# CS4102 Algorithms

Spring 2022

## Last Day of Unit A!

# Announcements

- Homework schedule on course website
  - Unit A Basic HW3 now available
  - Unit A Advanced and Programming HW now available
  - Unit A Programming submission opens Wednesday
  - **Hard deadline for Unit A Basic HW:  Friday, 11:30 pm on GradeScope**
- TA Office Hours
  - 7-10pm Sun-Thurs in Ols 011
  - Online hours also available
- Unit A Exam: Tuesday, February 22, in class

# Today's Keywords

- Finish up Median of Medians
  - how that applies to QuickSelect and Quicksort
- Quicksort and expected time
- Lower bounds proof for comparison sorts
- Review of sorts and their properties

# How to pick the pivot?

# Good Pivot

- What makes a good Pivot?
  - Roughly even split between left and right
  - Ideally: median
- Can we find median in linear time?
  - Yes!
  - Quickselect

# Quickselect

- Finds $i^{\text{th}}$ order statistic
  - $i^{\text{th}}$ smallest element in the list
  - $1^{\text{st}}$ order statistic: minimum
  - $n^{\text{th}}$ order statistic: maximum
  - $\frac{n}{2}^{\text{th}}$ order statistic: median
- CLRS, Section 9.1
  - **Selection problem**: Given a list of distinct numbers and value *i*, find value *x* in list that is larger than exactly *i-1* list elements

# Quickselect

- Finds $i^{\text{th}}$ order statistic
- Idea: pick a pivot element, partition, then recurse on sublist containing index $i$
- Divide: select an element $p$, Partition($p$)
- Conquer: if $i =$ index of $p$, done!
  - if $i <$ index of $p$ recurse left. Else recurse right
- Combine: Nothing!

# Partition (Divide step)

Given: a list, a pivot value $p$

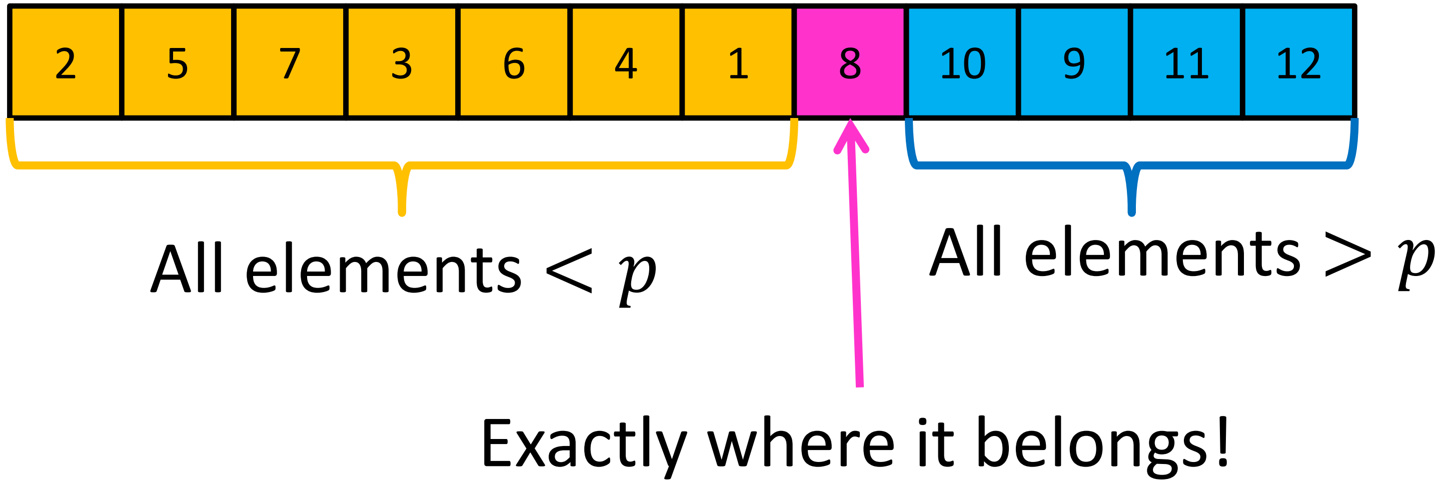
Start: unordered list

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 5 | 7 | 3 | 12 | 10 | 1 | 2 | 4 | 9 | 6 | 11 |

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

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 3 | 1 | 2 | 4 | 6 | 8 | 12 | 10 | 9 | 11 |

# Conquer




Recurse on sublist that contains index $i$

(adjust $i$ accordingly if recursing right)

# CLRS Pseudocode for Quickselect

A – the list
p – index of first item
r – index of last item
i – find *i*th smallest item
q – pivot location
k – number on left + 1

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q − p + 1        // number of elements in left sub-list + 1
if i == k            // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

note adjustment to *i* parameter when recursing on right side

Note: In CLRS, they're using a partition that randomly chooses the pivot element. That's why you see "Randomized" in the names here. Ignore that for the moment.

# Work These Examples!

- For each of the following calls, show
  - The value of *q* after each partition,
  - Which recursive calls made

1. Select( [3, 2, 9, 0, 7, 5, 6, 1], p=0, r=7, i=2)
2. Select( [3, 2, 9, 0, 7, 5, 6, 1], p=0, r=7, i=5)
3. Select( [3, 2, 9, 0, 7, 5, 6, 1], p=0, r=7, i=7)

# Quickselect Run Time

If the pivot is always the median:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 5 | 1 | 3 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 5 | 6 | 4 | 7 | 8 | 9 | 10 | 11 | 12 |

Then we divide in half each time

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

$$S(n) = O(n)$$

# Quickselect Run Time

If the partition is always unbalanced:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 2 | 3 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

Then we shorten by 1 each time

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

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

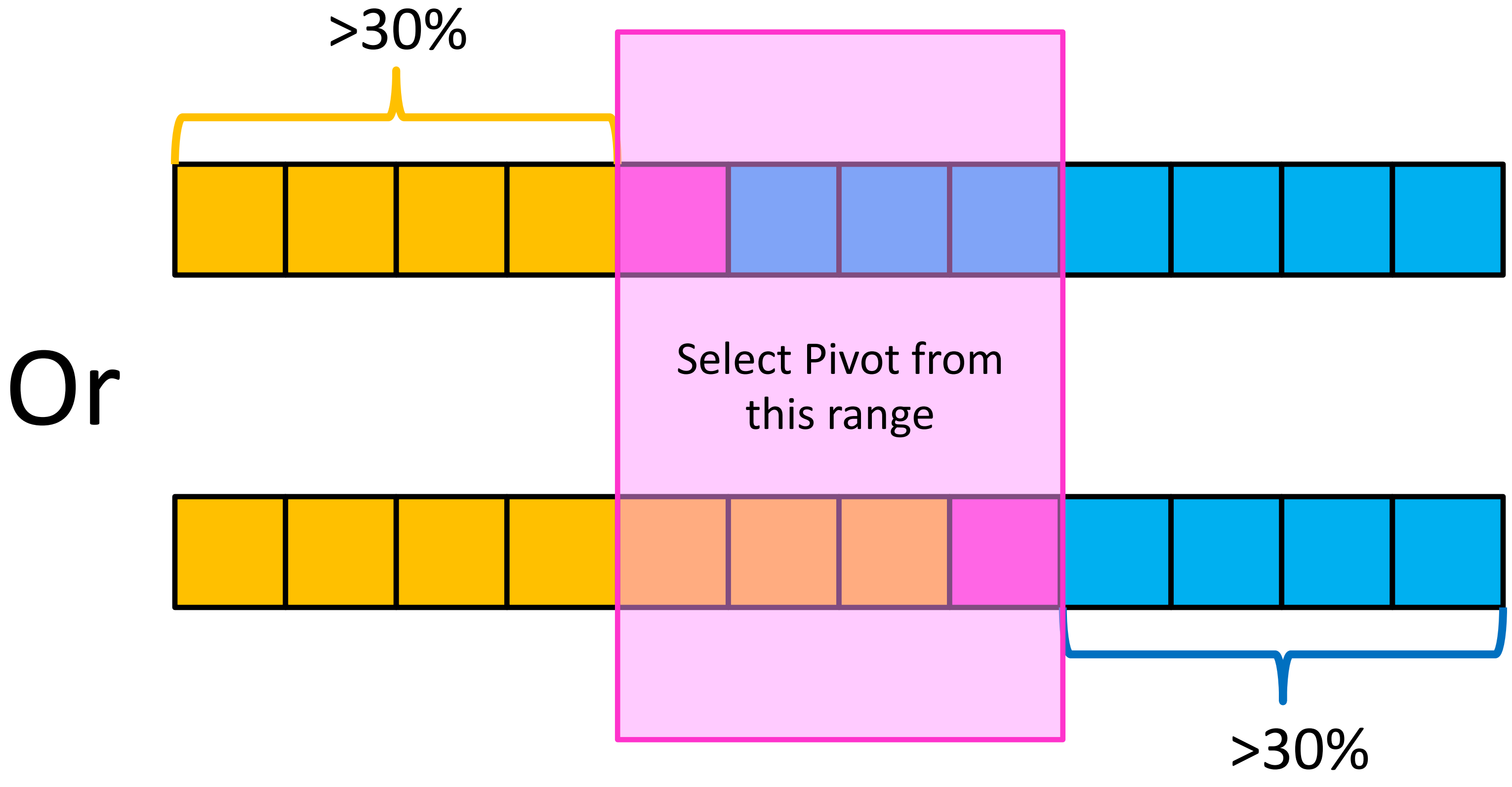
# Good Pivot for Quickselect

Déjà vu?

- What makes a good Pivot for Quickselect?
  - Roughly even split between left and right
  - Ideally: median
- Here's what's next:
  - First, **median of medians** algorithm
    - Finds something close to the median in $\Theta(n)$ time
  - Second, we can prove that when its result used with Quickselect's partition, then Quickselect is guaranteed $\Theta(n)$
    - Because we now have a $\Theta(n)$ way to find the median, this guarantees Quicksort will be $\Theta(n \lg n)$
  - Notes:
    - We have to do all this for every call to Partition in Quicksort
    - We could just use the value returned by median of medians for Quicksort's Partition

# Pretty Good Pivot

- What makes a "pretty good" Pivot?
  - Both sides of Pivot >30%



# Median of Medians

- Fast way to select a “pretty good” pivot
- Guarantees pivot is greater than 30% of elements and less than 30% of the elements
  - I.e. it’s in the middle 40% (±20% of the true median)
- Idea: break list into chunks, find the median of each chunk, use the median of those medians

- CLRS, pp. 220-221
- https://en.wikipedia.org/wiki/Median_of_medians

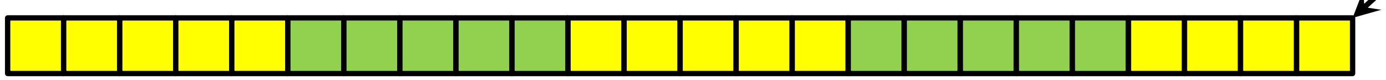
# Median of Medians

- Fast way to select a “good” pivot
- Guarantees pivot is greater than 30% of elements and less than 30% of the elements
- Idea: break list into chunks, find the median of each chunk, use the median of those medians

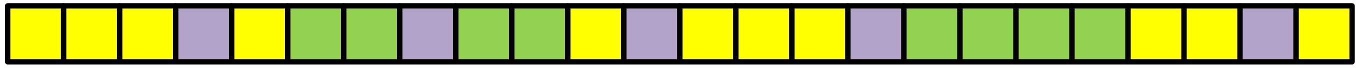
# Median of Medians

1. Break list into chunks of size 5

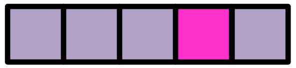
List could be long, many more than 5 chunks!



2. Find the median of each chunk
(using insertion sort: n=5, max 20 comparisons per chunk)

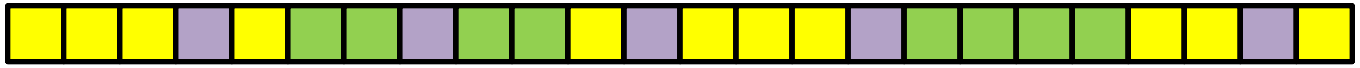


3. Return median of medians (using Quickselect, this algorithm, called recursively, on list of medians)



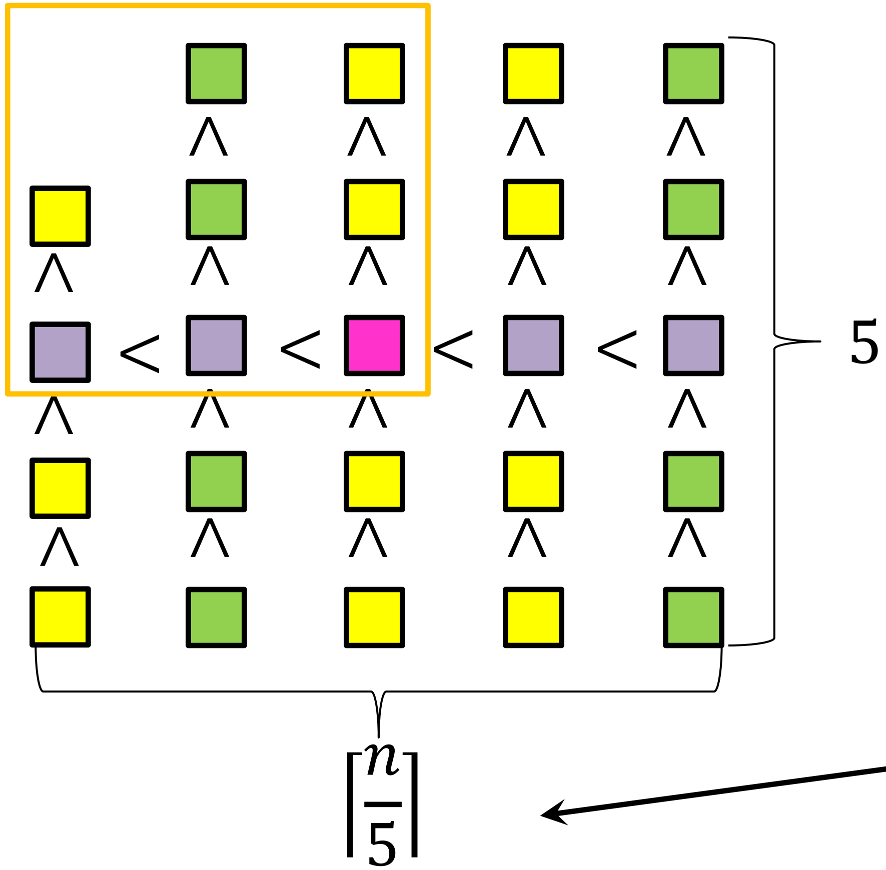
List could be long, many more than 5 medians!

# Why is this good?



Each chunk sorted, chunks ordered by their medians

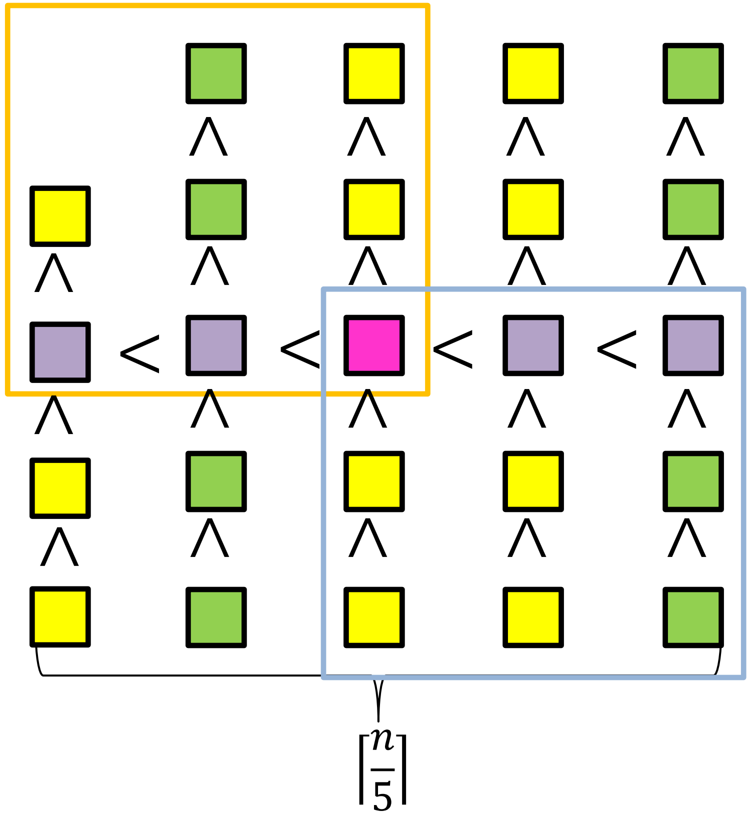
MedianofMedians
is Greater than all
of these




List could be long, so not a small number!

# Why is this good?

MedianofMedians is larger than all of these




Worried about the details of this math? See CLRS p. 221

Larger than 3 things in each (but one) list to the left

$$3\left(\frac{1}{2}\cdot\left\lceil\frac{n}{5}\right\rceil - 2\right) \approx \frac{3n}{10} - 6 \text{ elements } <$$

Similarly:

$$3\left(\frac{1}{2}\cdot\left\lceil\frac{n}{5}\right\rceil - 2\right) \approx \frac{3n}{10} - 6 \text{ elements } >$$

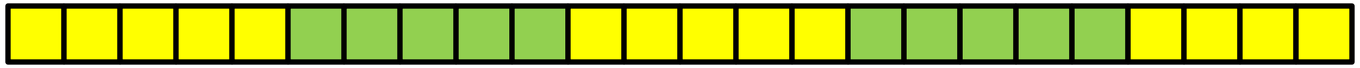
# Run-time of Quickselect with Median of Medians

- Divide: select an element $p$ using Median of Medians, Partition($p$) $M(n) + \Theta(n)$
- Conquer: if $i =$ index of $p$, done, if $i <$ index of $p$ recurse left. Else recurse right $\leq S\left(\frac{7}{10}n\right)$
- Combine: Nothing!

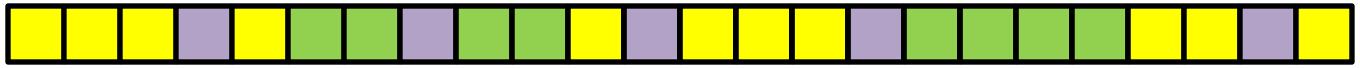
$$S(n) \leq S\left(\frac{7}{10}n\right) + M(n) + \Theta(n)$$

# Median of Medians, Run Time

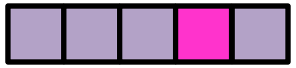
1. Break list into chunks of 5 $\Theta(n)$



2. Find the median of each chunk $\Theta(n)$



3. Return median of medians (using Quickselect)



$S\left(\frac{n}{5}\right)$

$$M(n) = S\left(\frac{n}{5}\right) + \Theta(n)$$

# Quickselect

$$S(n) \le S\left(\frac{7n}{10}\right) + M(n) + \Theta(n)$$

$$M(n) = S\left(\frac{n}{5}\right) + \Theta(n)$$

$$= S\left(\frac{7n}{10}\right) + S\left(\frac{n}{5}\right) + \Theta(n)$$

$$= S\left(\frac{7n}{10}\right) + S\left(\frac{2n}{10}\right) + \Theta(n)$$

$$\le S\left(\frac{9n}{10}\right) + \Theta(n)$$ Because $S(n) = \Omega(n)$

CLRS gives a more rigorous proof!
See p. 222 for more details

Master theorem Case 3!

$$S(n) = O(n)$$

$$S(n) = \Theta(n)$$

# Compare to 'Obvious' Approach

- An "obvious" approach to Selection Problem:
  - Given list and value *i*: Sort list, then choose *i*-th item
  - We've only seen sorting algorithms that are $\Omega(n \log n)$
  - Later we'll show this really is a lower-bound
  - So this approach is $\Theta(n \log n)$
- Therefore Quickselect is asymptotically better than this sorting-based solution for Selection Problem!

# Phew! Back to Quicksort

Using Quickselect, with a median-of-medians partition:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 5 | 1 | 3 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 5 | 6 | 4 | 7 | 8 | 9 | 10 | 11 | 12 |

Then we divide in half each time

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

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

# Is it worth it?

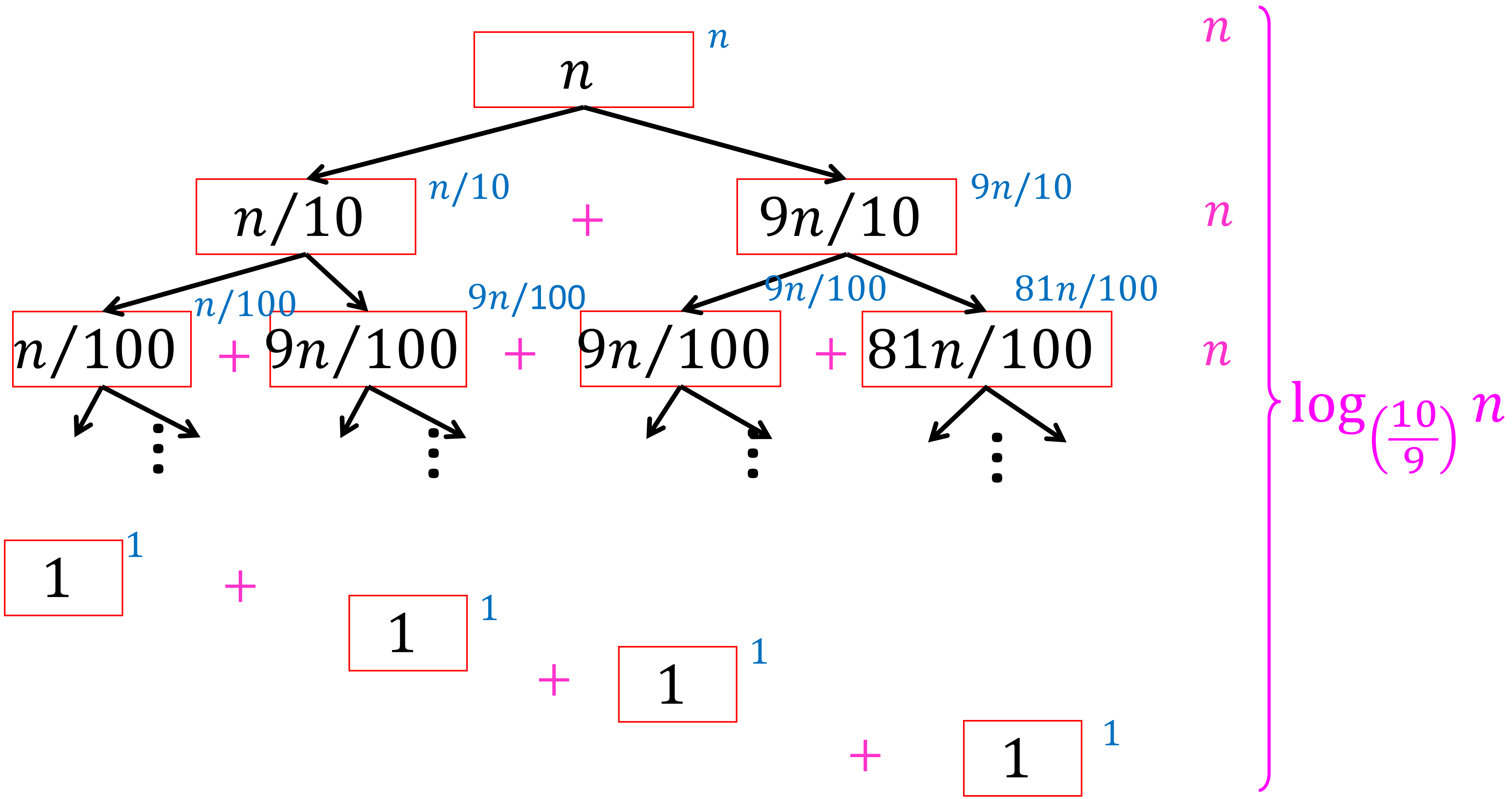
- Using Quickselect to pick median guarantees $\Theta(n \log n)$ run time
- But, this approach has very large constants
  - If you really want $\Theta(n \log n)$, better off using MergeSort
- Better approach: Choose random pivot for Quicksort
  - Very small constant (random() is a fast algorithm)
  - Can prove the *expected runtime* is $\Theta(n \log n)$
    - Why? Unbalanced partitions are very unlikely

# Quicksort Run Time

If the pivot is always $\frac{n}{10}$th order statistic:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$



# Quicksort Run Time

If the pivot is always $\frac{n}{10}$th order statistic:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

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

# Quicksort Run Time

If the pivot is always $d^{\text{th}}$ order statistic:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 2 | 3 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 4 | 7 | 8 | 10 | 9 | 11 | 12 |

Then we shorten by $d$ each time

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

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

What's the probability of this occurring?

# Probability of $n^2$ run time

We must consistently select pivot from within the first $d$ terms

Probability first pivot is among $d$ smallest: $\frac{d}{n}$

Probability second pivot is among $d$ smallest: $\frac{d}{n-d}$

Probability all pivots are among $d$ smallest:

$$\frac{d}{n} \cdot \frac{d}{n-d} \cdot \frac{d}{n-2d} \cdot \ldots \cdot \frac{d}{2d} \cdot 1 = \frac{1}{\left(\frac{n}{d}\right)!}$$

# Sorting, so far

- Sorting algorithms we have discussed:
  - Mergesort $O(n \log n)$
  - Quicksort $O(n \log n)$
- Other sorting algorithms (will discuss):
  - Bubblesort $O(n^2)$
  - Insertionsort $O(n^2)$
  - Heapsort $O(n \log n)$

Can we do better than $O(n \log n)$?

**Mental Stretch**

Show $\log(n!) = \Theta(n \log n)$

Hint: show $n! \leq n^n$

Hint 2: show $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

# $\log n! = O(n \log n)$

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

$$\begin{array}{ccccccccc} & \parallel & & \wedge & & \wedge & & \wedge & \wedge \end{array}$$

$$n^n = n \cdot \quad n \quad \cdot \quad n \quad \cdot \ldots \cdot n \cdot n$$

---

$n! \leq n^n$

$\Rightarrow \log(n!) \leq \log(n^n)$

$\Rightarrow \log(n!) \leq n \log n$

$\Rightarrow \log(n!) = O(n \log n)$

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

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2} - 1\right) \cdot \ldots \cdot 2 \cdot 1$$

$$\vee \quad \vee \quad \vee \quad \| \quad \vee \quad \vee \quad \|$$

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2} \cdot 1 \cdot \ldots \cdot 1 \cdot 1$$

---

$$n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$\Rightarrow \log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right)$$

$$\Rightarrow \log(n!) \geq \frac{n}{2} \log \frac{n}{2}$$

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

# Worst Case Lower Bounds

- Prove that there is no algorithm which can sort faster than $O(n \log n)$
- Non-existence proof!
  - Might seem like it would be hard to do?
  - But we're learning how to do lower bounds proofs!

# Example Lower Bound Proof: Find Min

## Show that finding the minimum of an unordered list requires $\Omega(n)$ comparisons

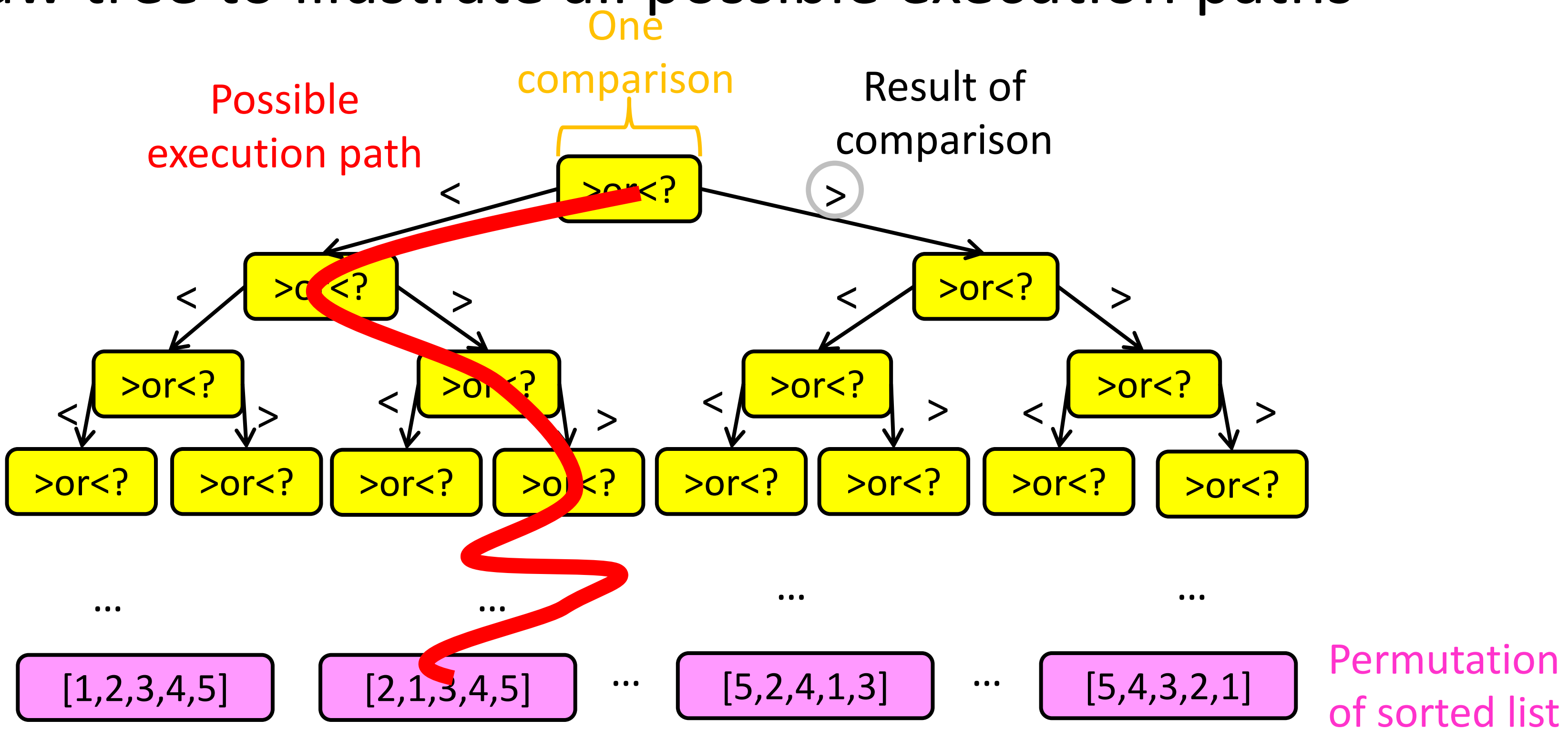
Suppose (toward contradiction) that there is an algorithm for Find Min that does fewer than $\frac{n}{2} = \Omega(n)$ comparisons.

This means there is at least one "uncompared" element
We can't know that this element wasn't the min!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 8 | 19 | 20 | [redacted] | 3 | 9 | -4 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

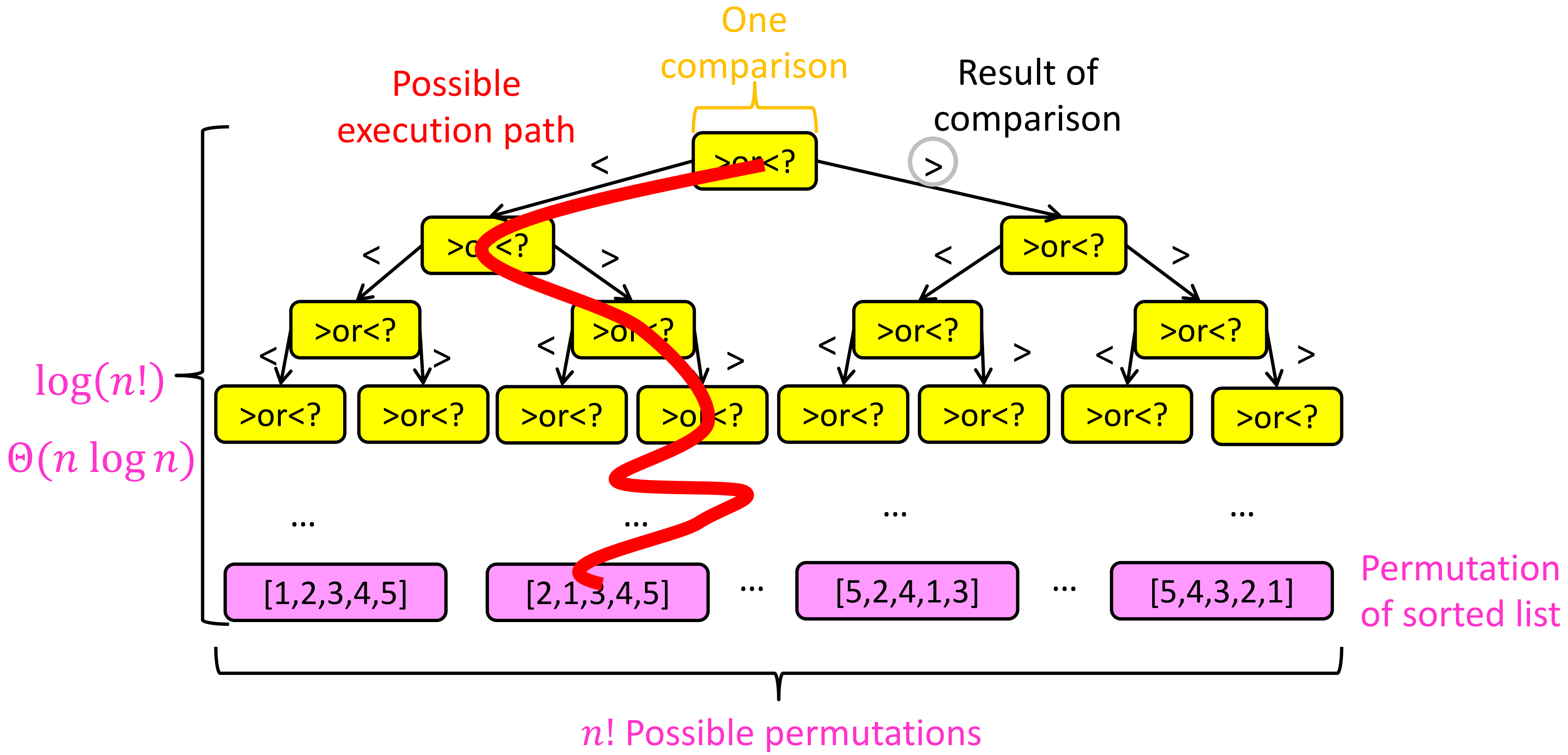
# Strategy: Decision Tree

- Sorting algorithms use comparisons to figure out the order of input elements
- Draw tree to illustrate all possible execution paths



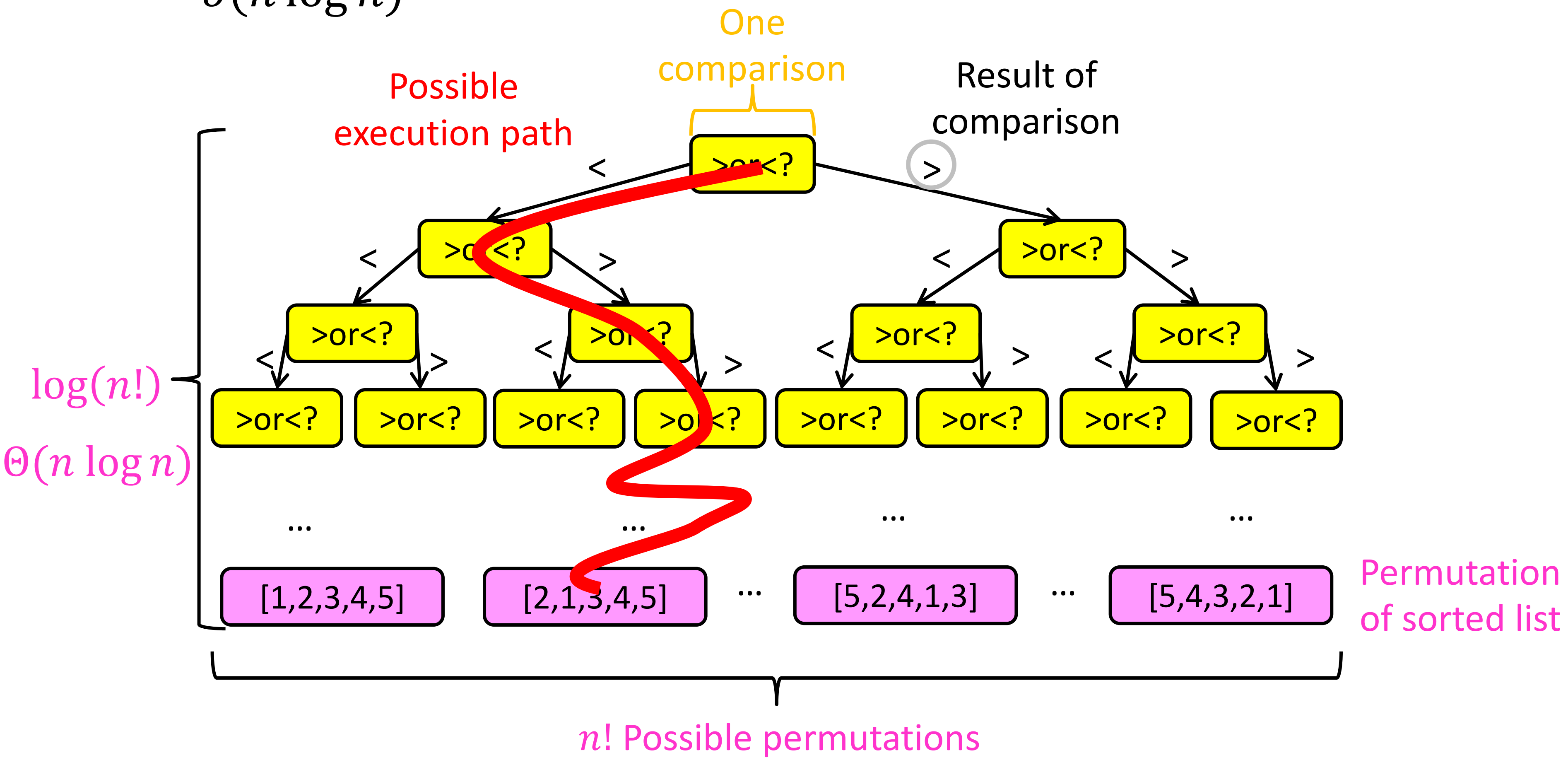
# Strategy: Decision Tree

- Worst case run time is the longest execution path
- i.e., "height" of the decision tree



# Strategy: Decision Tree

- Conclusion: Worst Case Optimal run time of sorting is $\Theta(n \log n)$
  - There is no (comparison-based) sorting algorithm with run time $o(n \log n)$



# Sorting, so far

- Sorting algorithms we have discussed:
  - Mergesort $O(n \log n)$ Optimal!
  - Quicksort $O(n \log n)$ Optimal!
- Other sorting algorithms (will discuss):
  - Bubblesort $O(n^2)$
  - Insertionsort $O(n^2)$
  - Heapsort $O(n \log n)$ Optimal!

# Speed Isn't Everything

Important properties of sorting algorithms:

- Run Time
  - Asymptotic Complexity
  - Constants
- In Place (or In-Situ)
  - Done with only constant additional space
- Adaptive
  - Faster if list is nearly sorted
- Stable
  - Equal elements remain in original order
- Parallelizable
  - Runs faster with multiple computers

# Mergesort

- **Divide**:
  - Break $n$-element list into two lists of $^n/_2$ elements
- **Conquer**:
  - If $n > 1$: Sort each sublist recursively
  - If $n = 1$: List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Run Time?

$\Theta(n \log n)$

Optimal!

| In Place? | Adaptive? | Stable? |
| --- | --- | --- |
| No | No | Yes! (usually) |

# Merge

- **Combine**: Merge sorted sublists into one sorted list
- We have:
  - 2 sorted lists ($L_1, L_2$)
  - 1 output list ($L_{out}$)

While ($L_1$ and $L_2$ not empty):

    If $L_1[0] \leq L_2[0]$:

        $L_{out}$.append($L_1$.pop())

    Else:

        $L_{out}$.append($L_2$.pop())

$L_{out}$.append($L_1$)

$L_{out}$.append($L_2$)

Stable:
If elements are equal, leftmost comes first

# Mergesort

- **Divide**:
  - Break $n$-element list into two lists of $n/2$ elements
- **Conquer**:
  - If $n > 1$: Sort each sublist recursively
  - If $n = 1$: List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Run Time?

$\Theta(n \log n)$

Optimal!

| In Place? | Adaptive? | Stable? | Parallelizable? |
| --- | --- | --- | --- |
| No | No | Yes! (usually) | Yes! |

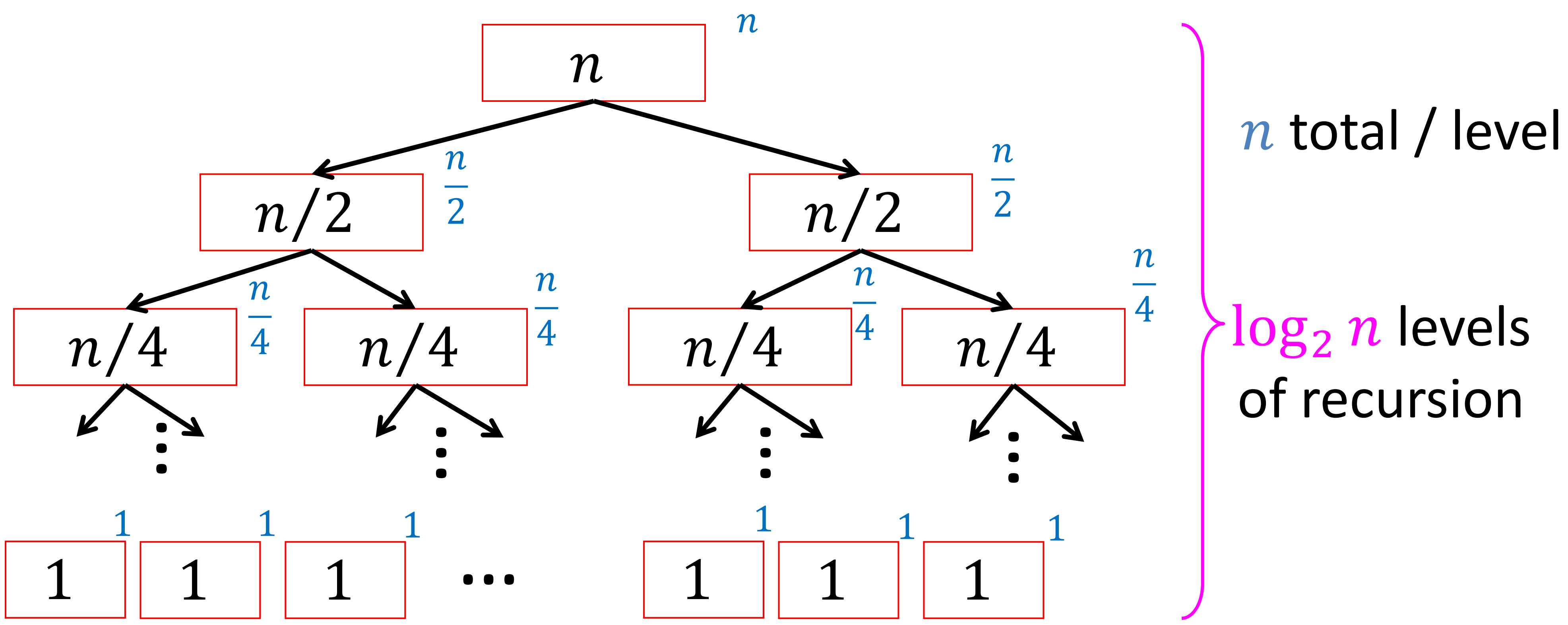
# Mergesort

- **Divide**:
  - Break $n$-element list into two lists of $^n/_2$ elements
- **Conquer**:
  - If $n > 1$:
    - Sort each sublist recursively
  - If $n = 1$:
    - List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Parallelizable: Allow different machines to work on each sublist

# Mergesort (Sequential)

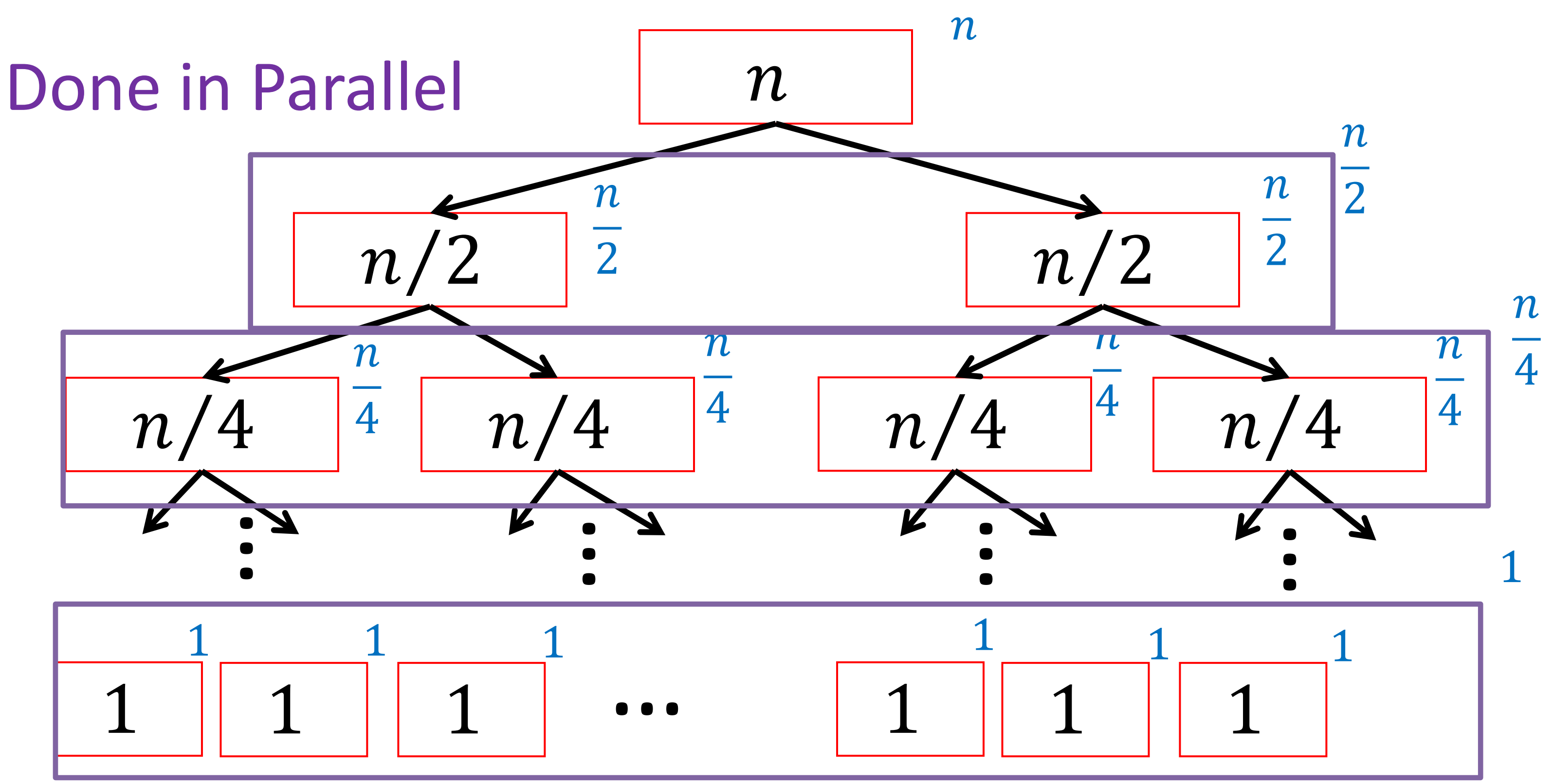
$$T(n) = 2T(\frac{n}{2}) + n$$




Run Time: $\Theta(n \log n)$

# Mergesort (Parallel)

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




Run Time: $\Theta(n)$

# Quicksort

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

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

Run Time?

$\Theta(n \log n)$

(almost always)

Better constants than Mergesort

In Place?

kinda

Uses stack for recursive calls

Adaptive?

No!

Stable?

No

Parallelizable?

Yes!

- Horton’s lecture on 2/15 stopped here (or one slide earlier?)

# Bubble Sort

Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 5 | 7 | 9 | 12 | 10 | 1 | 2 | 4 | 3 | 6 | 11 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 7 | 9 | 12 | 10 | 1 | 2 | 4 | 3 | 6 | 11 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | 9 | 12 | 10 | 1 | 2 | 4 | 3 | 6 | 11 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | 9 | 12 | 10 | 1 | 2 | 4 | 3 | 6 | 11 |

# Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

## Run Time?

$\Theta(n^2)$

Constants worse than Insertion Sort

## In Place?

Yes

## Adaptive?

Kinda

“Compared to straight insertion […], bubble sorting requires a more complicated program and takes about twice as long!” –Donald Knuth

# Bubble Sort is "almost" Adaptive

Idea: March through list, swapping adjacent elements if out of order

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Only makes one "pass"

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 1 |

After one "pass"

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 1 | 12 |

Requires $n$ passes, thus is $O(n^2)$

# Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

**Run Time?**

$\Theta(n^2)$

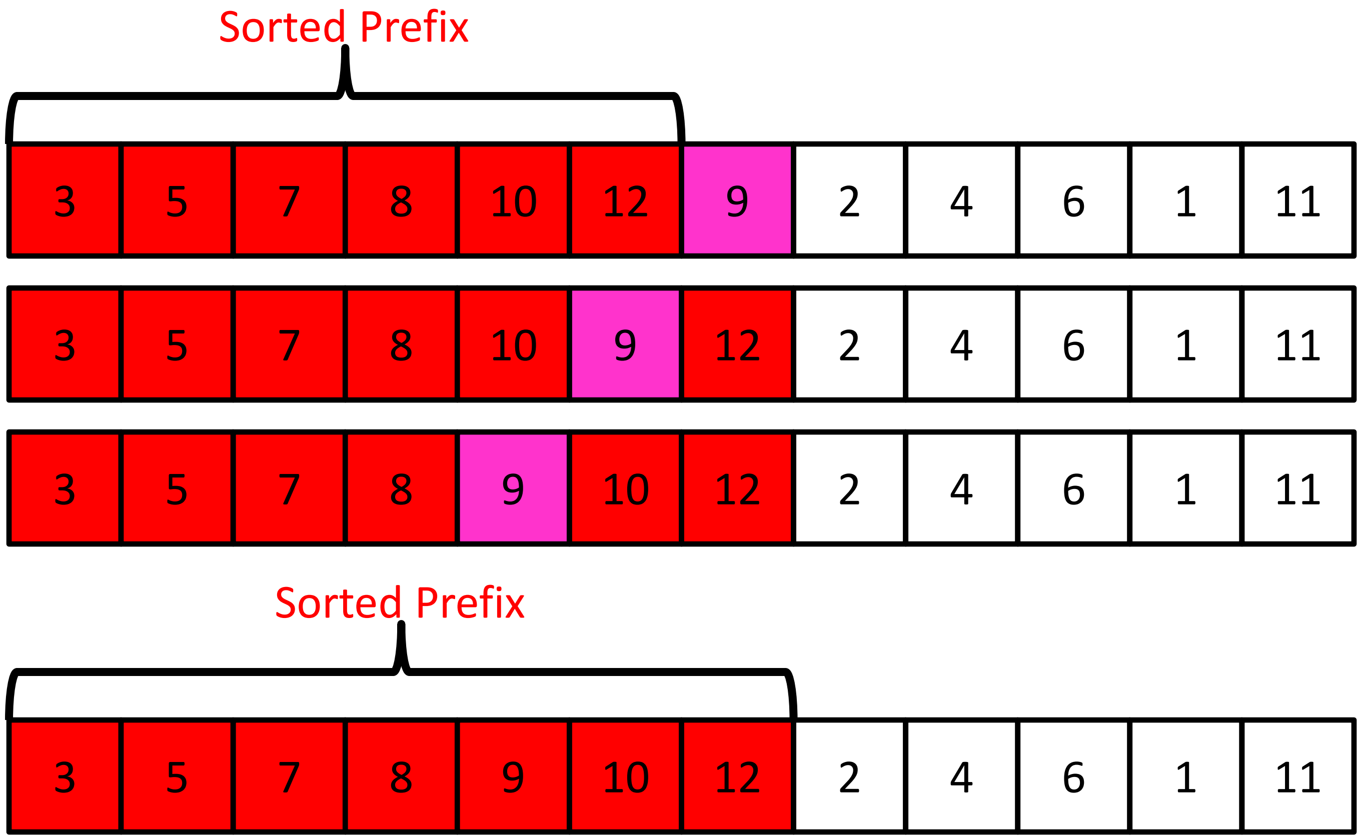
Constants worse than Insertion Sort

| In Place? | Adaptive? | Stable? | Parallelizable? |
|---|---|---|---|
| Yes! | ~~Kinda~~ Not really | Yes | No |

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems" –Donald Knuth, The Art of Computer Programming

# Insertion Sort

Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element



# Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element

**Run Time?**

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

**In Place?**

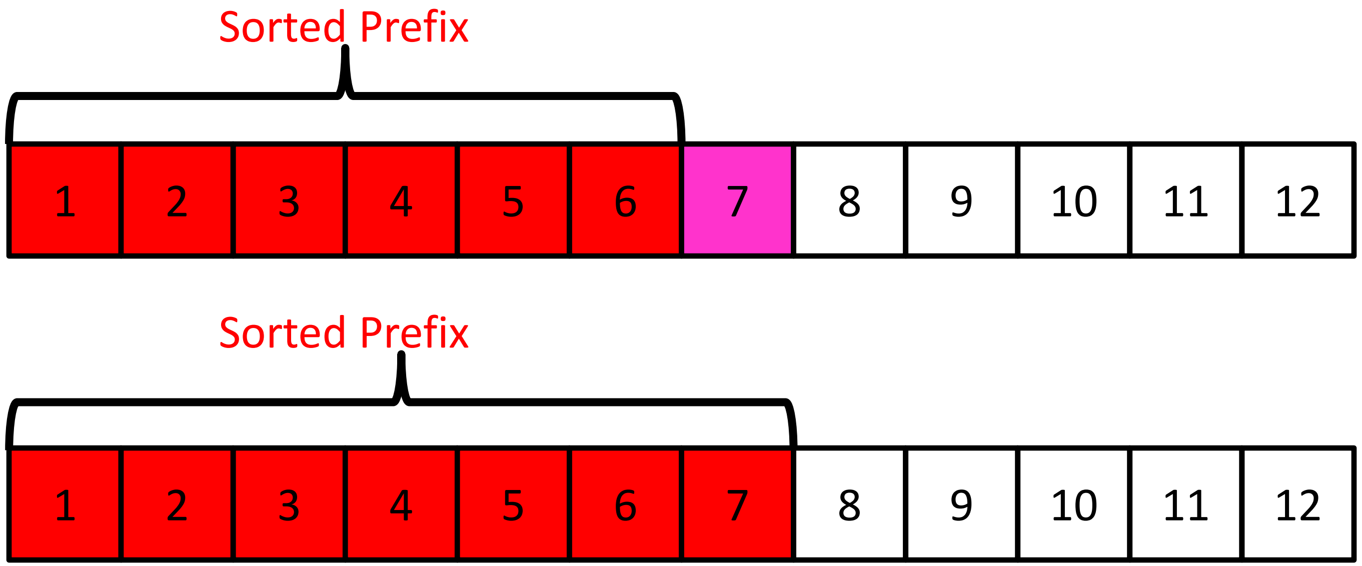
Yes!

**Adaptive?**

Yes

# Insertion Sort is Adaptive

Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element




Only one comparison needed per element!   Runtime: $O(n)$

# Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element

**Run Time?**

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

**In Place?**

Yes!

**Adaptive?**

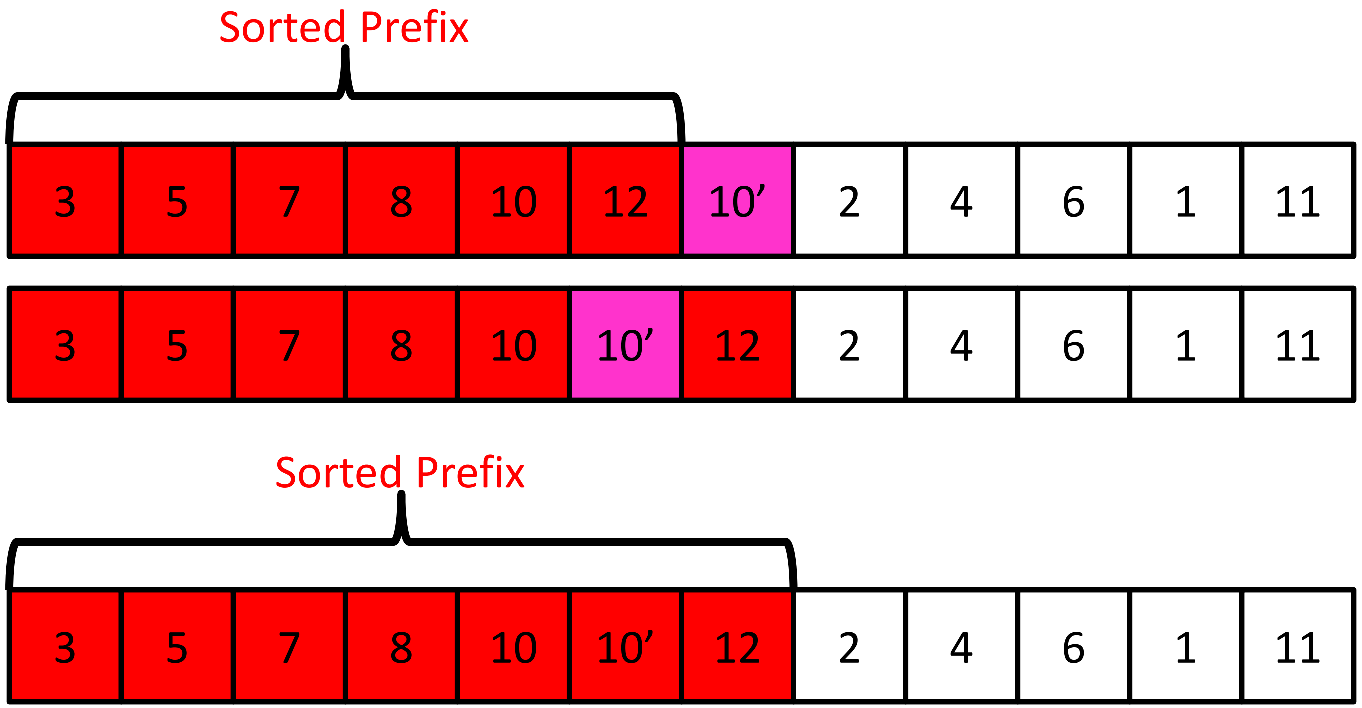
Yes

**Stable?**

Yes

# Insertion Sort is Stable

- Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element




The “second” 10 will stay to the right

# Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by "inserting" the next element

**Run Time?**

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

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

"All things considered, it's actually a pretty good sorting algorithm!" –Nate Brunelle

Can sort a list as it is received, i.e., don't need the entire list to begin sorting

**Online?**

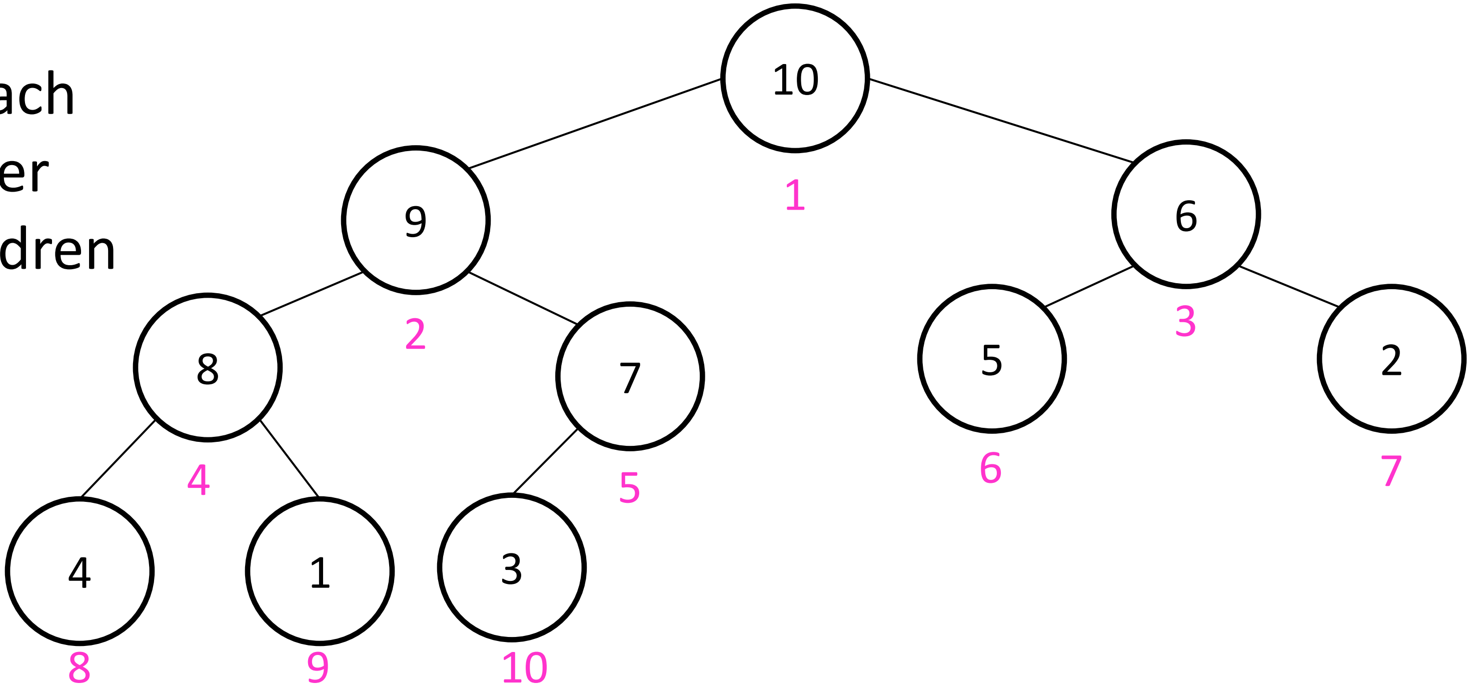
Yes

# Heap Sort

- Idea: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 10 | 9 | 6 | 8 | 7 | 5 | 2 | 4 | 1 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Max Heap Property: Each node is larger than its children

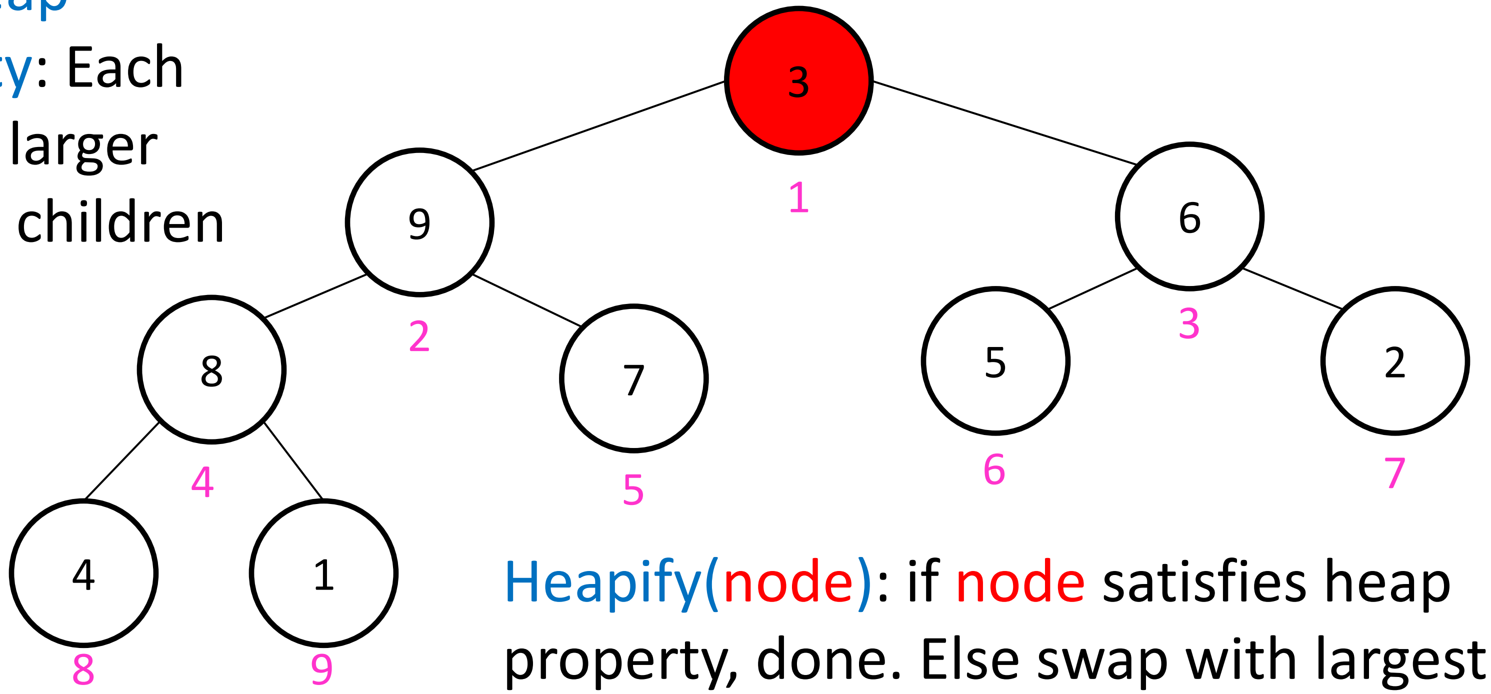


# Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 9 | 6 | 8 | 7 | 5 | 2 | 4 | 1 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

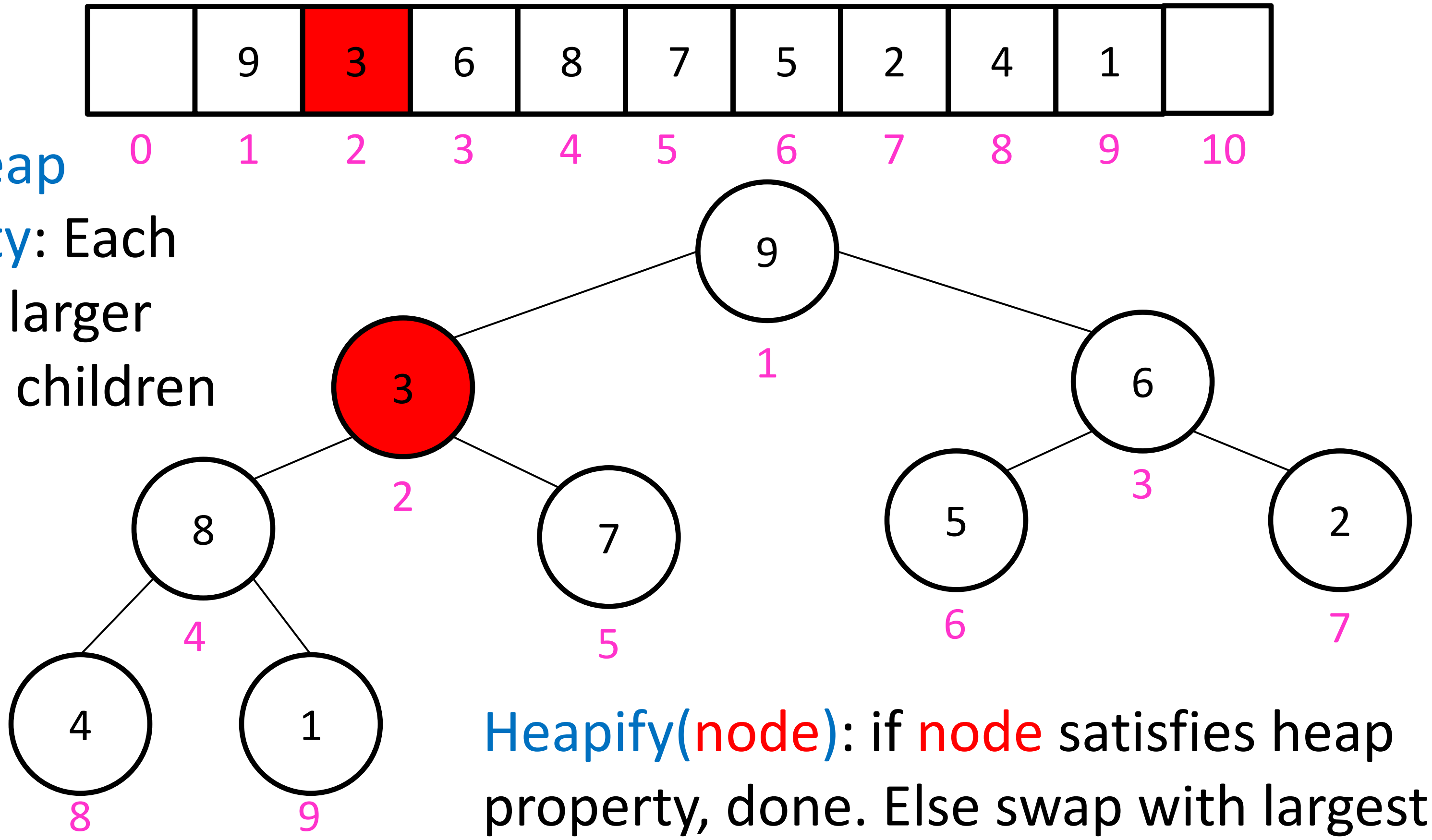
Max Heap Property: Each node is larger than its children




Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

# Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

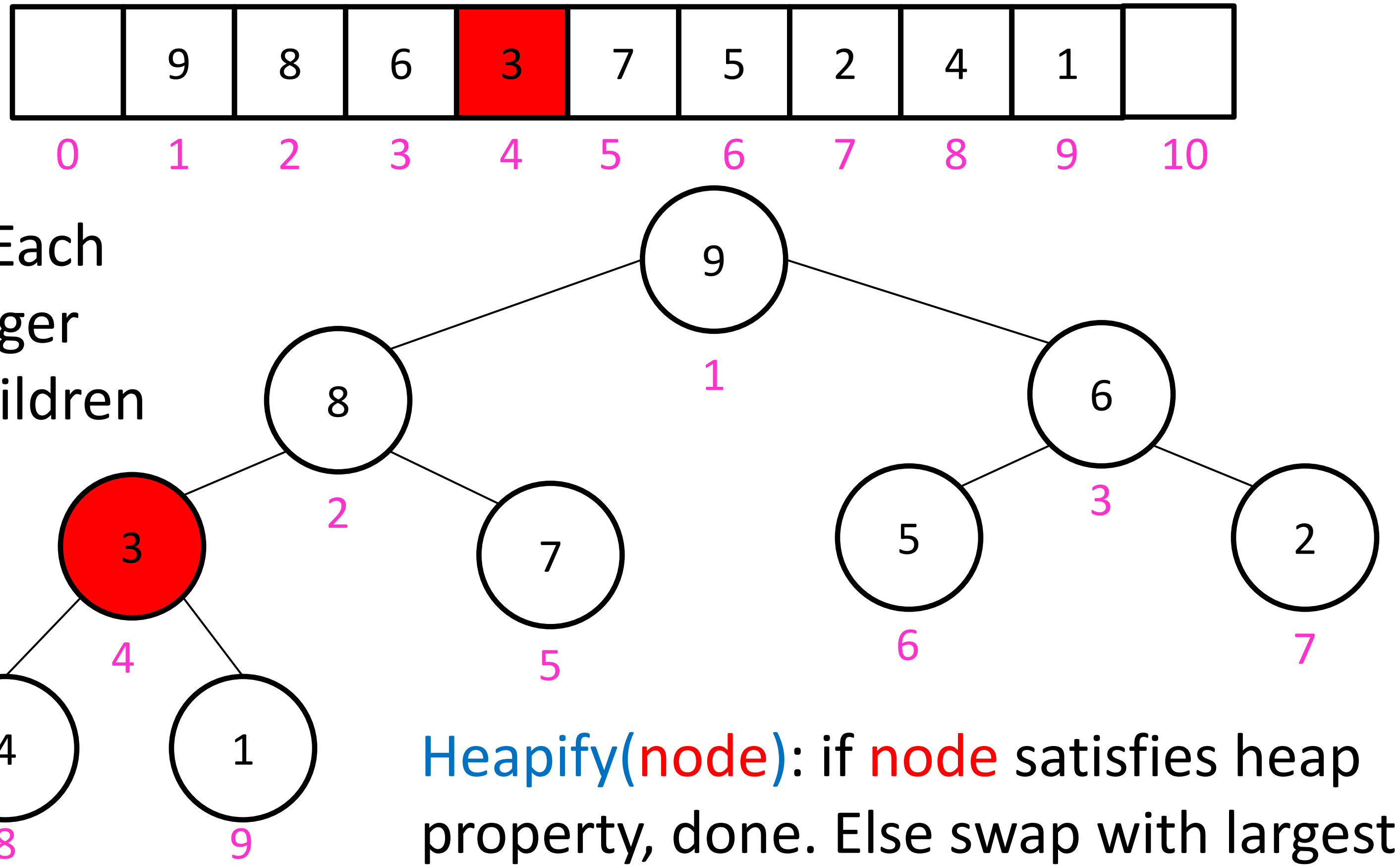



Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

# Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)




Max Heap Property: Each node is larger than its children

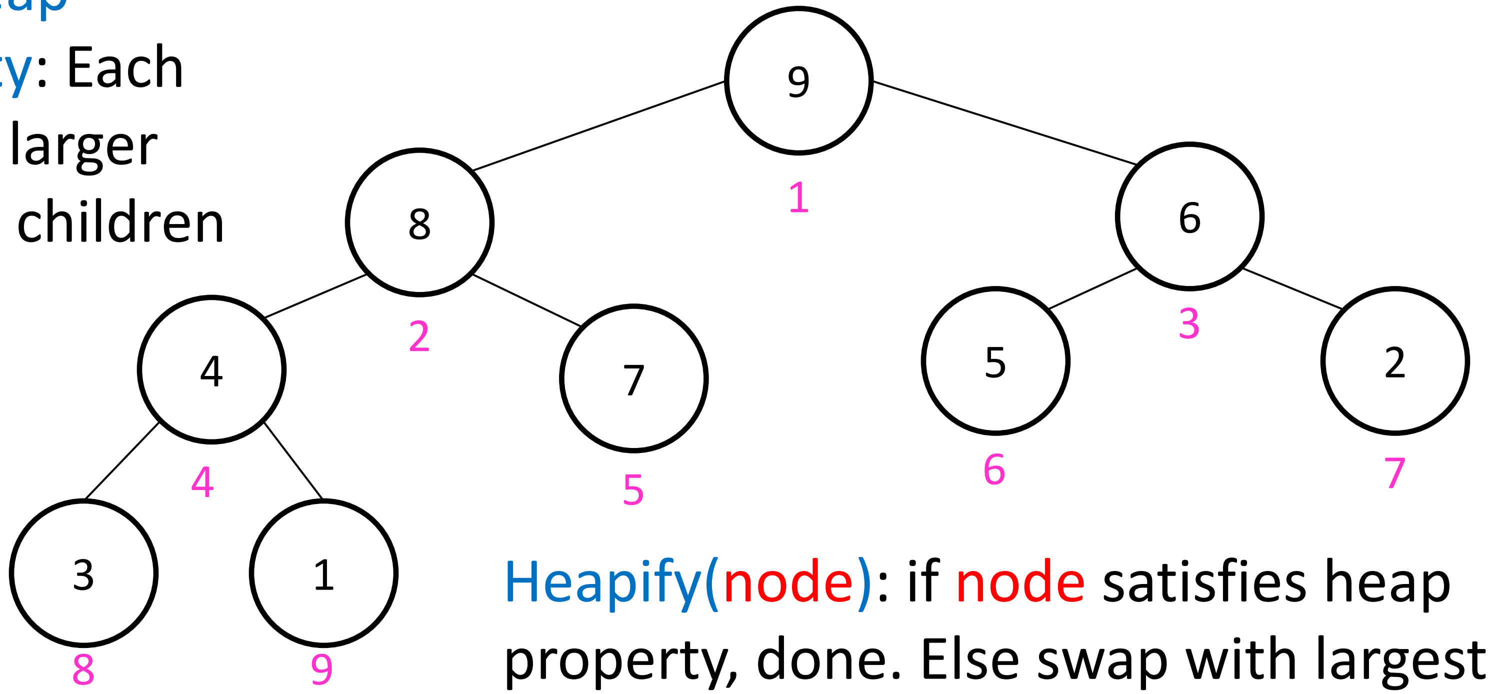
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

# Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

| | 9 | 8 | 6 | 4 | 7 | 5 | 2 | 3 | 1 | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Max Heap Property: Each node is larger than its children




Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

# Heap Sort

- Idea: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

**Run Time?**

$\Theta(n \log n)$

Constants worse than Quick Sort

**In Place?**

Yes!

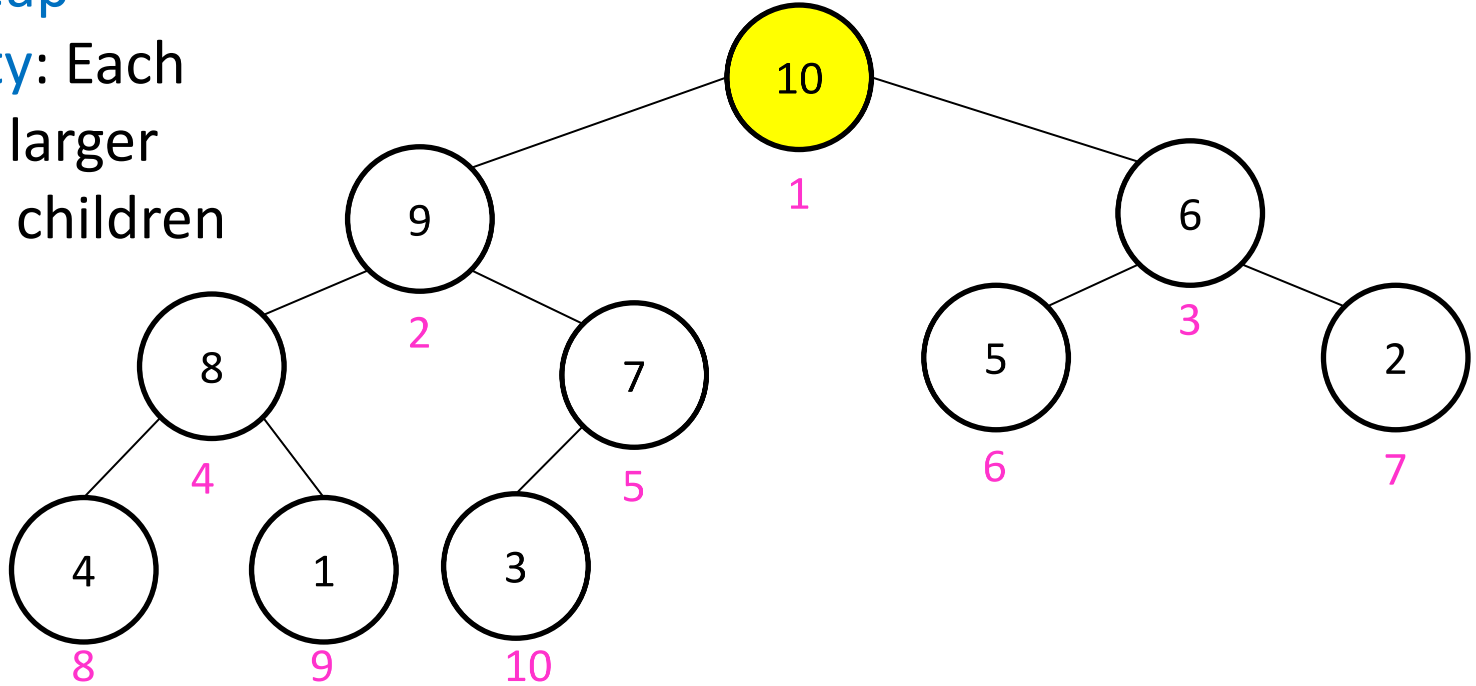
When removing an element from the heap, move it to the (now unoccupied) end of the list

# In Place Heap Sort

- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

| | 10 | 9 | 6 | 8 | 7 | 5 | 2 | 4 | 1 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

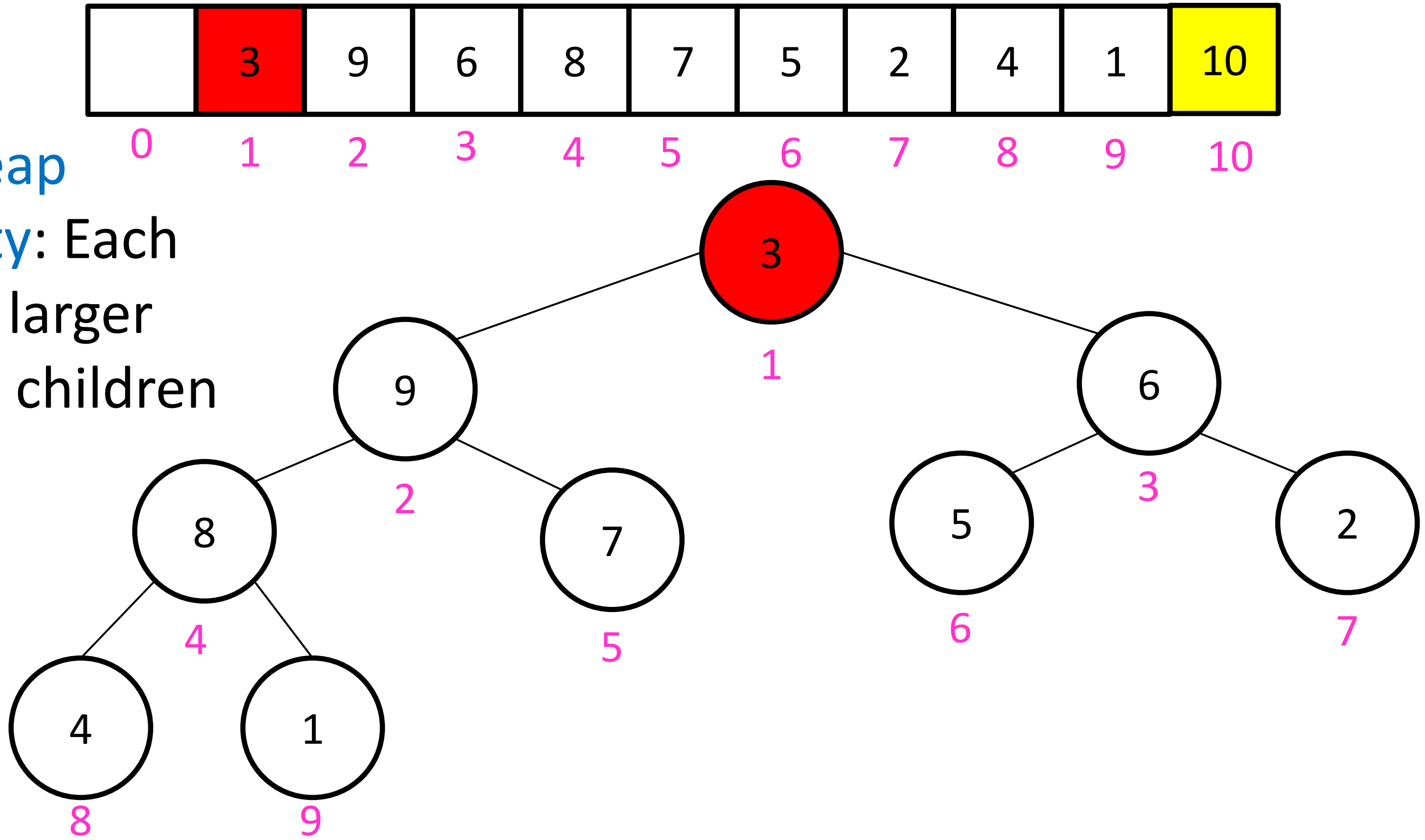
Max Heap Property: Each node is larger than its children



# In Place Heap Sort

- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

Max Heap Property: Each node is larger than its children

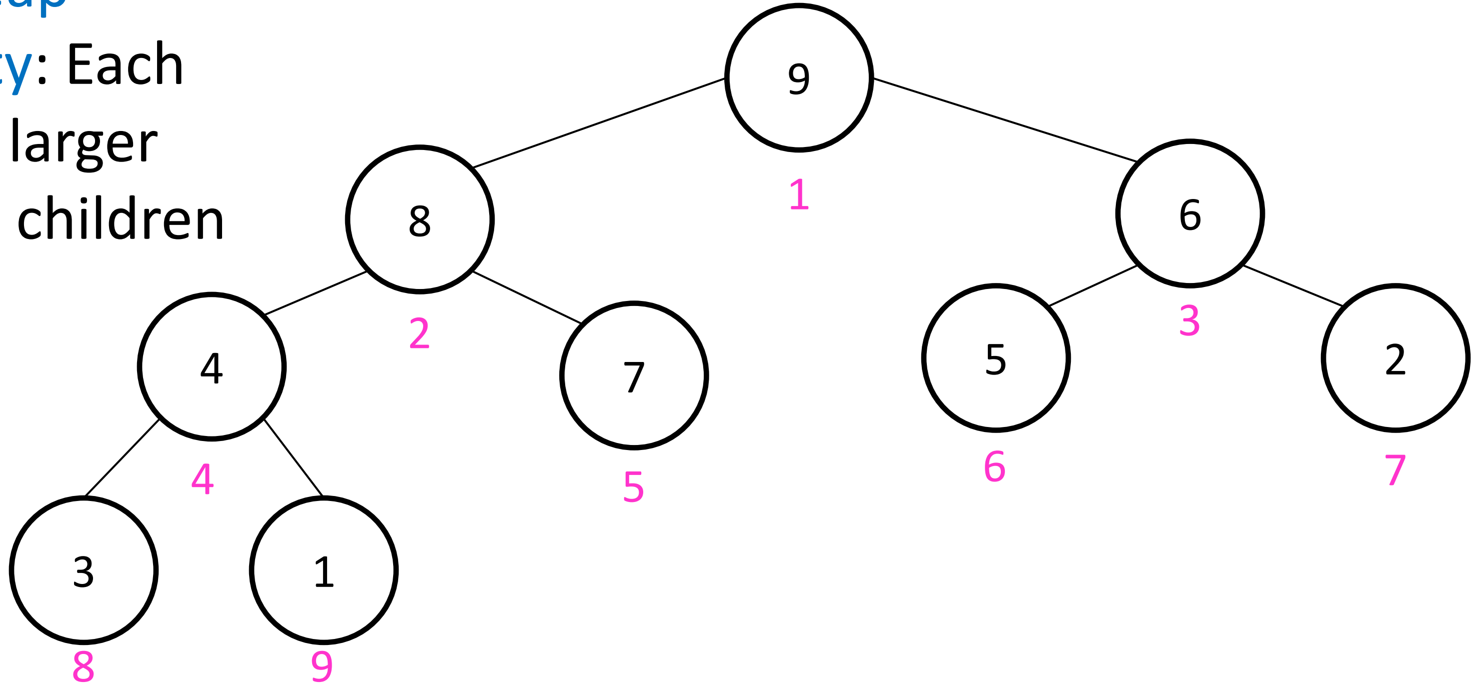


# In Place Heap Sort

- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 9 | 8 | 6 | 4 | 7 | 5 | 2 | 3 | 1 | 10 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Max Heap Property: Each node is larger than its children

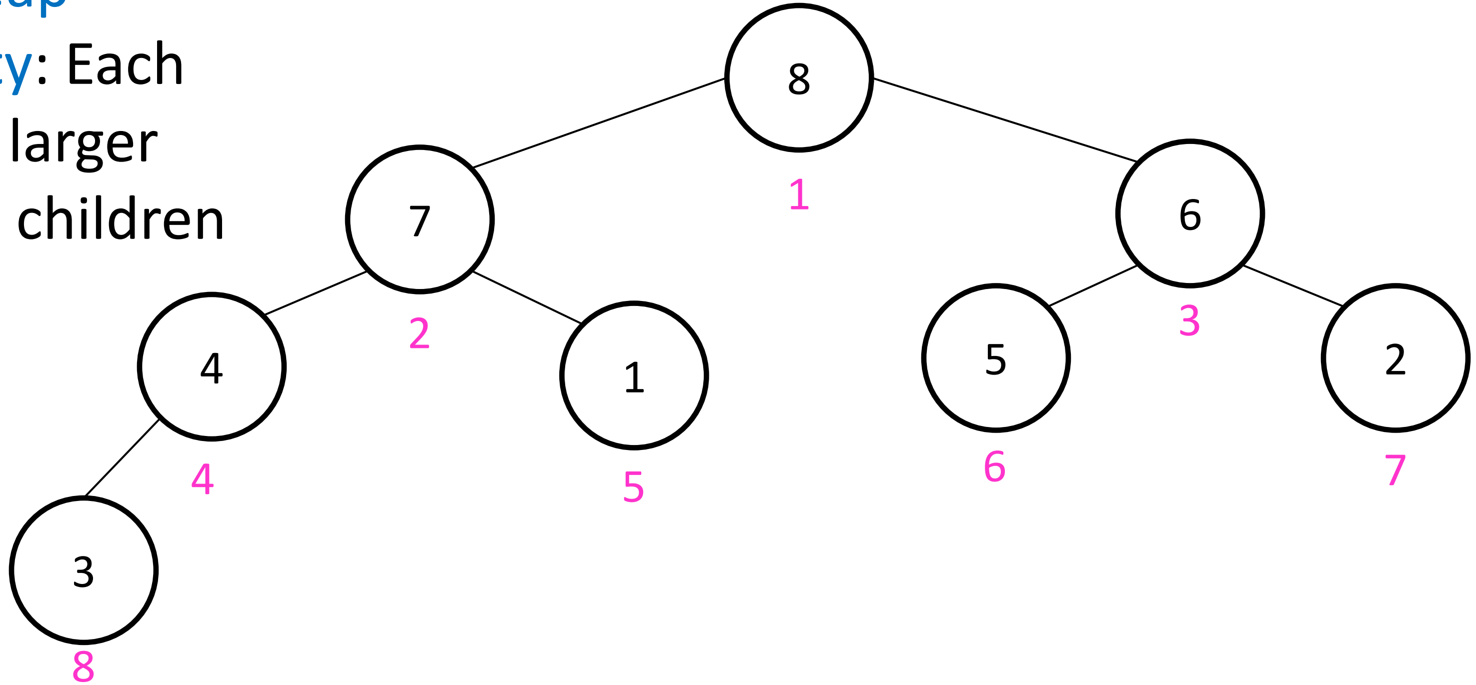


# In Place Heap Sort

- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

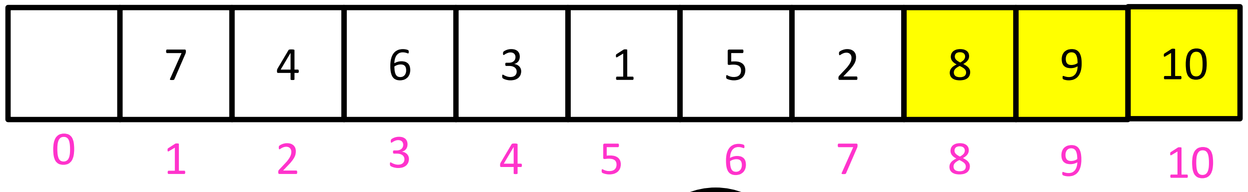
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 8 | 7 | 6 | 4 | 1 | 5 | 2 | 3 | 9 | 10 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Max Heap Property: Each node is larger than its children

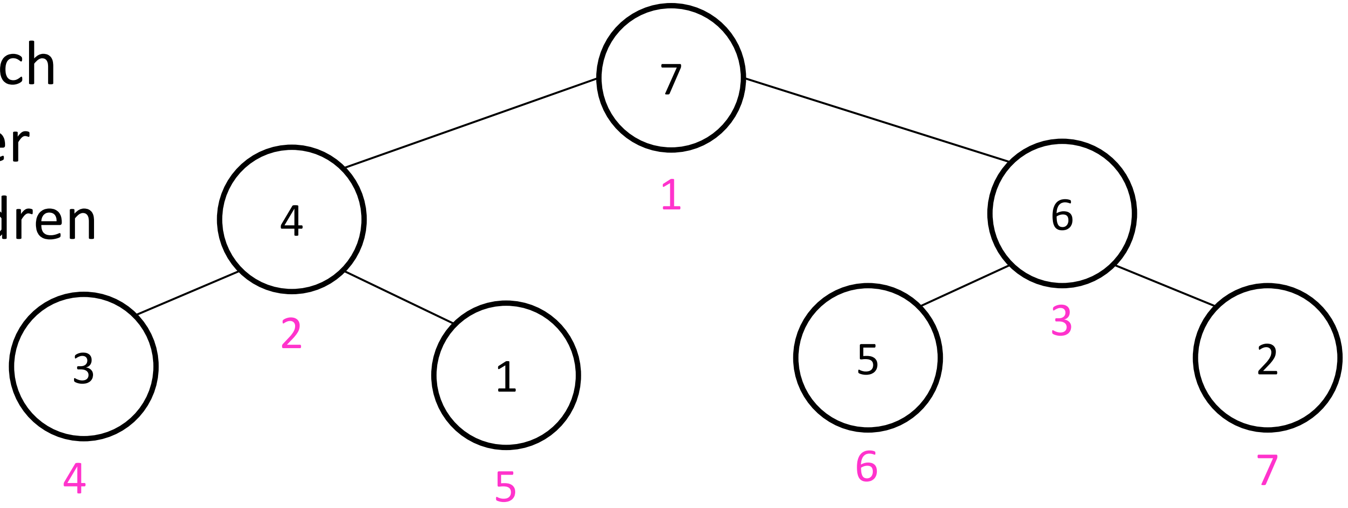


# In Place Heap Sort

- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list




Max Heap Property: Each node is larger than its children



# Heap Sort

- Idea: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

**Run Time?**

$\Theta(n \log n)$

Constants worse than Quick Sort

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

# Sorting, so far

- Sorting algorithms we have discussed:
  - Mergesort $O(n \log n)$ Optimal!
  - Quicksort $O(n \log n)$ Optimal!
- Other sorting algorithms (will discuss):
  - Bubblesort $O(n^2)$
  - Insertionsort $O(n^2)$
  - Heapsort $O(n \log n)$ Optimal!