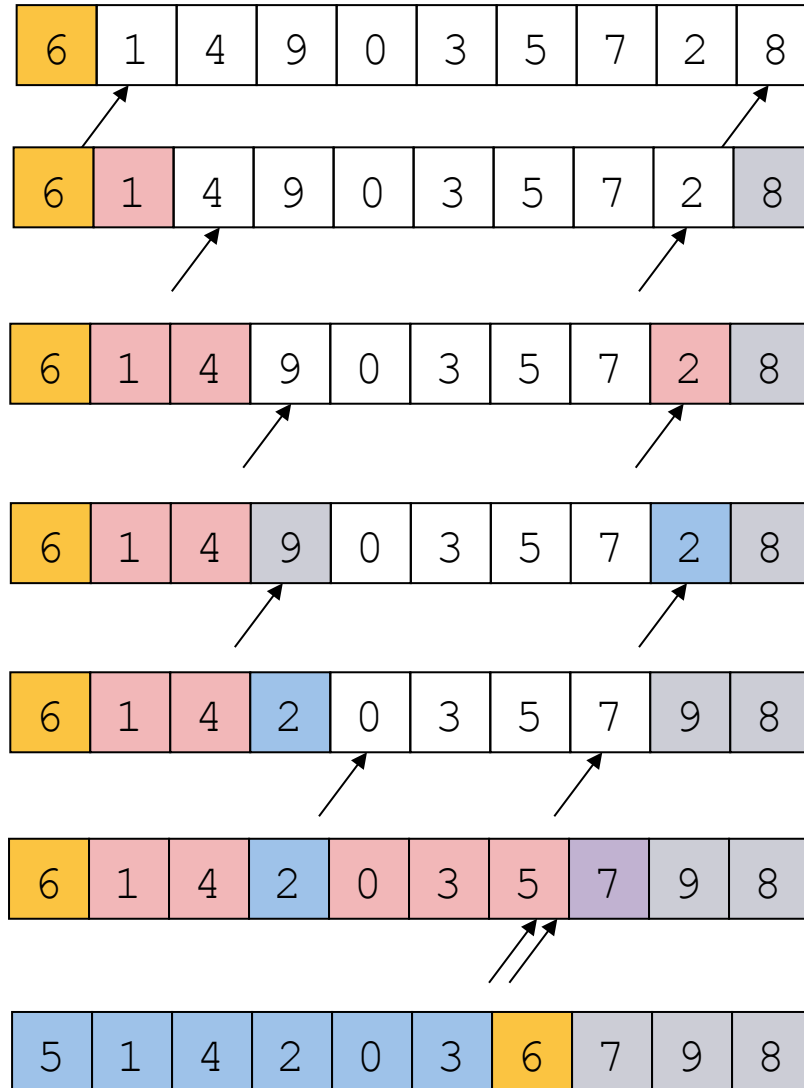
0	1	2	3	4	5	6	7	8	9
8	1	4	9	0	3	5	2	7	6

- Step two: move pivot to the l_o position

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



QUICK SORT PARTITION EXAMPLE



CUTOFFS

For small n , all that recursion tends to cost more than doing a quadratic sort

- Remember asymptotic complexity is for large n

Common engineering technique: switch algorithm below a cutoff

- Reasonable rule of thumb: use insertion sort for $n < 10$

Notes:

- Could also use a cutoff for merge sort
- Cutoffs are also the norm with parallel algorithms
 - Switch to sequential algorithm
- None of this affects asymptotic complexity

QUICK SORT ANALYSIS

Best-case: Pivot is always the median

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

$$T(n)=2T(n/2) + n \quad \text{-- linear-time partition}$$

Same recurrence as mergesort: $O(n \log n)$

Worst-case: Pivot is always smallest or largest element

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

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

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

Average-case (e.g., with random pivot)

- $O(n \log n)$, not responsible for proof

HOW FAST CAN WE SORT?

Heapsort & mergesort have $O(n \log n)$ worst-case running time

Quicksort has $O(n \log n)$ average-case running time

- **Assuming our comparison model:** The only operation an algorithm can perform on data items is a 2-element comparison. There is no lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$

COUNTING COMPARISONS

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

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

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*

COUNTING COMPARISONS

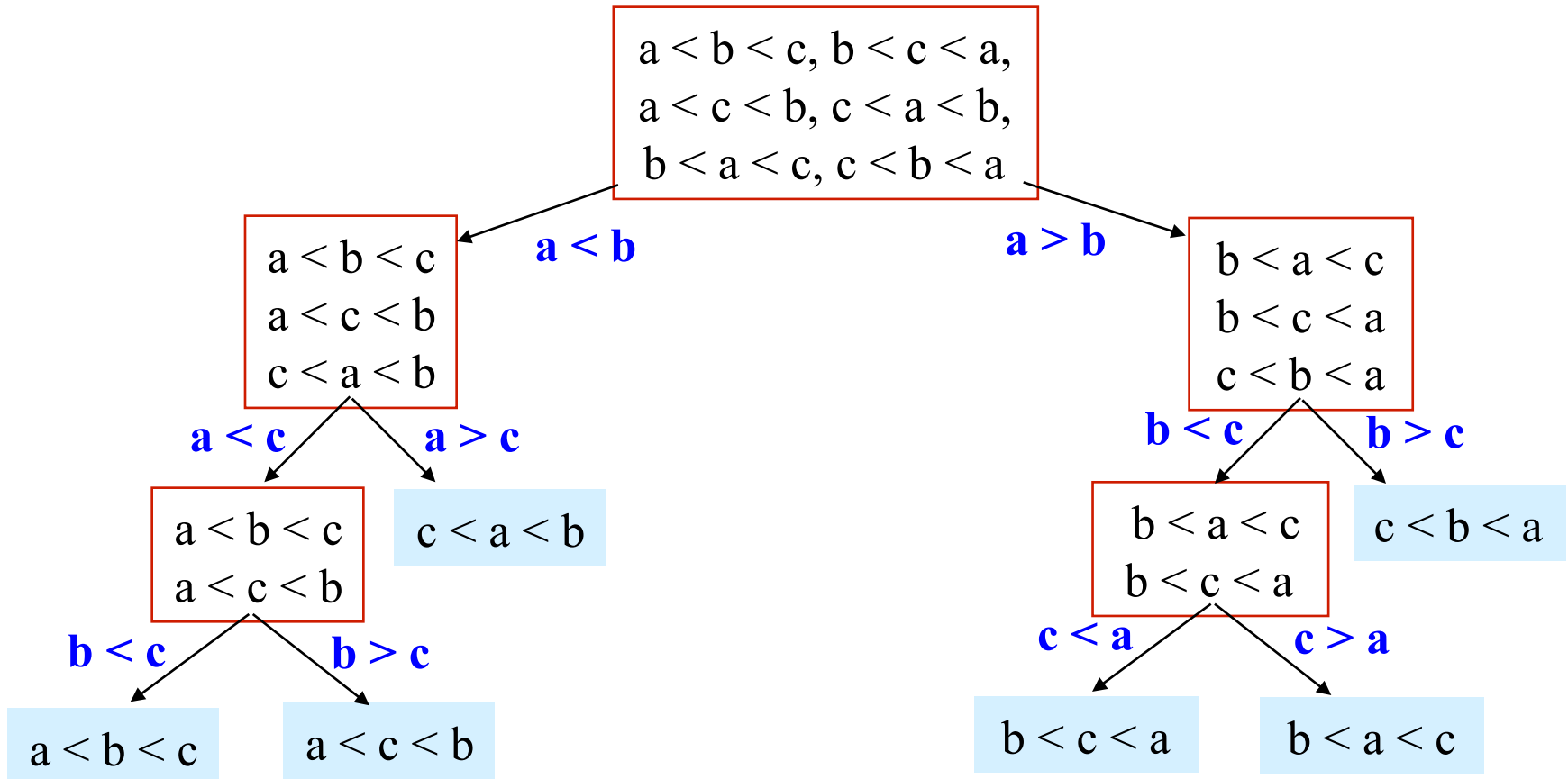
No matter what the algorithm is, it cannot make progress without doing comparisons

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

Can represent this process as a *decision tree*

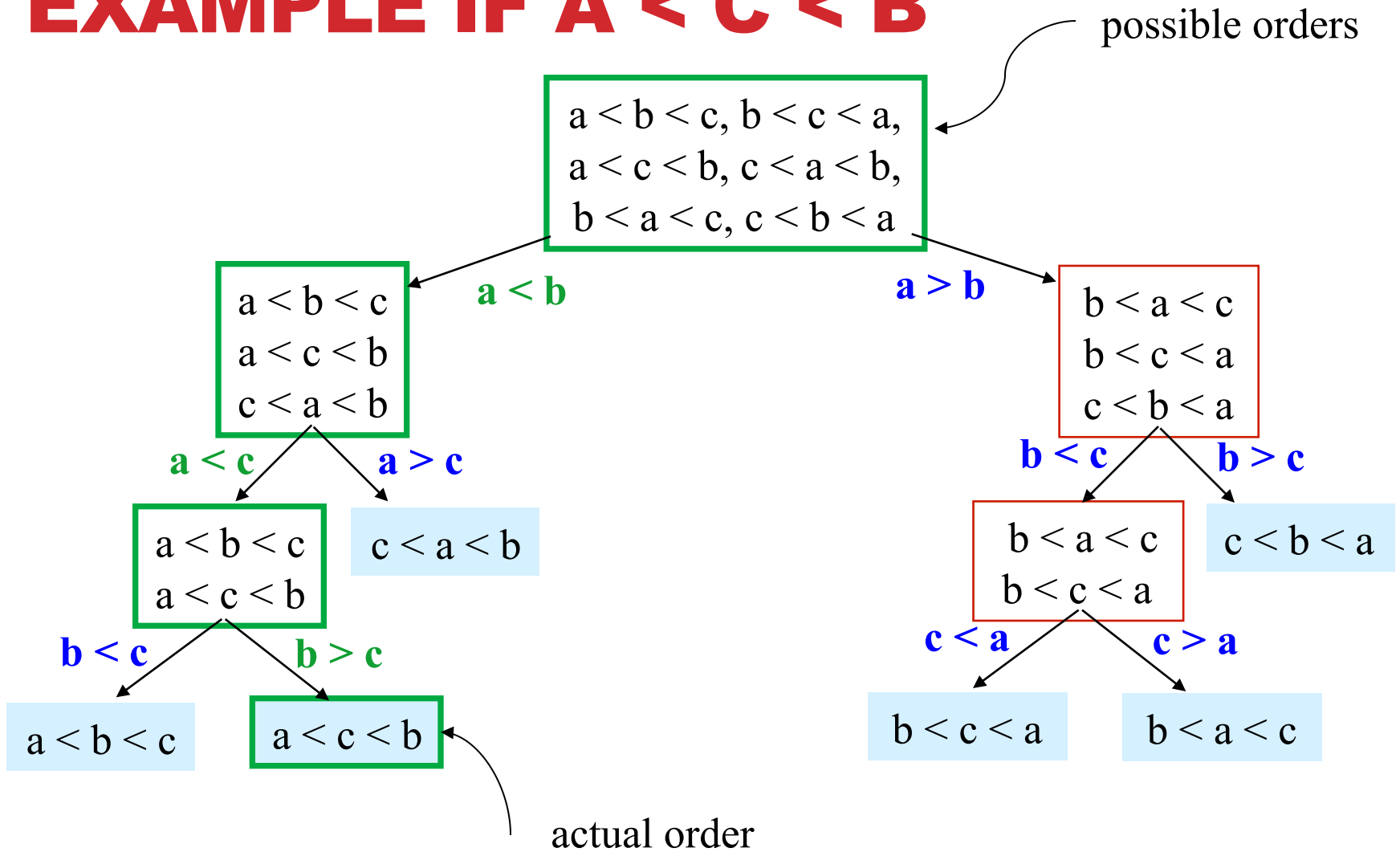
- Nodes contain “set of remaining possibilities”
- Edges are “answers from a comparison”
- The algorithm does not actually build the tree; it’s what our *proof* uses to represent “the most the algorithm could know so far” as the algorithm progresses

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 $\langle \text{num possible orderings} \rangle$ 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, $n=3$

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

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

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

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

In general, n choices for least element, $n-1$ for next, $n-2$ for next, ...

- $n(n-1)(n-2)\dots(2)(1) = n!$ 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 $\Omega(\lg(n!))$. Can we write that more clearly?

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

**Nice! Any sorting algorithm must do *at best* $(1/2)(n \lg n - n)$ comparisons:
 $\Omega(n \lg n)$**

SORTING

- **This is the lower bound for comparison sorts**

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**
 - They need to know something about the data

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**
 - They need to know something about the data
- **Strings and Ints are very well ordered**

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**
 - They need to know something about the data
- **Strings and Ints are very well ordered**
 - If I told you to put “Apple” into a list of words, where would you put it?

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

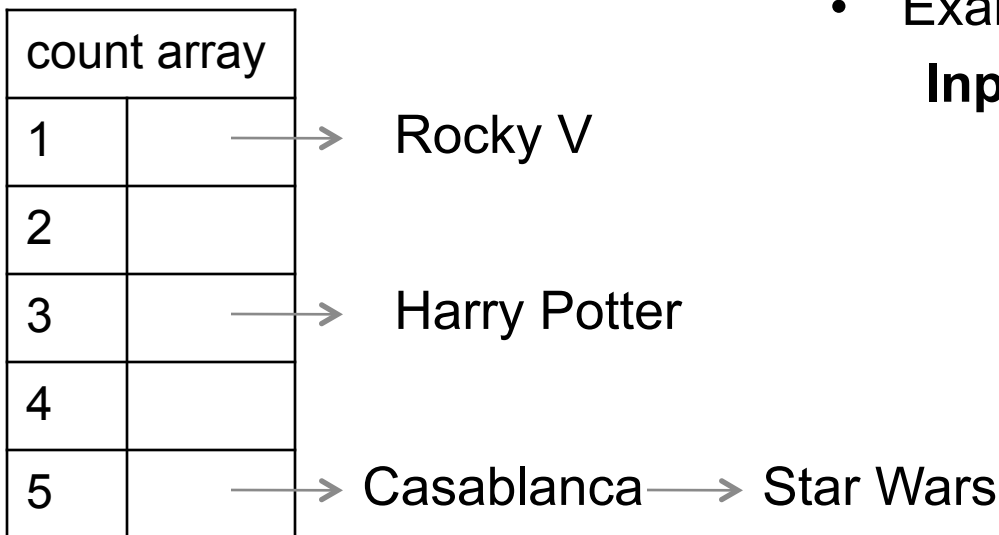
For data in addition to integer keys, use list at each bucket

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:

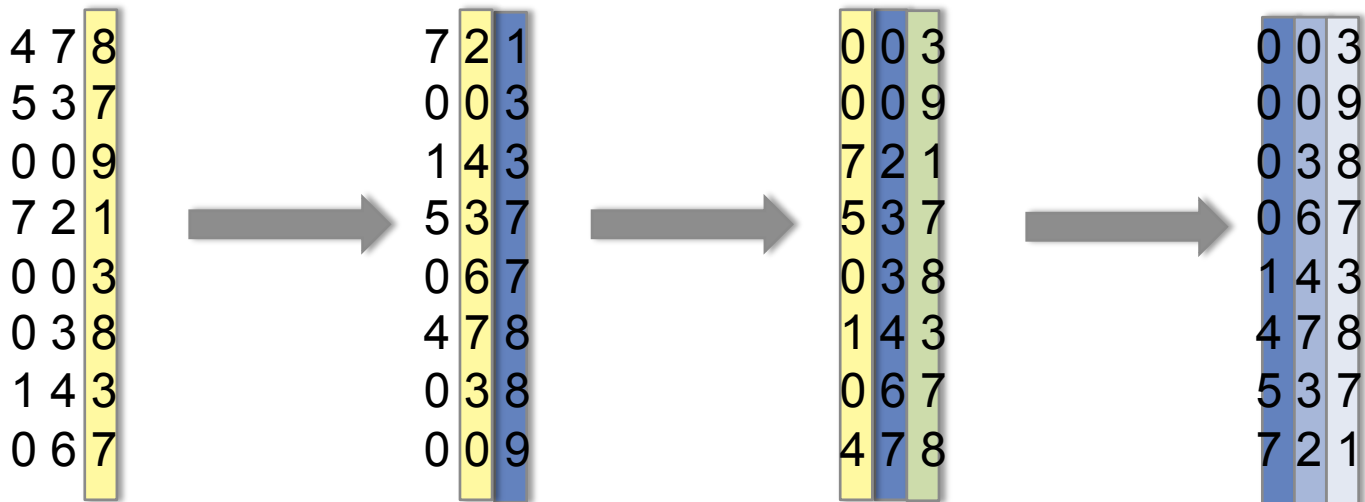
- Bucket sort on one digit at a time
 - Number of buckets = radix
 - Starting with *least* significant digit
 - Keeping sort *stable*
- Do one pass per digit
- **Invariant:** After k passes (digits), the last k digits are sorted

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

$O(n \log n)$ 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

$\Omega(n \log n)$ 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!

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!