CS 4102: Algorithms

Lecture 8: Sorting Algorithms

David Wu

Fall 2019

Warm Up

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

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

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

Warm Up

$$n! = n \cdot \overset{<n}{(n-1)} \cdot \overset{<n}{(n-2)} \cdot \cdots \cdot \overset{<n}{2} \cdot \overset{<n}{1}$$

$$n! \leq n^n$$

$$\Rightarrow \log(n!) \leq \log(n^n) = n \log n \in O(n \log n)$$

Warm Up

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

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

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

Today’s Keywords

Divide and Conquer

Sorting Algorithms

Quicksort

Decision Tree

Sorting Lower Bounds

CLRS Readings: Chapter 7 and 8

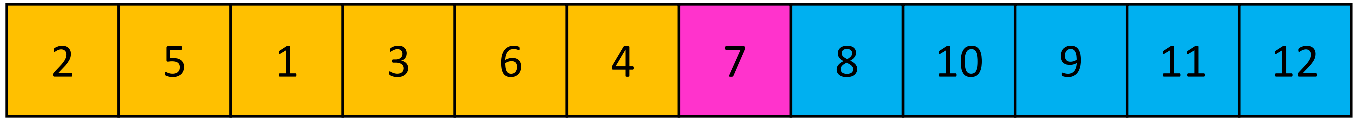
Homework

- **HW3** due **Tuesday, October 1, 11pm**
  - Divide and conquer algorithms
  - Written (use LaTeX!) – Submit both **zip** and **pdf**!

- **HW0** grades posted on Collab
  - **Regrade office hours:**
    - Thursday 11am-12pm (Rice 210)
    - Thursday 4pm-5pm (Rice 501)
  - Please be prepared to verbally explain your submission if requesting a regrade

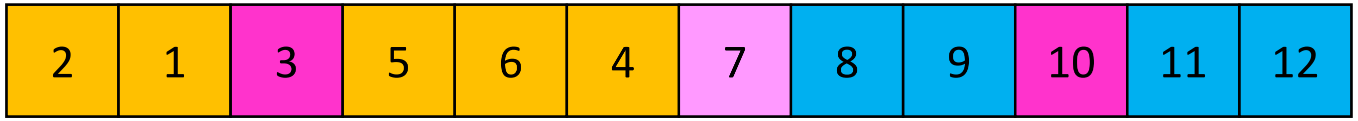
Randomized Quicksort

Divide: Select a random pivot, and partition about the pivot



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

Conquer: Recursively sort left and right sublists

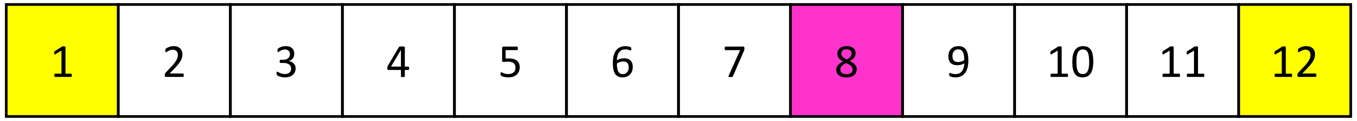


2	1	3	5	6	4	7	8	9	10	11	12

Expected running time: $O(n \log n)$

Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?



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

Consider the sorted version of the list

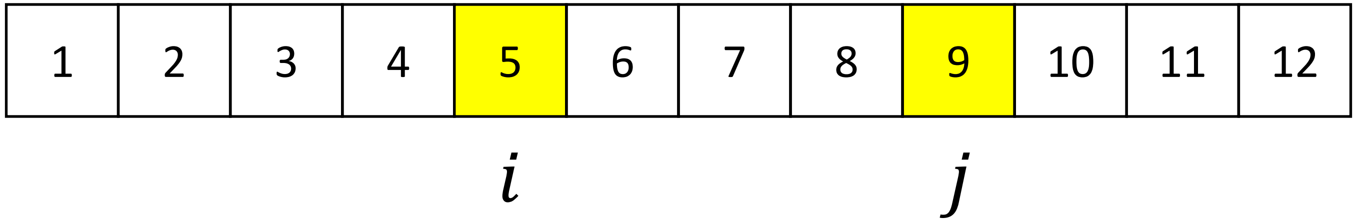
$$\Pr[\text{we compare } 1 \text{ and } 12] = \frac{2}{12}$$

Assuming pivot is chosen uniformly at random

Elements only compared if 1 or 12 was chosen as the first pivot since otherwise they are in different sublists

Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?



1	2	3	4	5	6	7	8	9	10	11	12
				i				j			

Case 1: Pivot less than i

$$\Pr[\text{we compare } i \text{ and } j] = \Pr[\text{we compare } i \text{ and } j \text{ in Quicksort}([p+1, \dots, n])]$$

Case 2: Pivot greater than j

$$\Pr[\text{we compare } i \text{ and } j] = \Pr[\text{we compare } i \text{ and } j \text{ in Quicksort}([1, \dots, p])]$$

Case 3: Pivot in $[i, i+1, \dots, j]$

$$\Pr[\text{we compare } i \text{ and } j] = \Pr[i \text{ or } j \text{ is selected as pivot}] = \frac{2}{j - i + 1}$$

Formal Argument for $n \log n$ Average

Probability of comparing element i with element j:

$$\Pr[\text{we compare } i \text{ and } j] = \frac{2}{j - i + 1}$$

Formal Argument for $n \log n$ Average

Probability of comparing element i with element j:

$$\Pr[\text{we compare } i \text{ and } j] = \frac{2}{j-i+1}$$

<u>Expected</u> number of comparisons:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

Substitution:
$k = j - i$

$$\frac{1}{k+1} < \frac{1}{k}$$

Formal Argument for $n \log n$ Average

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

Substitution:
$k = j - i$

$$\frac{1}{k+1} < \frac{1}{k}$$

Useful fact: $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$

Intuition (not proof!):

$$\sum_{k=1}^{n} \frac{1}{k} \approx \int_{1}^{n} \frac{1}{x}\,dx = \ln n$$

Formal Argument for $n \log n$ Average

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

$$= 2 \sum_{i=1}^{n-1} \Theta(\log n) = \Theta(n \log n)$$

Useful fact: $\displaystyle\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$

Sorting Algorithms

Sorting algorithms we have discussed:

- Mergesort $O(n \log n)$
- Quicksort $O(n \log n)$

Other sorting algorithms (will discuss):

- Bubble sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heapsort $O(n \log n)$

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

Worst Case Lower Bounds

Prove that there is no algorithm which can sort faster than $O(n \log n)$

Non-existence proof!

- Very hard to do

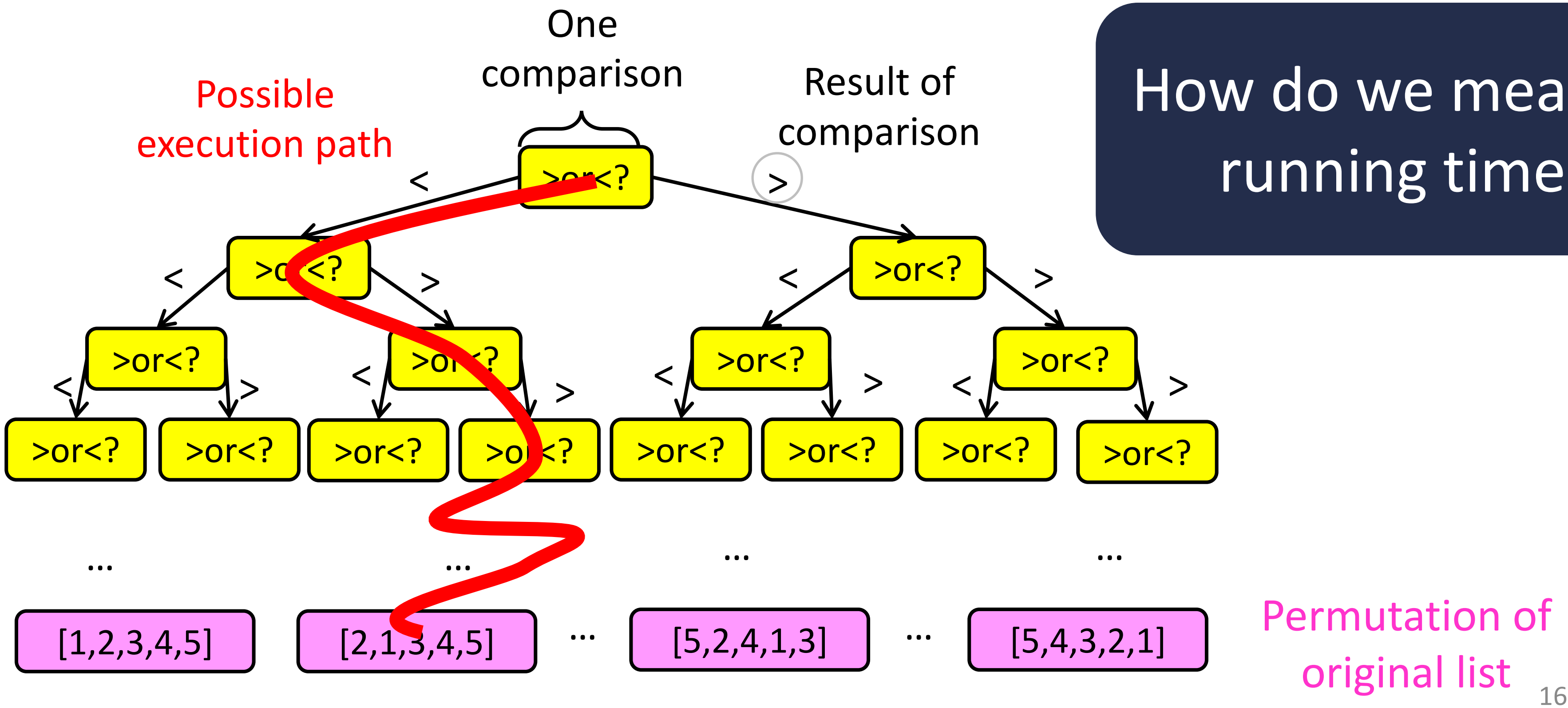
We will show such a lower bound for comparison sorts

Algorithm that only assumes elements can be compared (nothing about representation of the elements)

Strategy: Decision Tree

Comparison sorts use <u>comparisons</u> to determine ordering

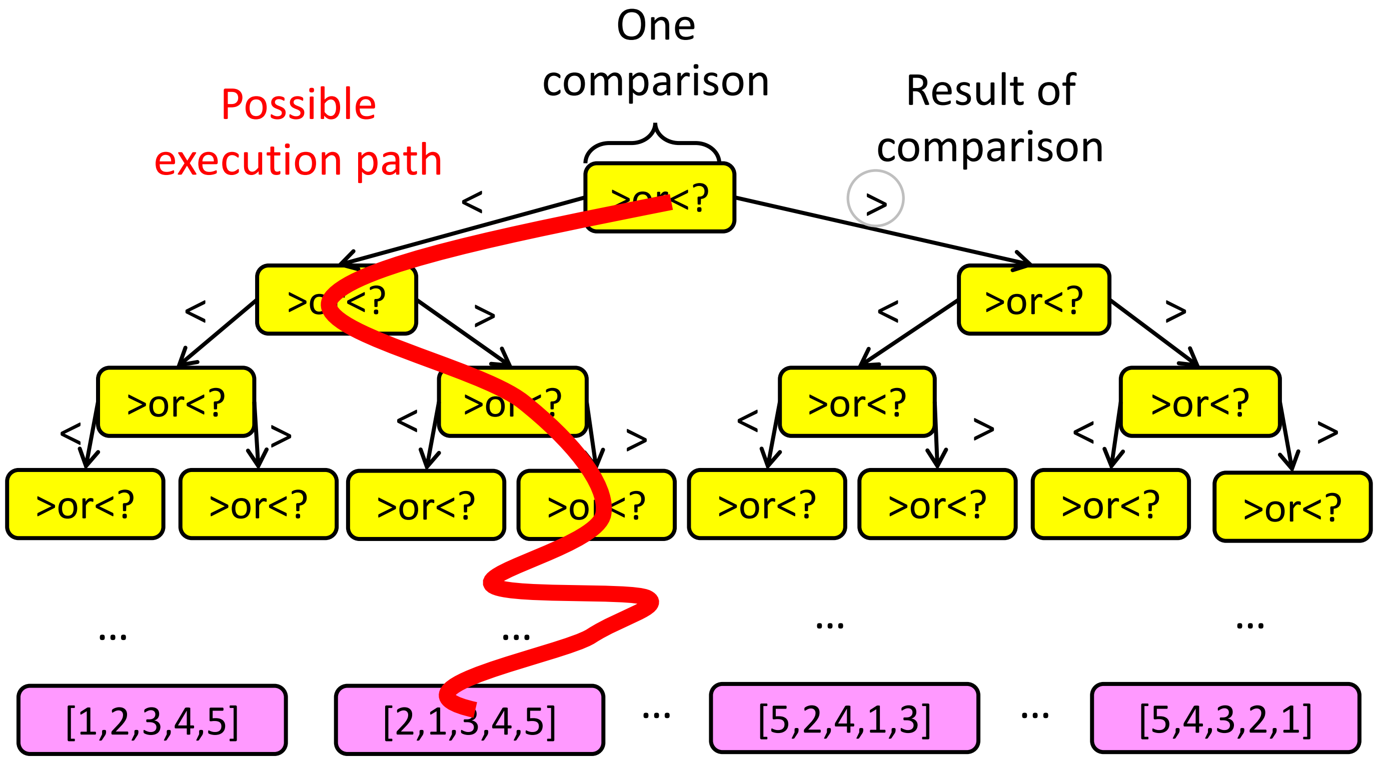
Strategy: Draw tree to illustrate all possible execution paths



How do we measure running time?

Strategy: Decision Tree

Worst case running time is the longest execution path (measures number of comparisons) – this is the <u>height</u> of the decision tree

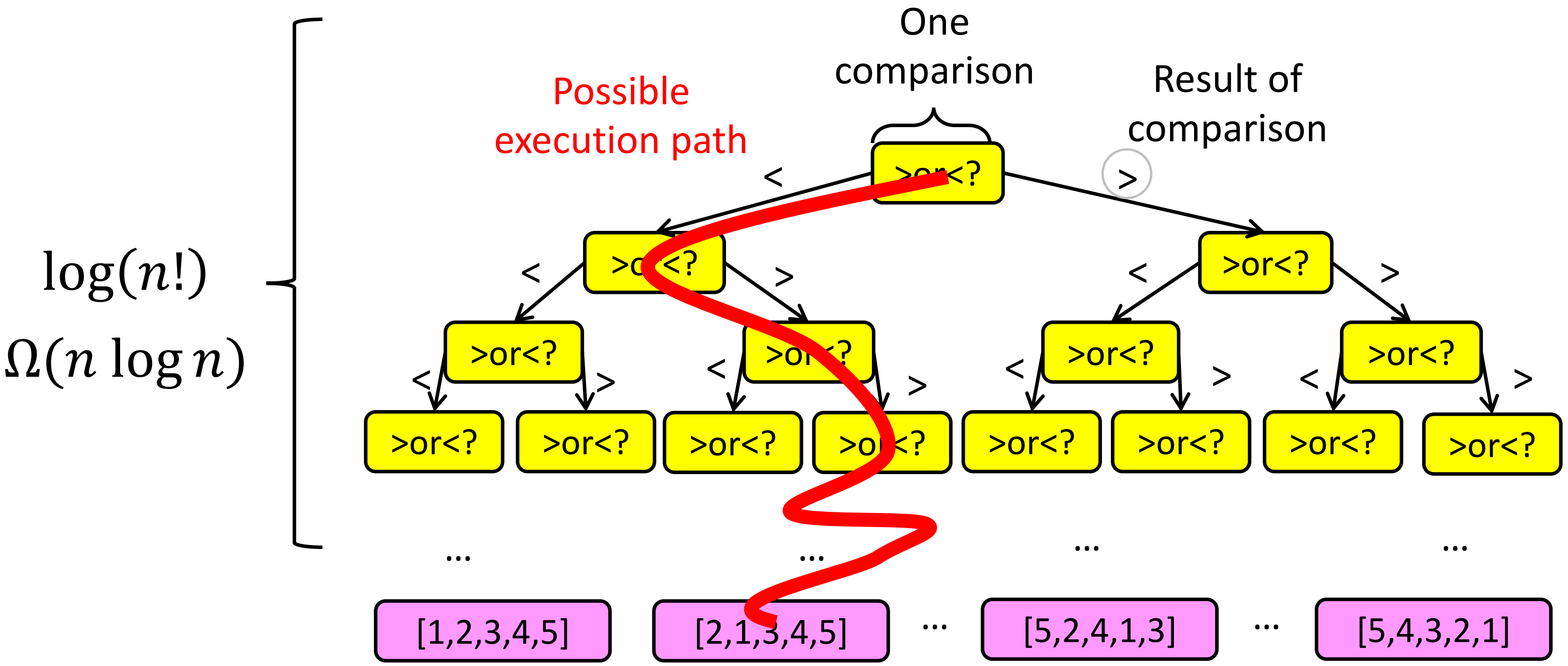


How many such permutations do we need?

Permutation of original list

Strategy: Decision Tree

Worst case running time is the longest execution path (measures number of comparisons) – this is the height of the decision tree



$\log(n!)$

$\Omega(n \log n)$

One comparison

Possible execution path

Result of comparison

>or<?

< >

>or<? >or<?

>or<? >or<? >or<? >or<?

>or<? >or<? >or<? >or<? >or<? >or<? >or<? >or<?

… … … …

[1,2,3,4,5] [2,1,3,4,5] … [5,2,4,1,3] … [5,4,3,2,1]

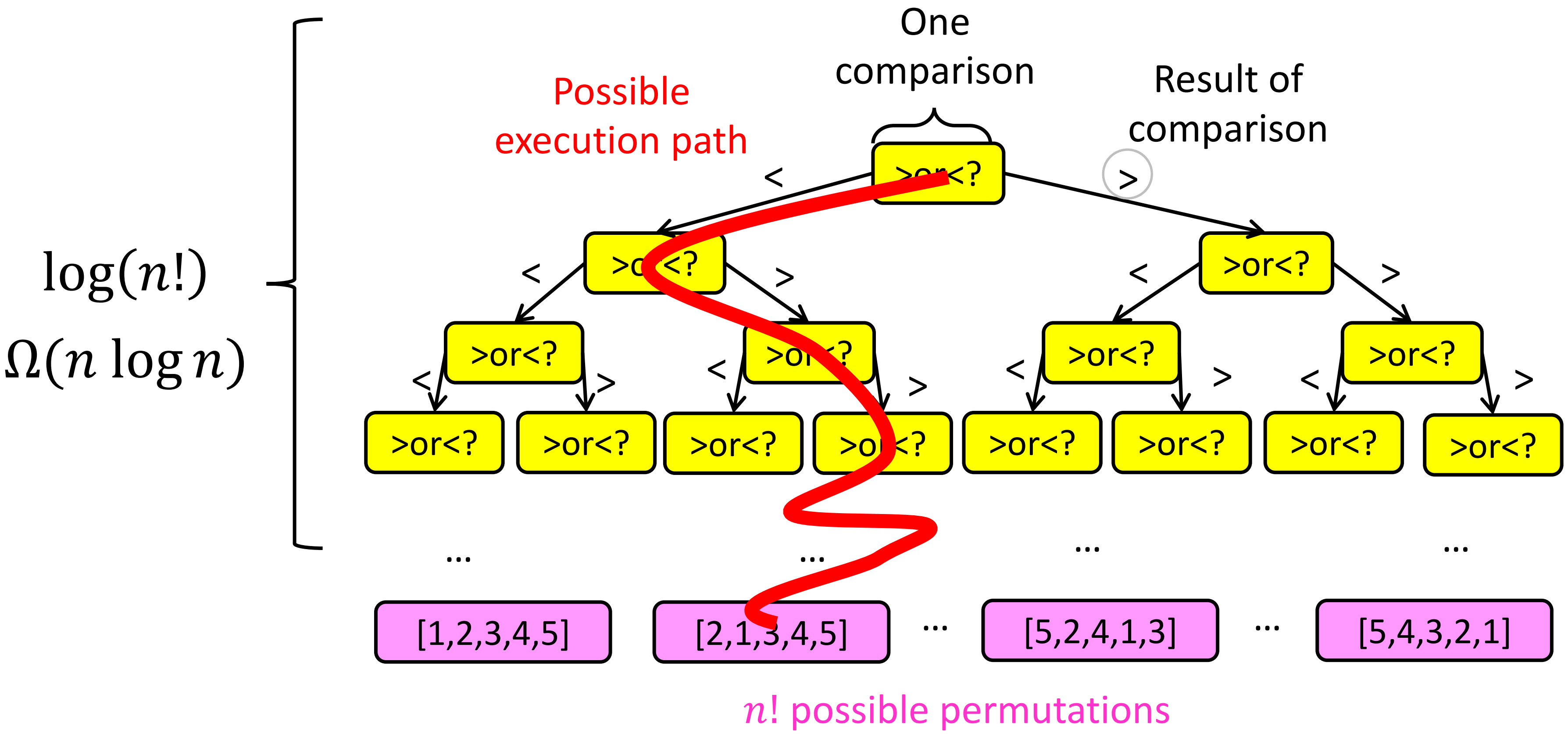
$n!$ possible permutations

How many such permutations do we need?

Permutation of original list

Strategy: Decision Tree

Conclusion: Running time of any comparison sort is $\Omega(n \log n)$



How many such permutations do we need?

Permutation of original list

Sorting Algorithms

Sorting algorithms we have discussed:

• Mergesort	$O(n \log n)$	Optimal!
• Quicksort	$O(n \log n)$	Optimal!

Other sorting algorithms (will discuss):

• Bubble sort	$O(n^2)$	
• Insertion sort	$O(n^2)$	
• Heapsort	$O(n \log n)$	Optimal!

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

Not with comparison sorts…

Speed Isn’t Everything

Important properties of sorting algorithms:

Run Time

- Asymptotic Complexity
- Constants

In Place

- Only requires constant additional space

Relaxed definition: only need to copy a constant number of elements

Adaptive

- Faster if list is nearly sorted

Stable

- Equal elements remain in original order

Parallelizable

- Runs faster with many processors

Merge Sort

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?

$O(n \log n)$

Optimal!

In Place?	Adaptive?	Stable?
No	No	Yes*

Technically: depends on how merge is implemented

Merge Sort

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

Merge Sort

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?

$O(n \log n)$

Optimal!

In Place?	Adaptive?	Stable?	Parallelizable?
No	No	Yes*	Yes

Merge Sort

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)

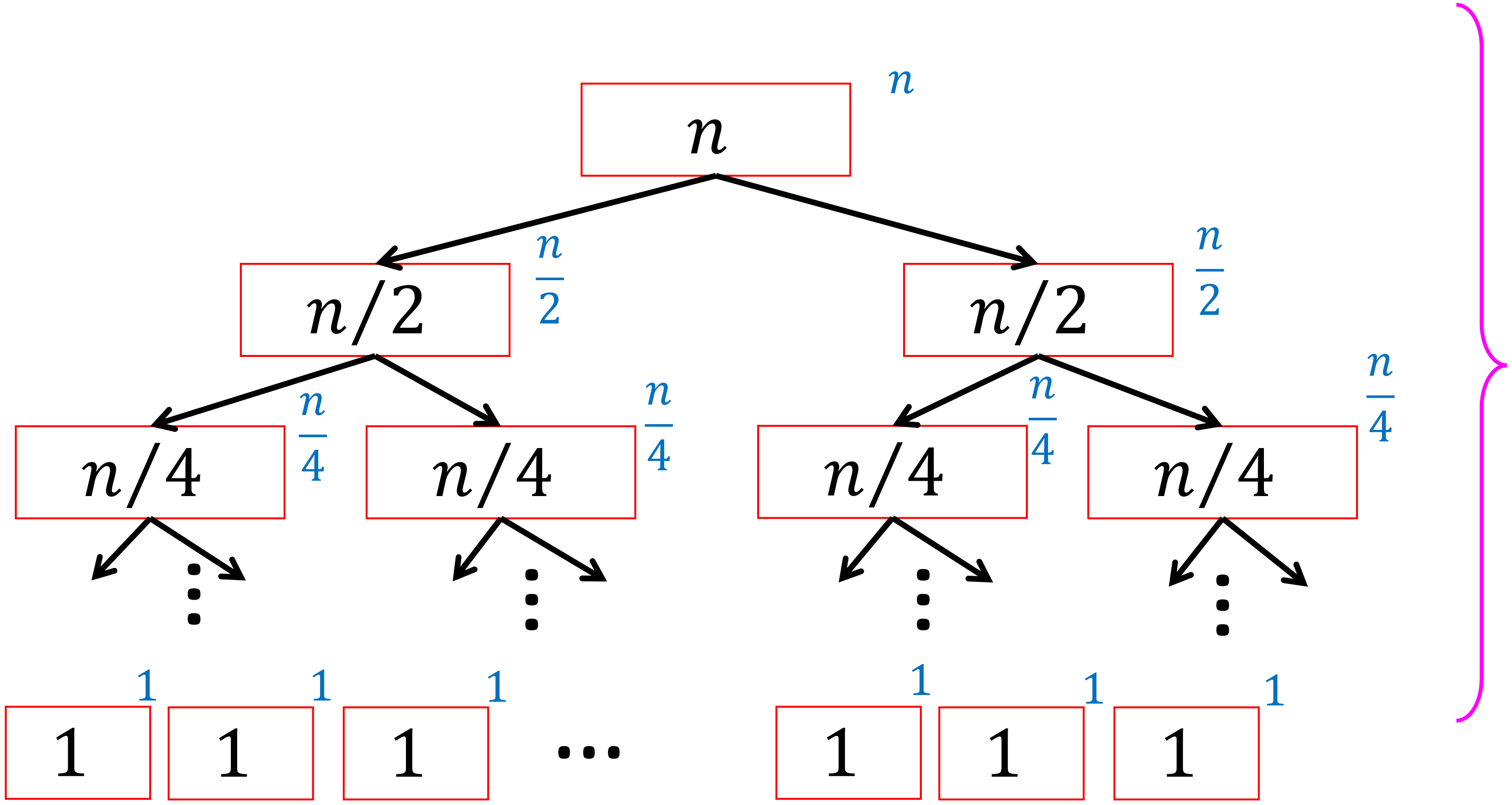
Combine:

- Merge together sorted sublists into one sorted list

Parallelizable:
Allow different processors
to sort each sublist

Merge Sort (Sequential)

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



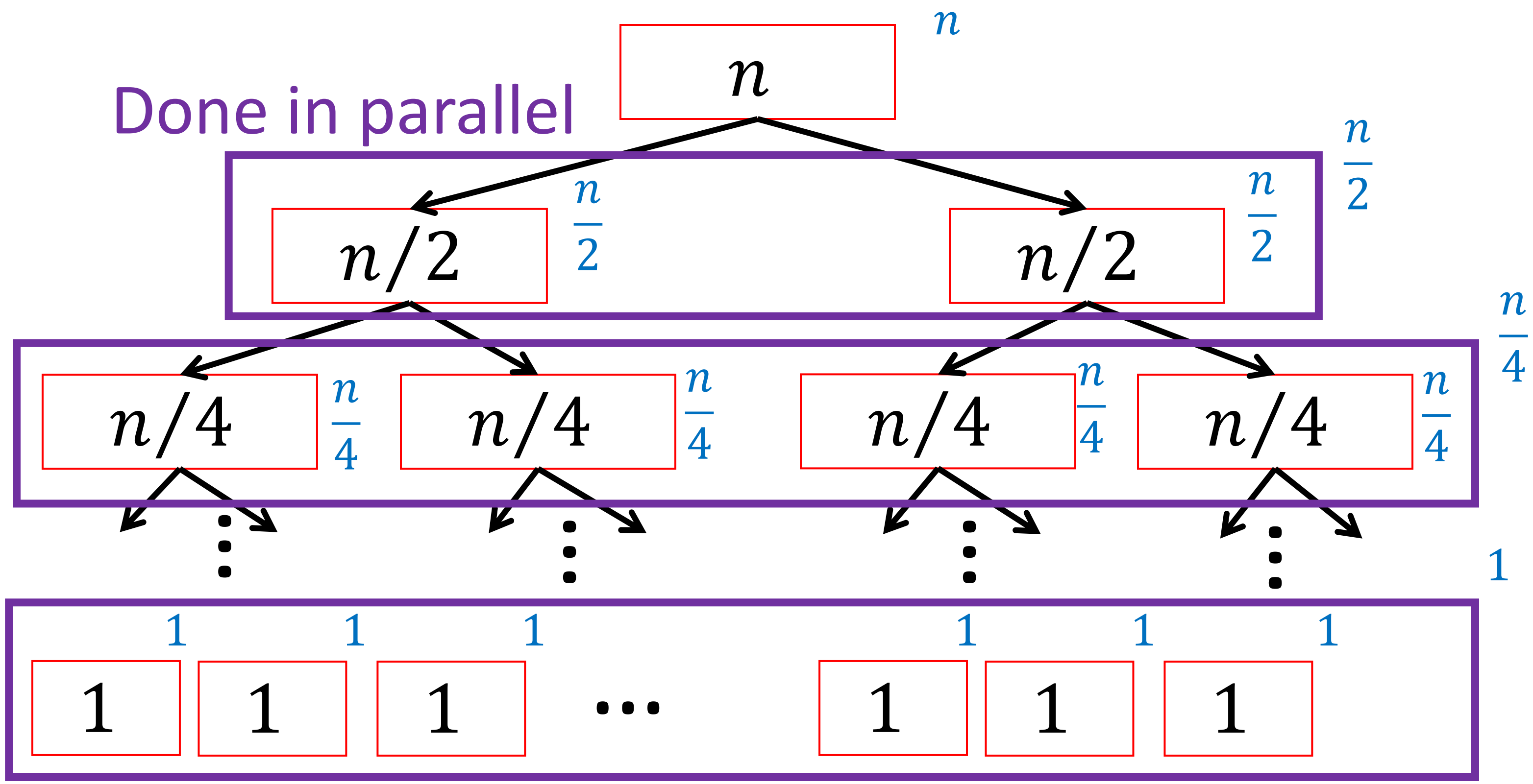
n total / level

$\log_2 n$ levels of recursion

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

Merge Sort (Parallel)

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



Run Time: $O(\log n)$

Quicksort

Divide:

- Choose random pivot p, Partition(p)

Conquer:

- Recursively sort left and right sublists

Combine:

- Nothing

Run Time?

$O(n \log n)$ on expectation

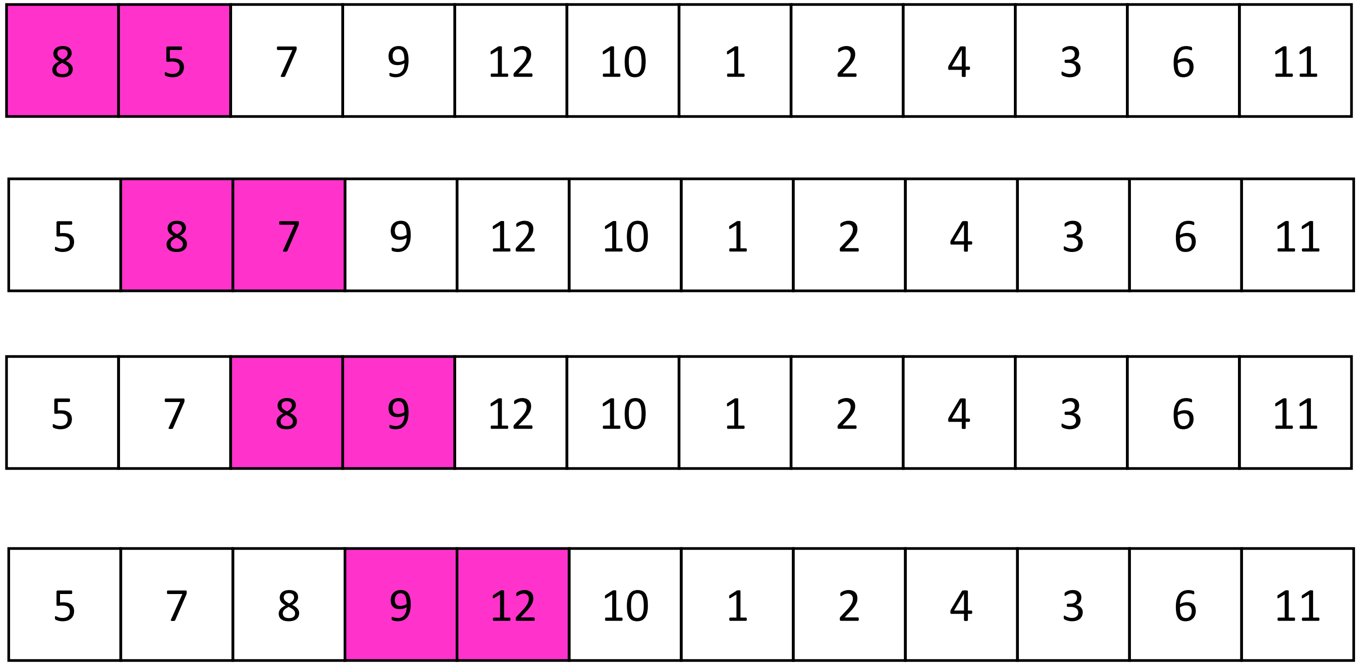
(Better constants than merge sort)

In Place?	Adaptive?	Stable?	Parallelizable?
Yes*	No	No	Yes

Can sort the list in place, but requires $\Theta(\log n)$ space on the stack (so not "in place" in a strict sense"

Bubble Sort

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

Run Time?

$O(n^2)$

(Constants worse than insertion sort)

In Place?	Adaptive?
Yes	No

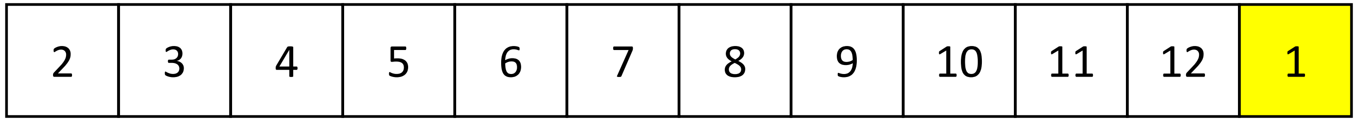
"Compared to straight insertion […], bubble sorting requires a more complicated program and takes about twice as long!" – Donald Knuth

How Adaptive is Bubble Sort?

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

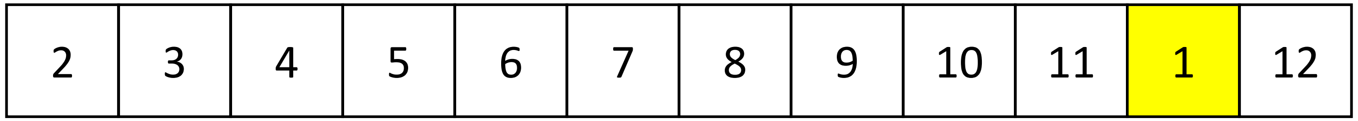
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12

Only makes one “pass” if list is already sorted



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

Still requires n passes, thus is $\Omega(n^2)$

Bubble Sort

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

Run Time?

$O(n^2)$

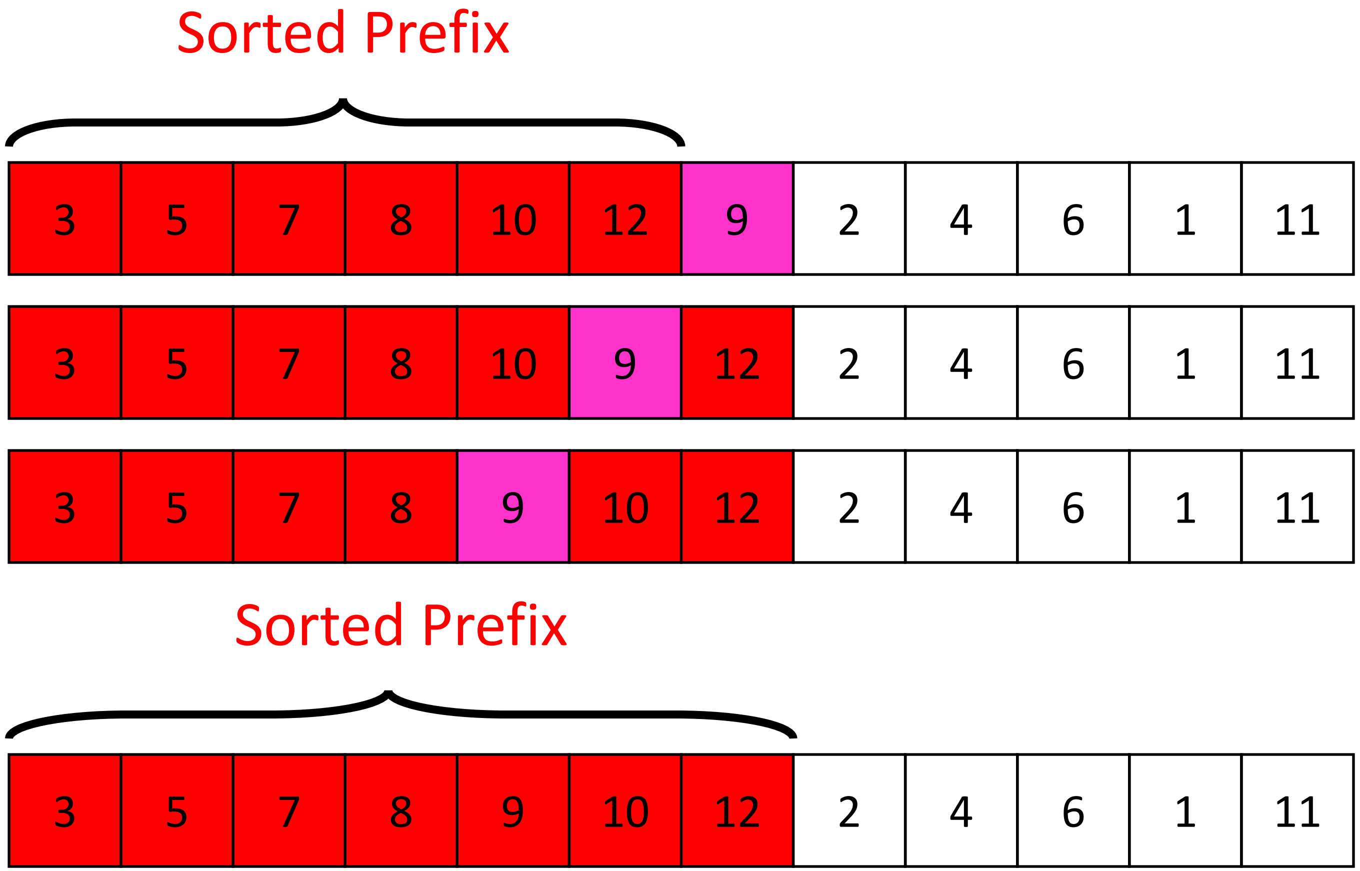
(Constants worse than insertion sort)

In Place?	Adaptive?	Stable?	Parallelizable?
Yes	No	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



Sorted Prefix

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

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

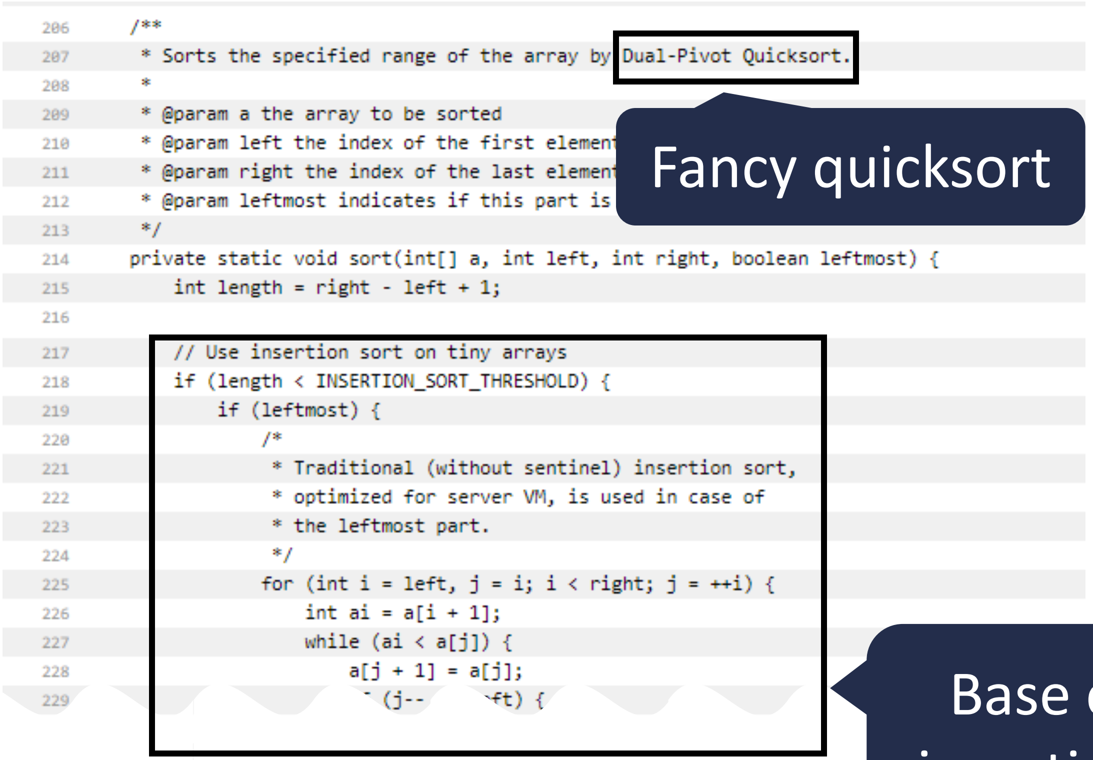
3	5	7	8	9	10	12	2	4	6	1	11

Sorted Prefix

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

Insertion Sort

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



```
/**
 * Sorts the specified range of the array by Dual-Pivot Quicksort.
 *
 * @param a the array to be sorted
 * @param left the index of the first element
 * @param right the index of the last element
 * @param leftmost indicates if this part is
 */
private static void sort(int[] a, int left, int right, boolean leftmost) {
    int length = right - left + 1;

    // Use insertion sort on tiny arrays
    if (length < INSERTION_SORT_THRESHOLD) {
        if (leftmost) {
            /*
             * Traditional (without sentinel) insertion sort,
             * optimized for server VM, is used in case of
             * the leftmost part.
             */
            for (int i = left, j = i; i < right; j = ++i) {
                int ai = a[i + 1];
                while (ai < a[j]) {
                    a[j + 1] = a[j];
                      (j--     ft) {
```

Fancy quicksort

Base case is insertion sort!

Run Time?

$$O(n^2)$$

(but with very small constants; great for short lists)

Code snippet from Java `Arrays.sort` implementation

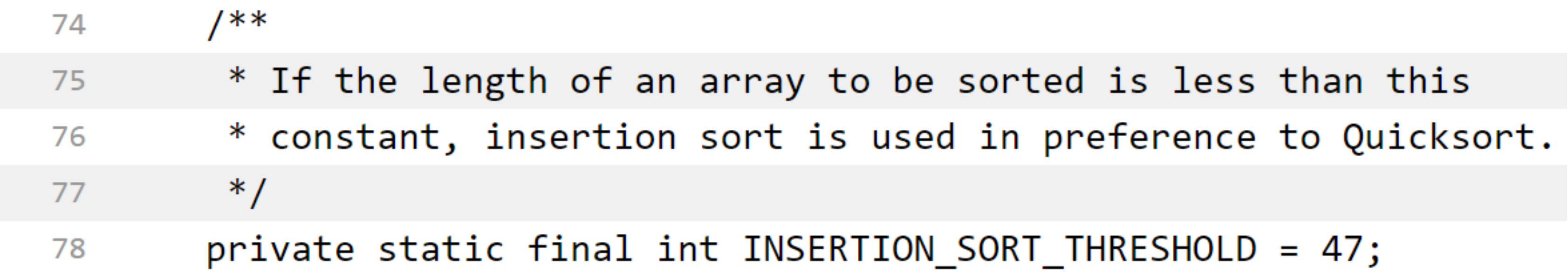
Insertion Sort

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

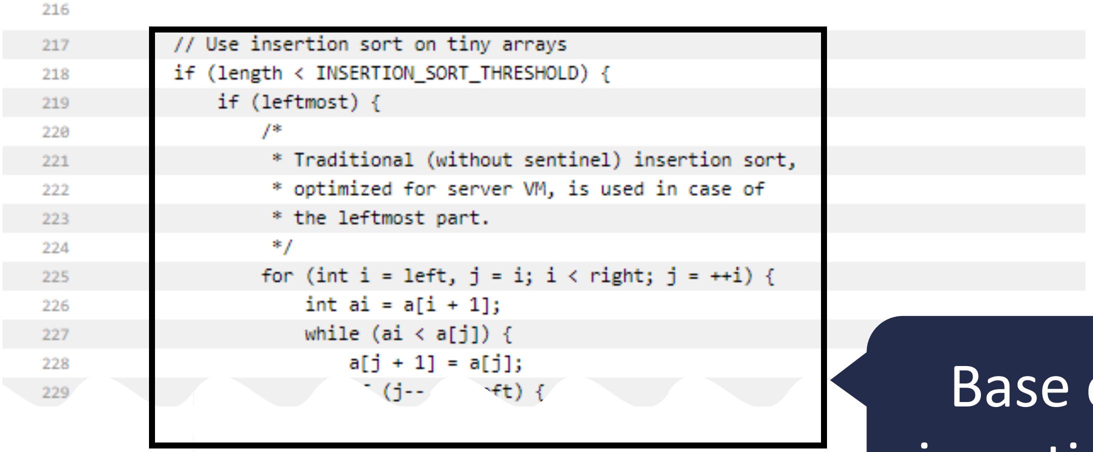
Run Time?

$O(n^2)$

(but with very small constants; great for short lists)



```
/**
 * If the length of an array to be sorted is less than this
 * constant, insertion sort is used in preference to Quicksort.
 */
private static final int INSERTION_SORT_THRESHOLD = 47;
```



```
// Use insertion sort on tiny arrays
if (length < INSERTION_SORT_THRESHOLD) {
    if (leftmost) {
        /*
         * Traditional (without sentinel) insertion sort,
         * optimized for server VM, is used in case of
         * the leftmost part.
         */
        for (int i = left, j = i; i < right; j = ++i) {
            int ai = a[i + 1];
            while (ai < a[j]) {
                a[j + 1] = a[j];
                  (j--       ft) {
```

Code snippet from Java `Arrays.sort` implementation

Base case is insertion sort!

Insertion Sort

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

Run Time?

$O(n^2)$

(but with very small constants; great for short lists)

In Place?

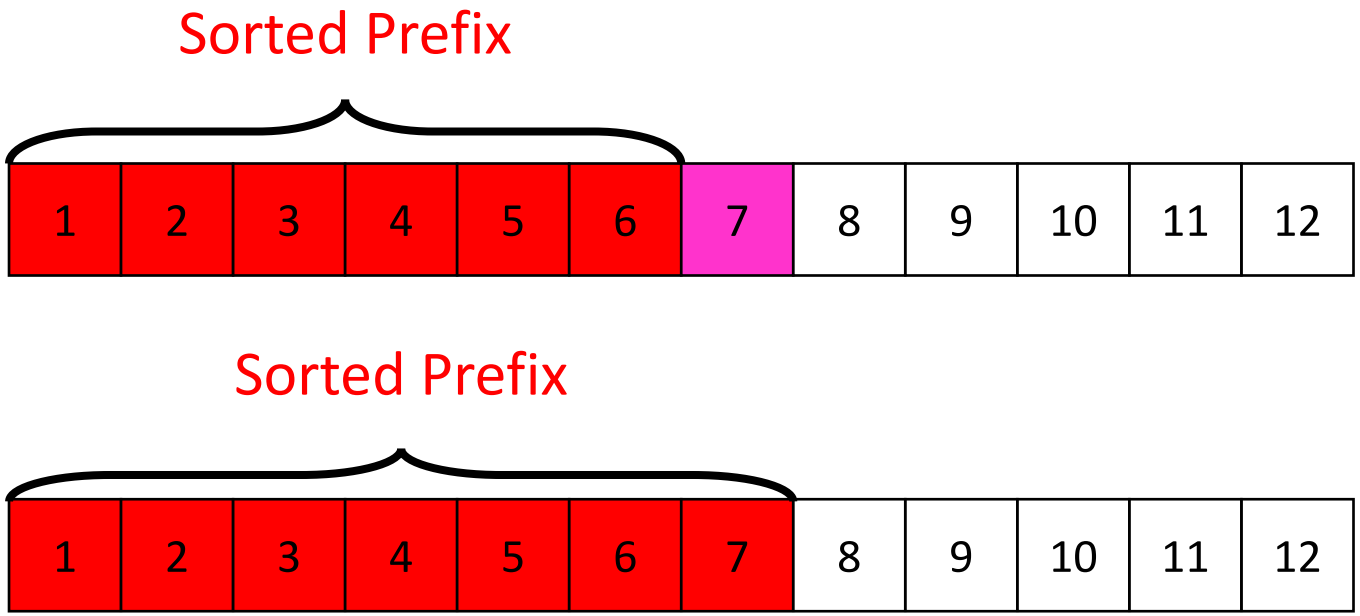
Yes

Adaptive?

Yes

Insertion Sort is Adaptive

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



Sorted Prefix

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

Sorted Prefix

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

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?

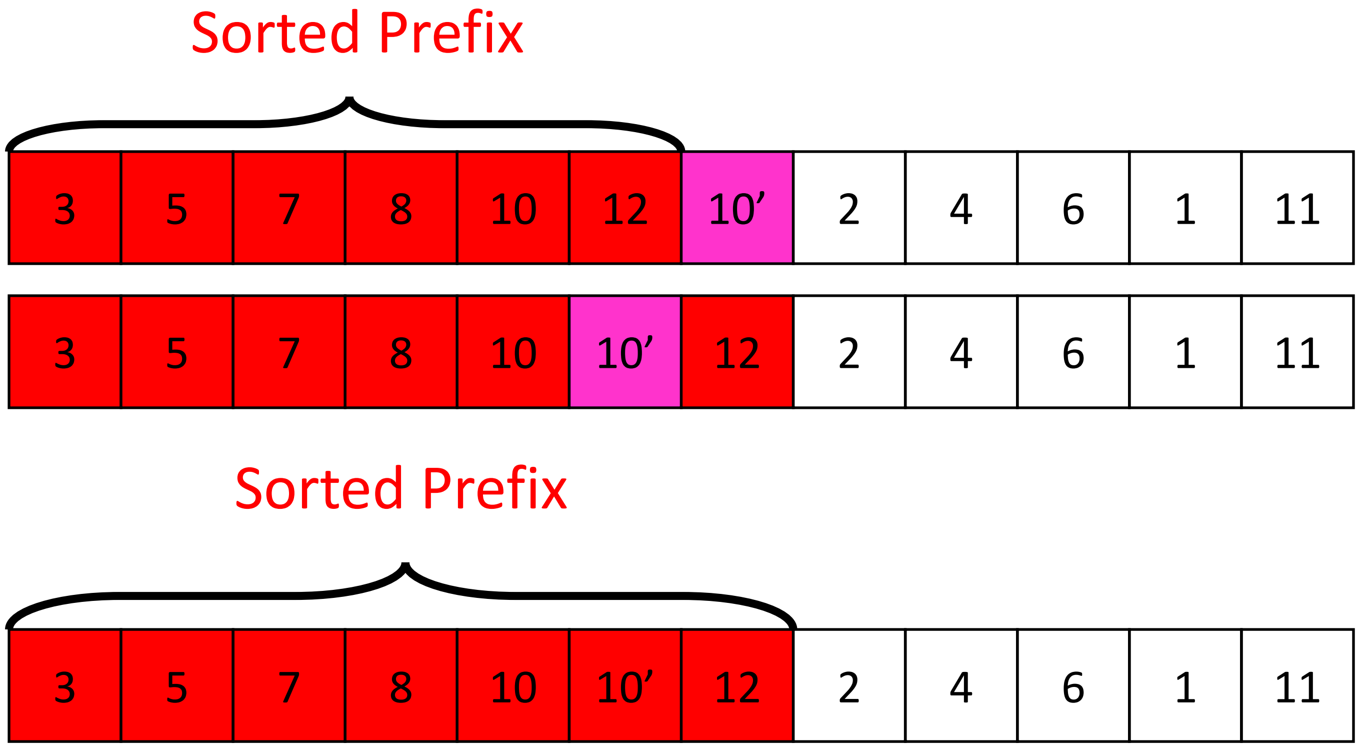
$O(n^2)$

(but with very small constants; great for short lists)

In Place?	Adaptive?	Stable?
Yes	Yes	Yes

Insertion Sort is Stable

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



Sorted Prefix

3	5	7	8	10	12	10'	2	4	6	1	11

3	5	7	8	10	10'	12	2	4	6	1	11

Sorted Prefix

3	5	7	8	10	10'	12	2	4	6	1	11

Observation: 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?

$O(n^2)$

(but with very small constants; great for short lists)

In Place?	Adaptive?	Stable?	Parallelizable?
Yes	Yes	Yes	No

Online: Can sort a list as it is received (e.g., streamed from the network); we do not require the entire list to begin sorting

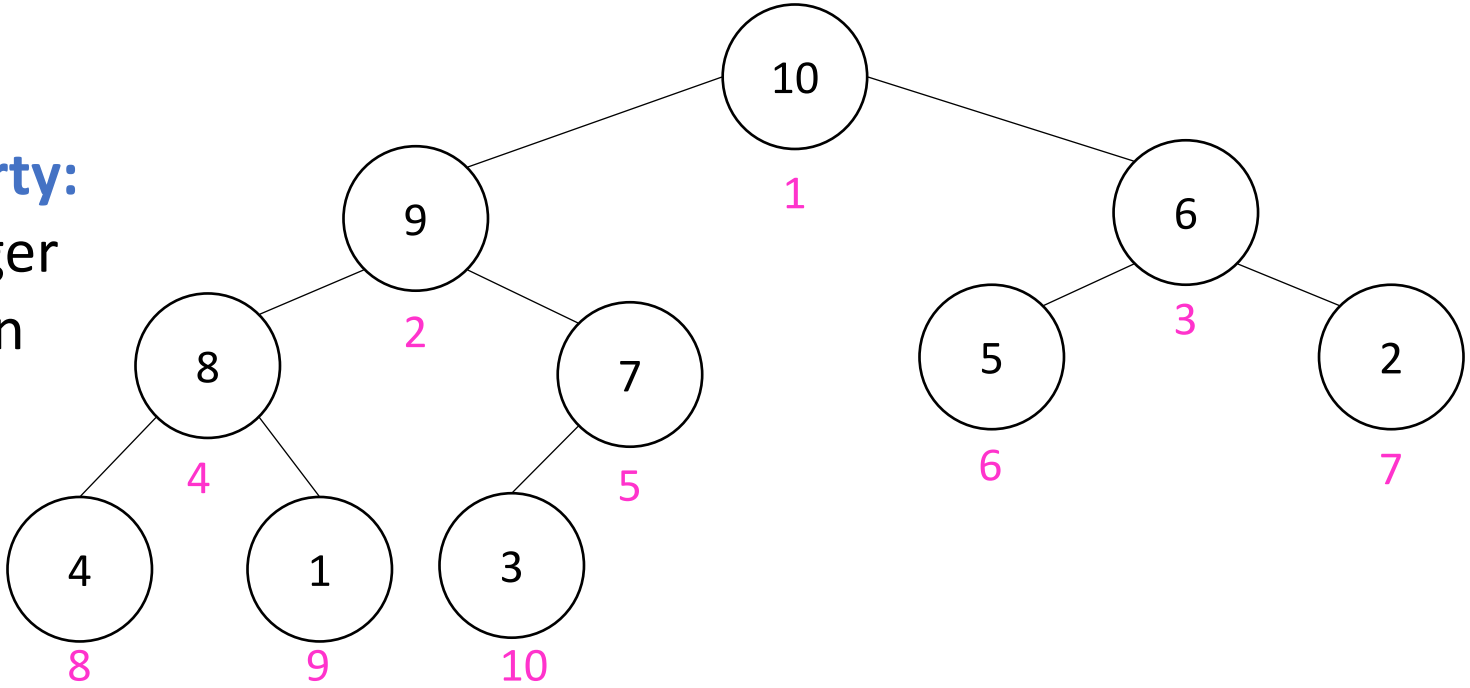
Online?
Yes

Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build a sorted list (form 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

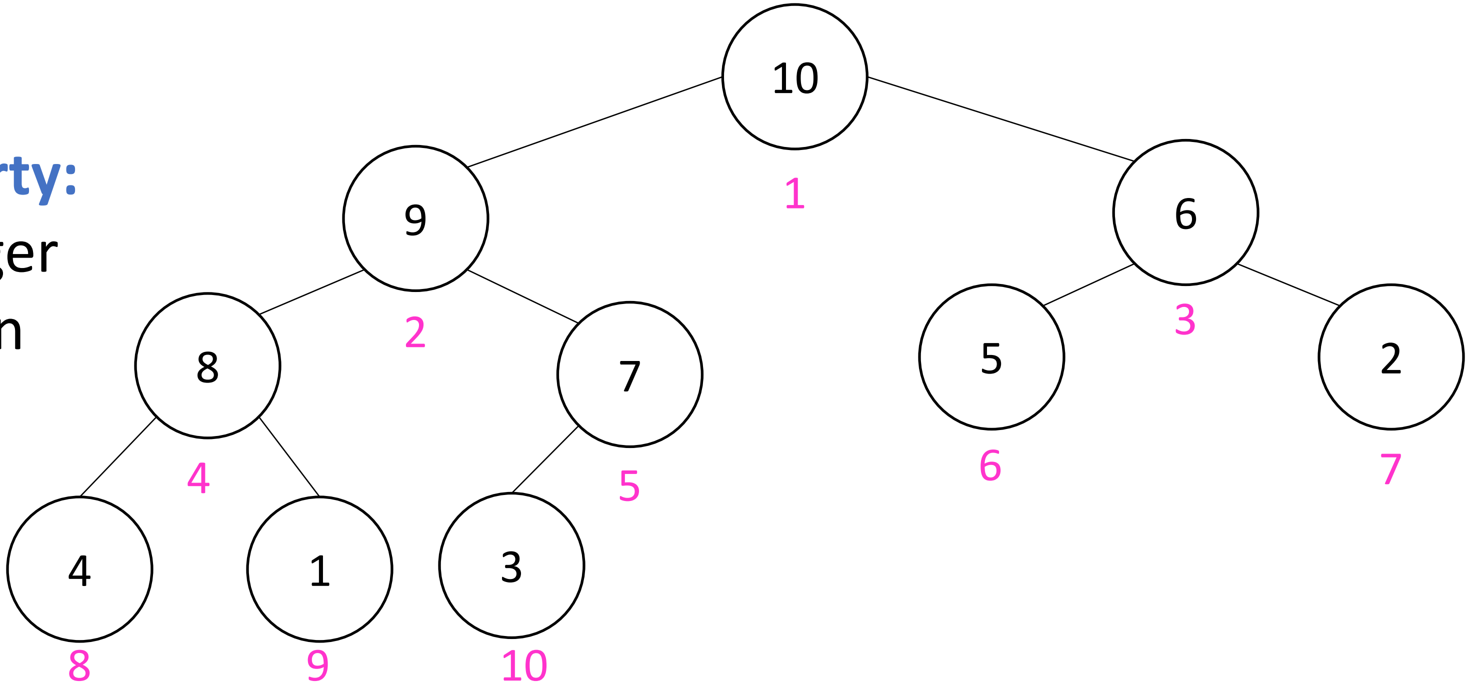


Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

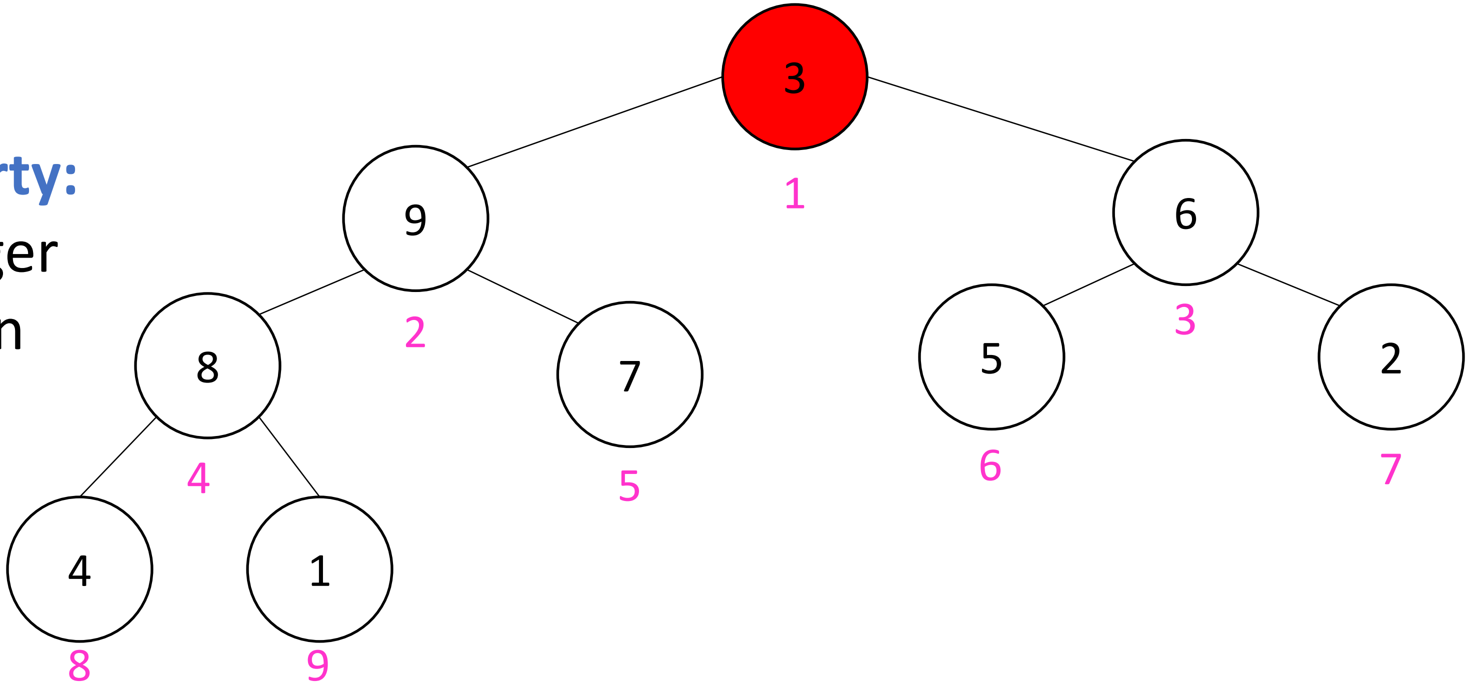
Max heap property:
Each node is larger than its children



Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

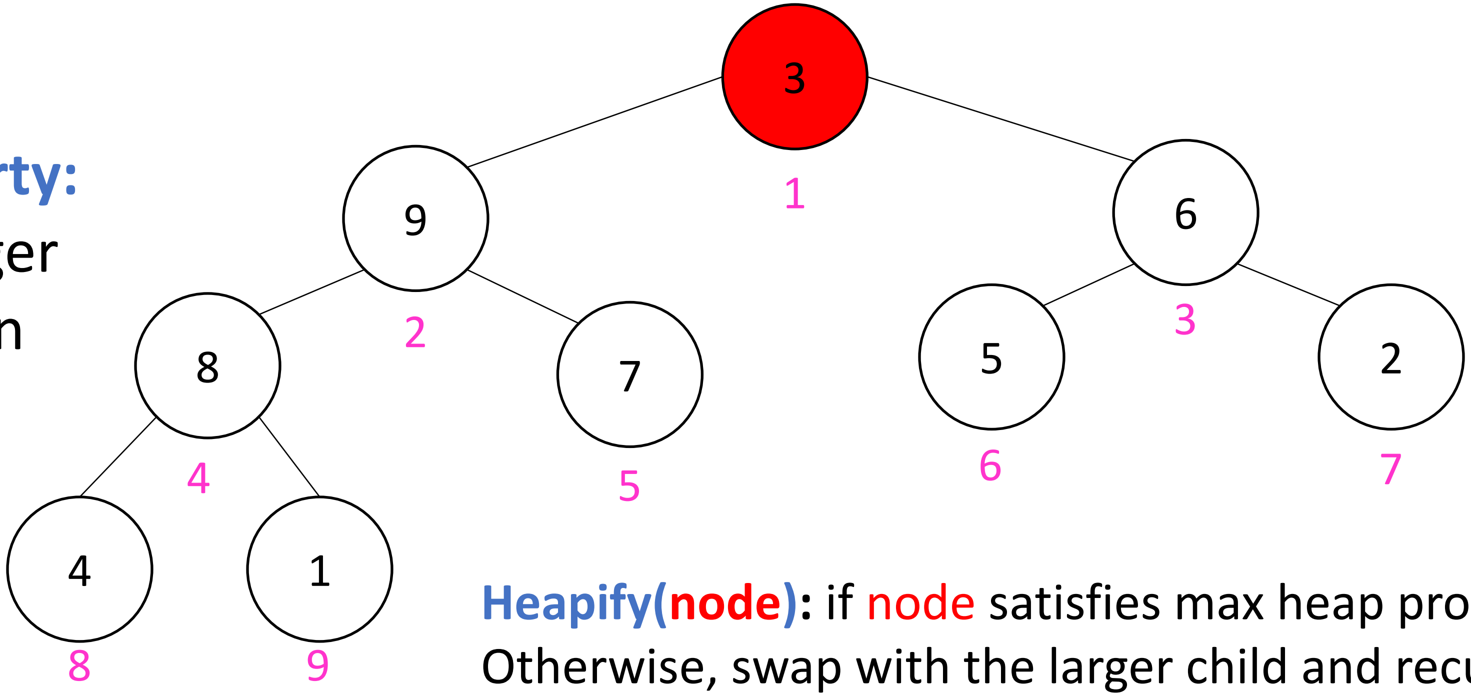


Max heap property:
Each node is larger than its children

Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

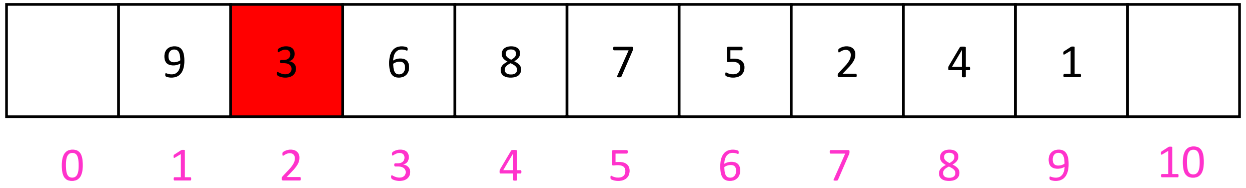


Max heap property:
Each node is larger than its children

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

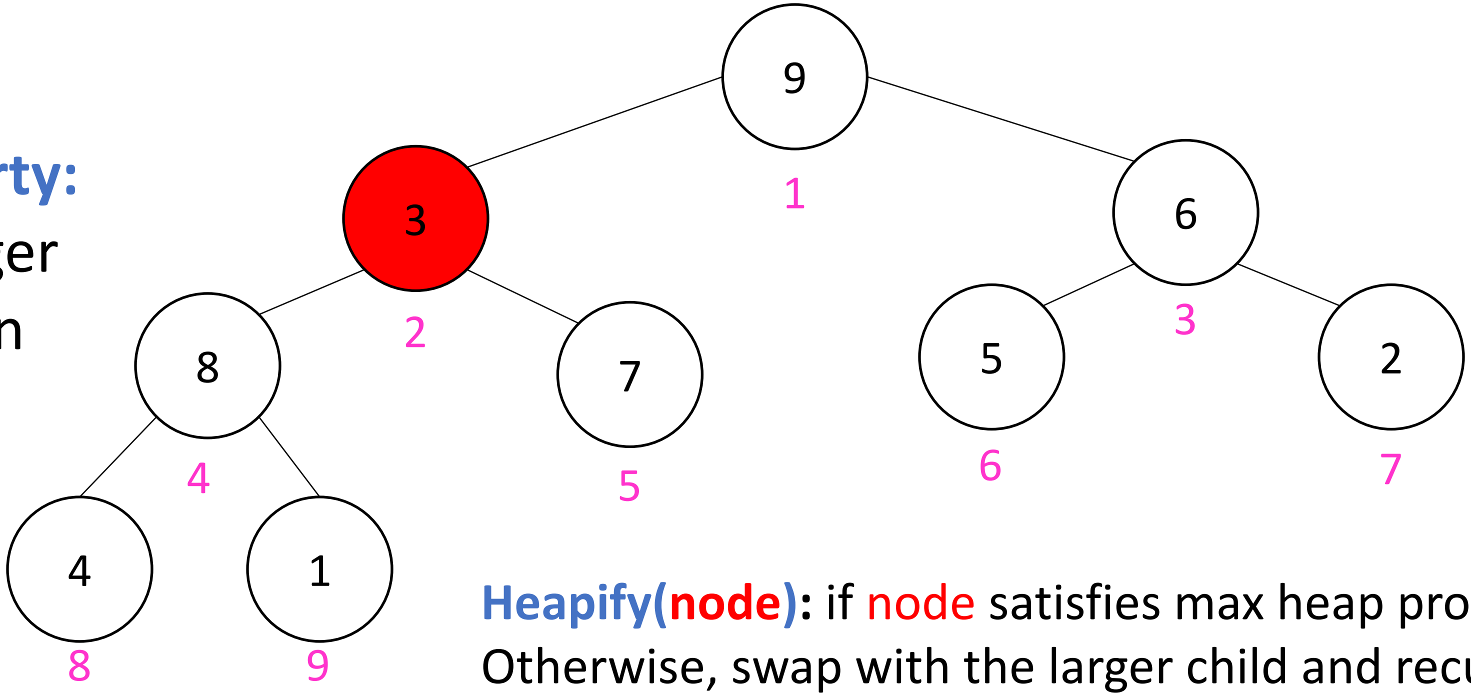
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)



	9	3	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



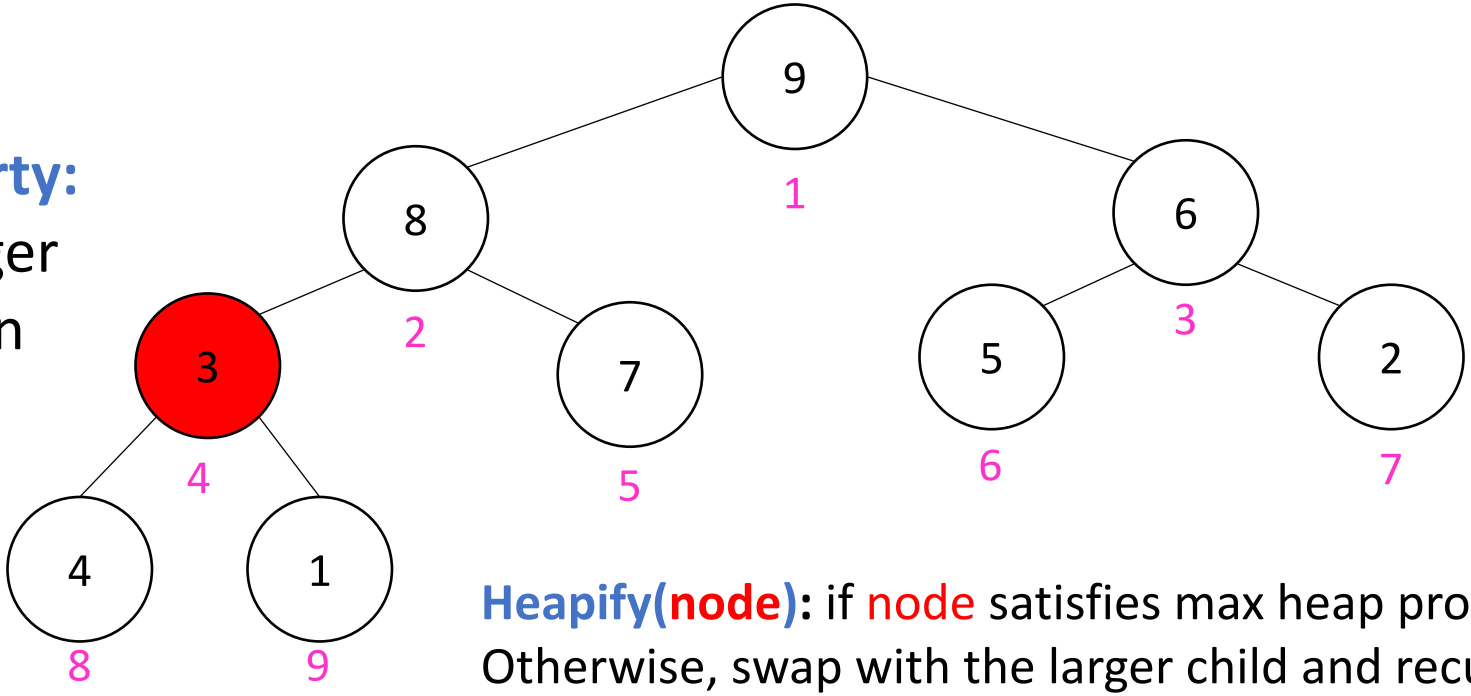
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

	9	8	6	3	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



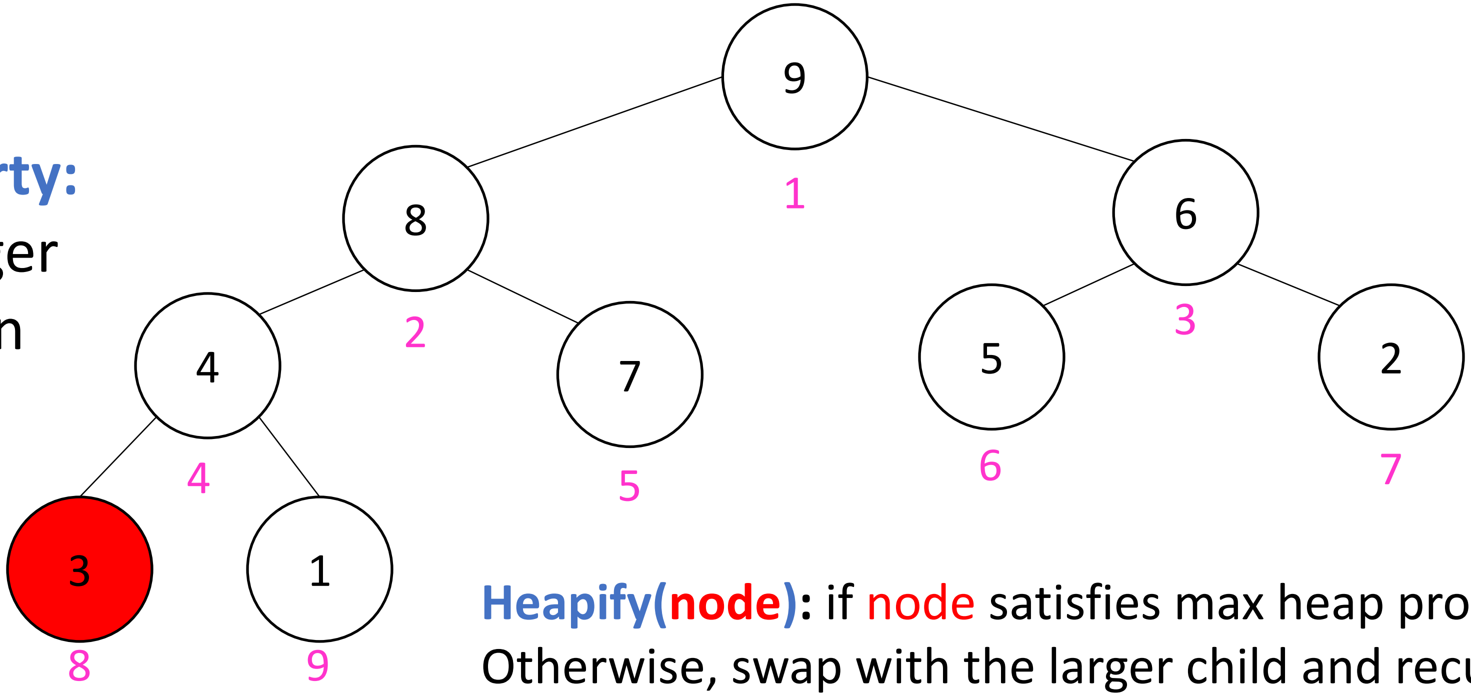
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling 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



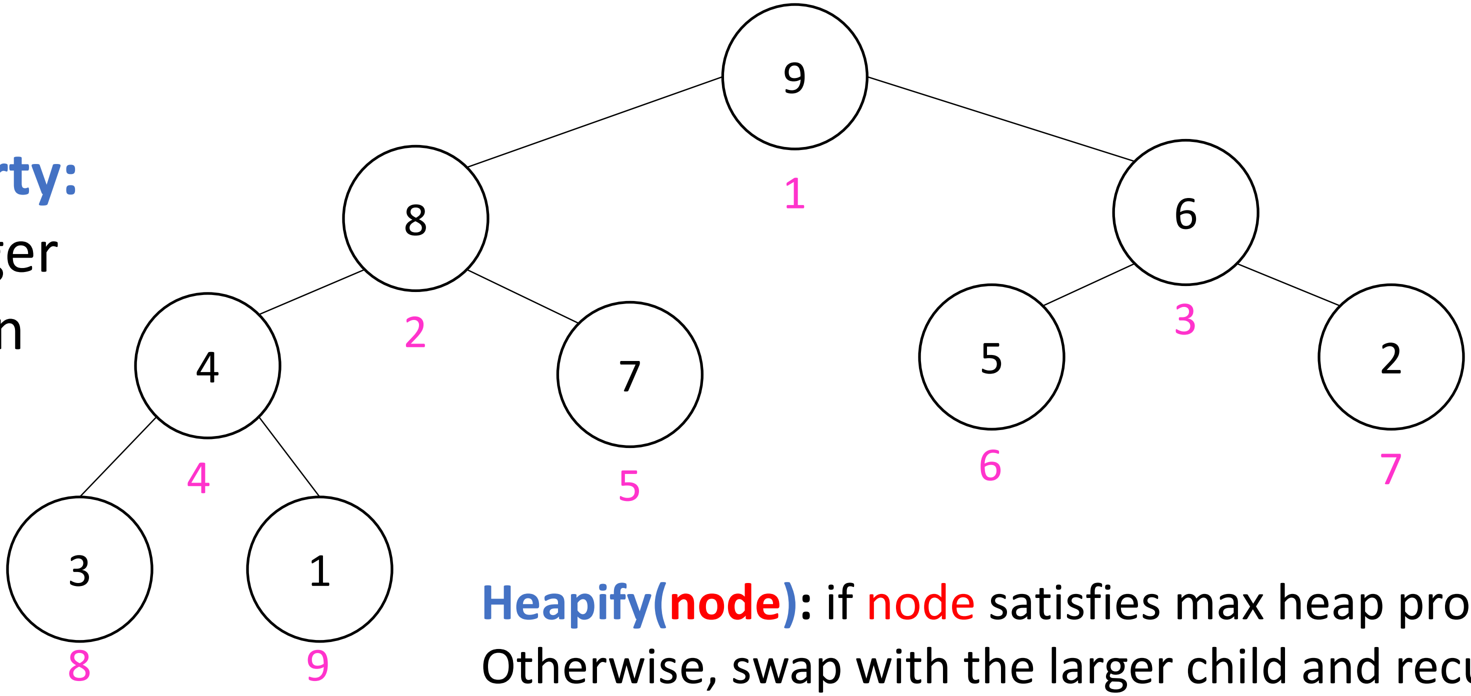
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling 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



Running time: $O(\log n)$

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

Running time:

- Constructing heap by calling Heapify on each node in tree (bottom up): $O(n \log n)$
- Extracting maximum element to sort list: $O(n \log n)$

Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

In Place?

Yes

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

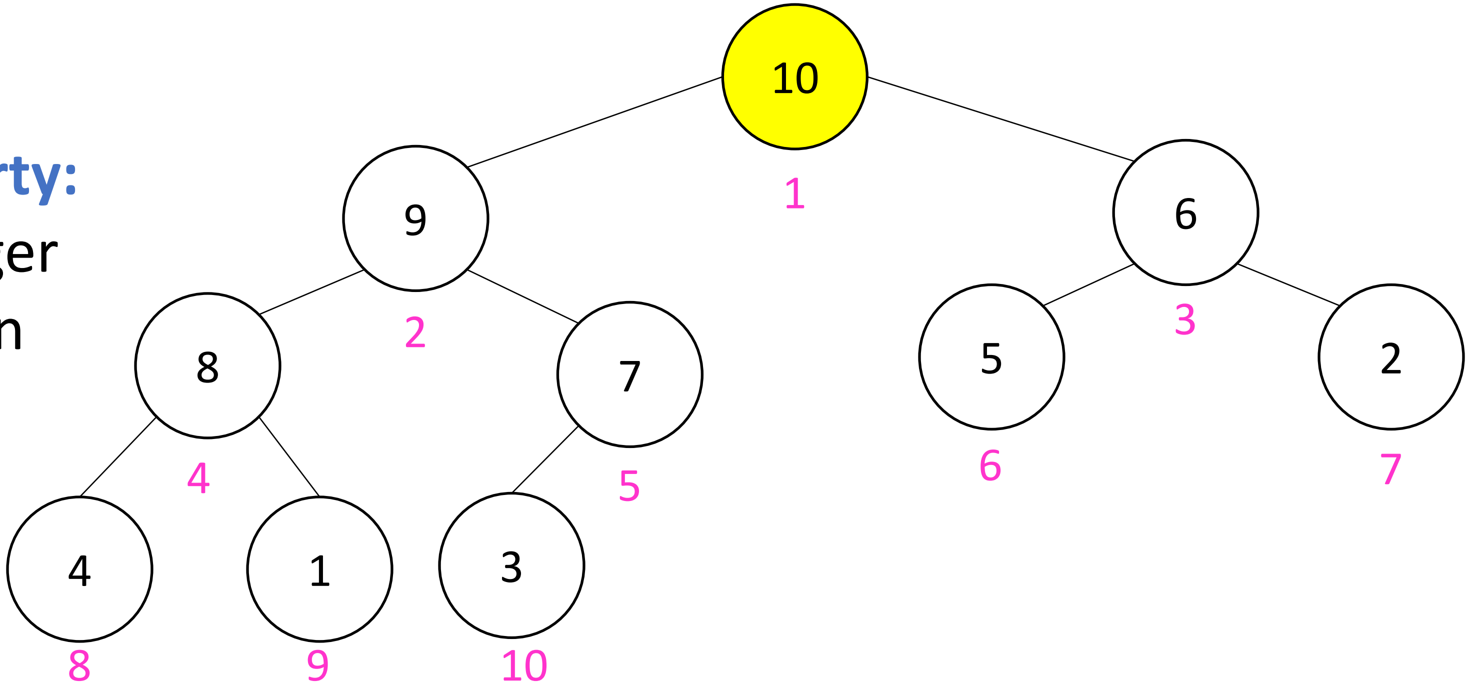
Constructing heap is also in-place (just requires calling Heapify)

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

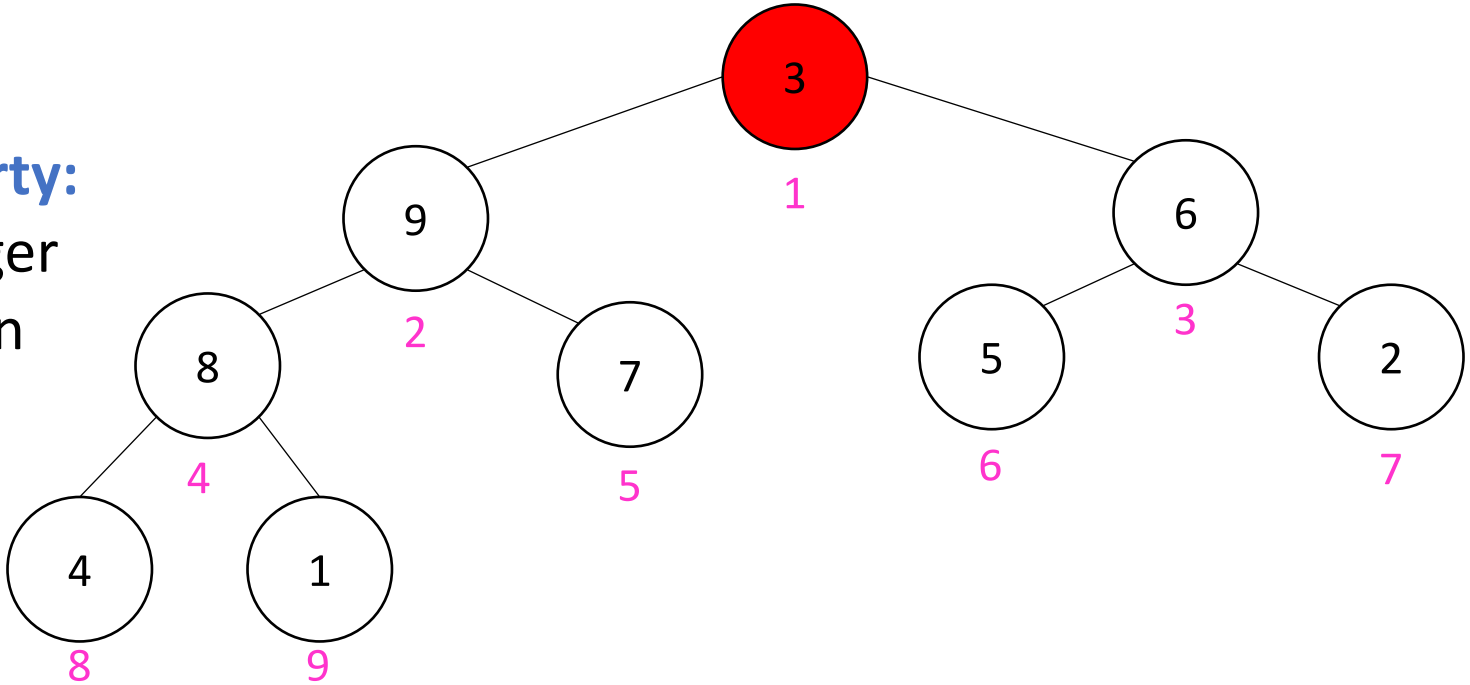


In-Place Heap Sort

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

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

Max heap property:
Each node is larger than its children

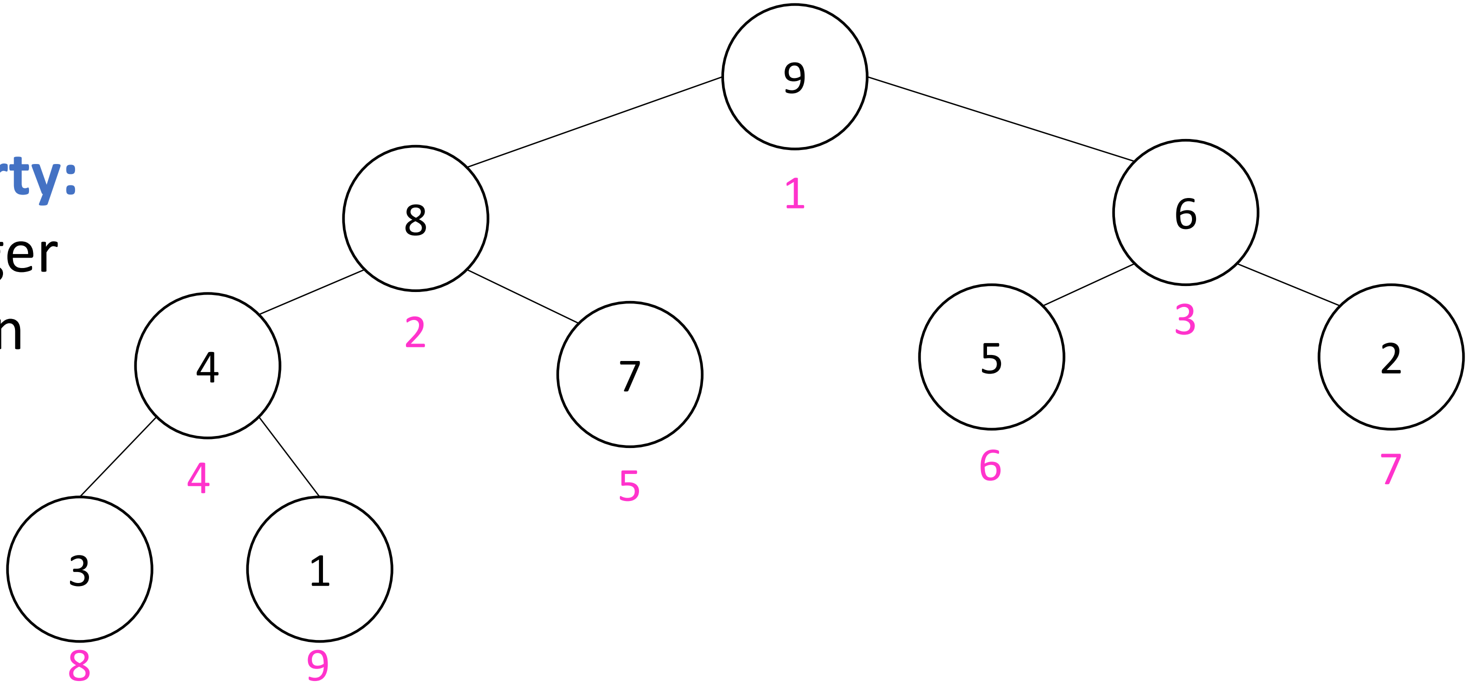


In-Place Heap Sort

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

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

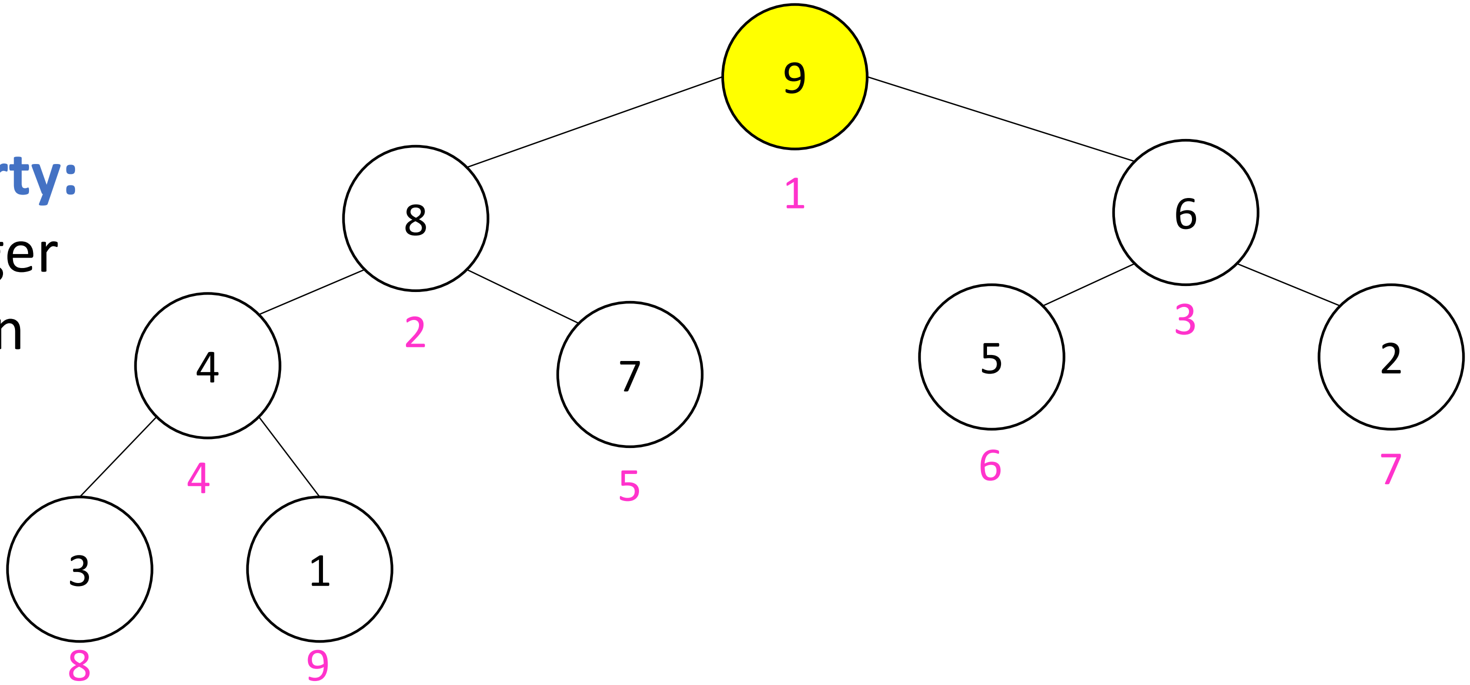
Max heap property:
Each node is larger than its children



In-Place Heap Sort

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

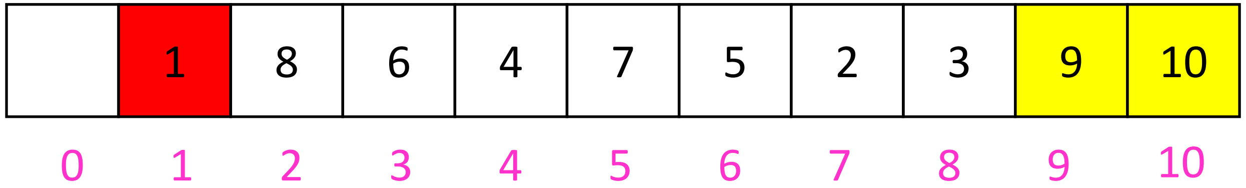
	9	8	6	4	7	5	2	3	1	10
0	1	2	3	4	5	6	7	8	9	10



Max heap property:
Each node is larger than its children

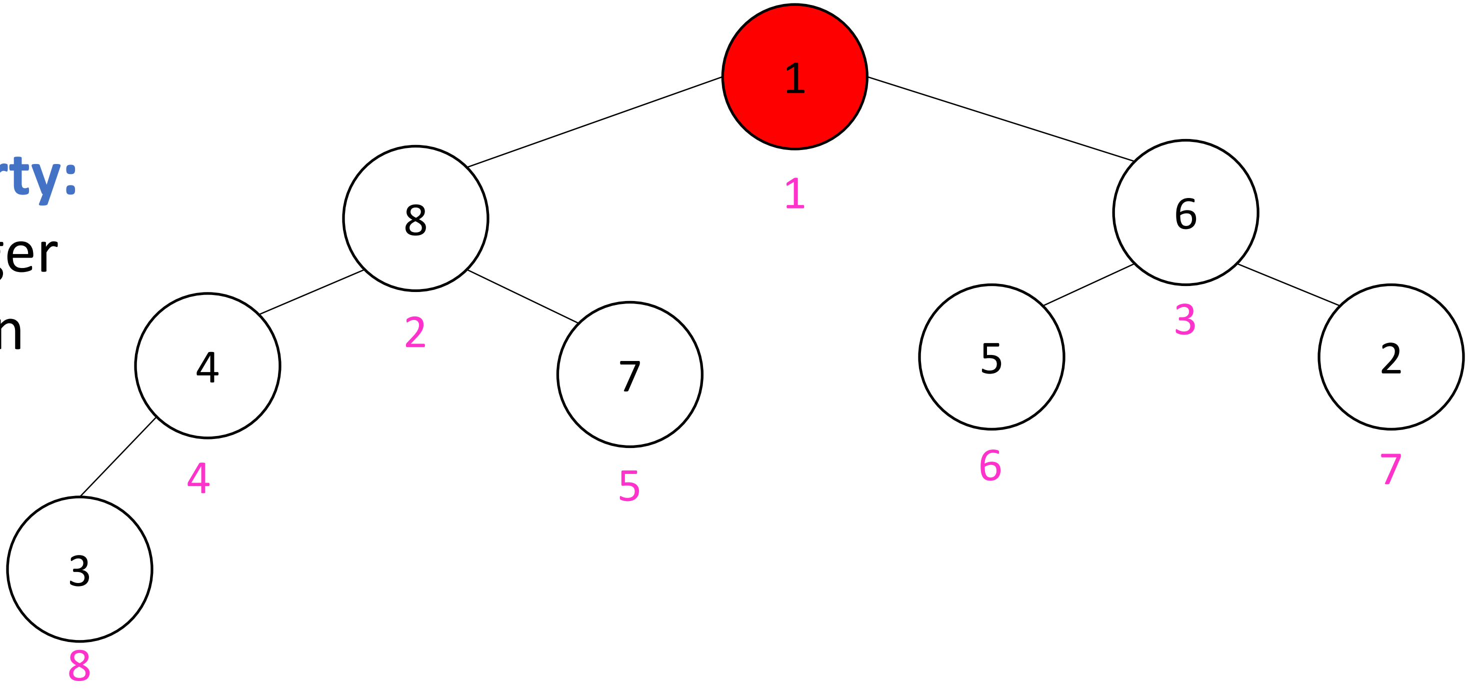
In-Place Heap Sort

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



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

Max heap property:
Each node is larger than its children

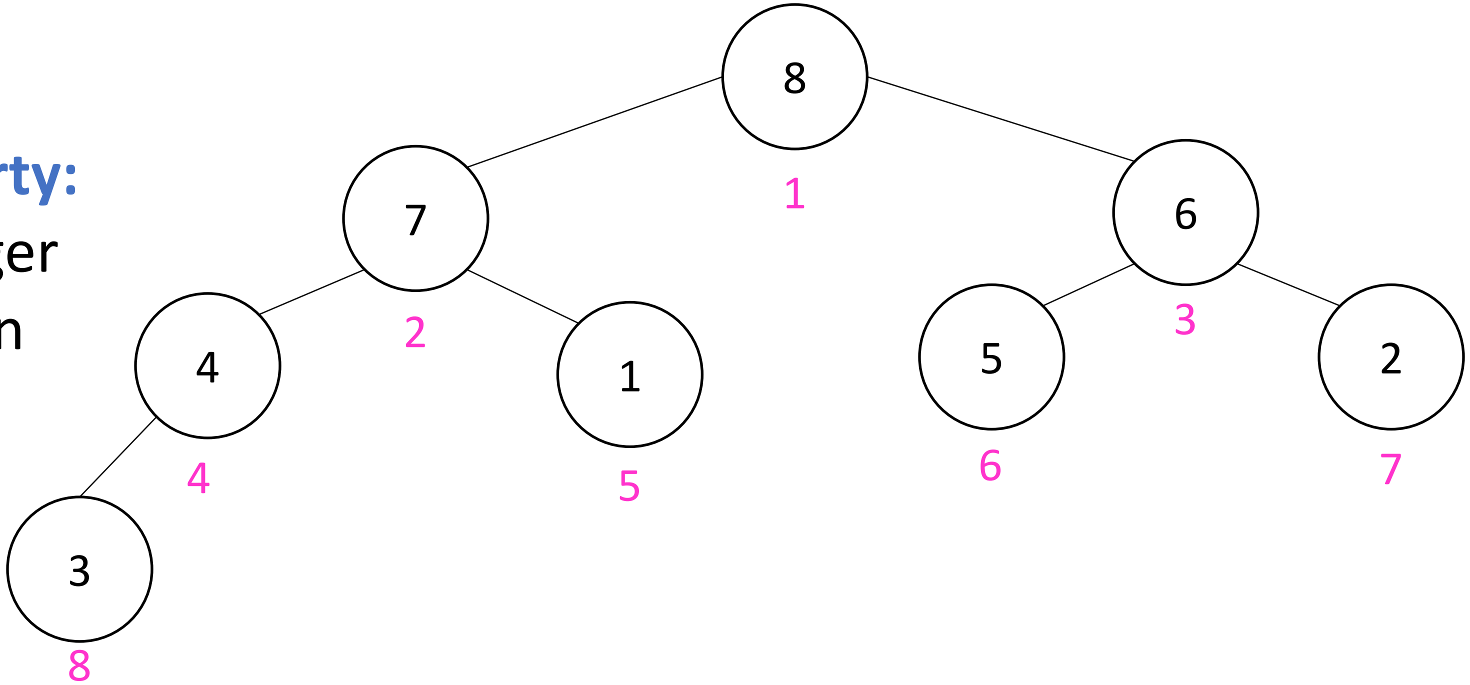


In-Place Heap Sort

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

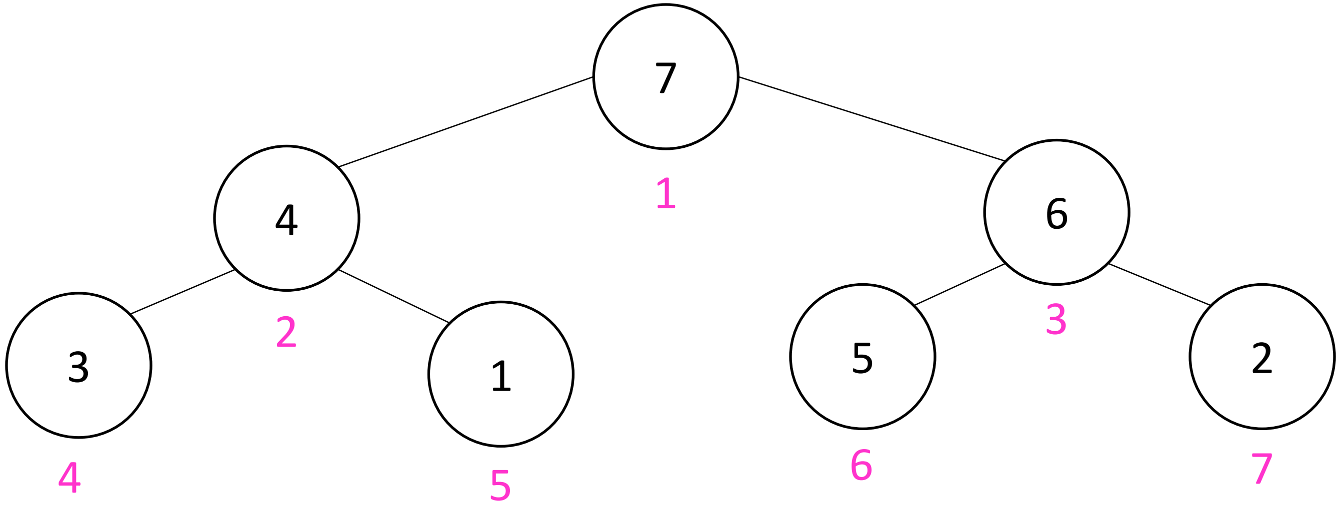


In-Place Heap Sort

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

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

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 (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

In Place?	Adaptive?	Stable?	Parallelizable?
Yes	No	No	No

Sorting Algorithms

Sorting algorithms we have discussed:

- Mergesort $O(n \log n)$
- Quicksort $O(n \log n)$

Other sorting algorithms (will discuss):

- Bubble sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heapsort $O(n \log n)$

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