CS4102 Algorithms

Fall 2019

Warm up

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

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

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

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

$$\begin{array}{cccccccccc} n! = & n & \cdot & (n-1) & \cdot & (n-2) & \cdot \ldots \cdot & 2 & \cdot & 1 \\ & \shortparallel & & \wedge & & \wedge & & \wedge & & \wedge \\ n^n = & n & \cdot & n & \cdot & n & \cdot \ldots \cdot & n & \cdot & n \end{array}$$

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

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

$$\begin{array}{ccccccccccccccccc}
n! & = & n & \cdot & (n-1) & \cdot & (n-2) & \cdot\ \ldots\ \cdot & \frac{n}{2} & \cdot & \left(\frac{n}{2}-1\right) & \cdot\ \ldots\ \cdot & 2 & \cdot & 1 \\
 & & \vee & & \vee & & \vee & & \parallel & & \vee & & \vee & & \parallel \\
\left(\frac{n}{2}\right)^{\frac{n}{2}} & = & \frac{n}{2} & \cdot & \frac{n}{2} & \cdot & \frac{n}{2} & \cdot\ \ldots\ \cdot & \frac{n}{2} & \cdot & 1 & \cdot\ \ldots\ \cdot & 1 & \cdot & 1
\end{array}$$

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

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

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

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

Today’s Keywords

- Divide and Conquer
- Quicksort
- Decision Tree
- Worst case lower bound
- Sorting

CLRS Readings

- Chapter 7
- Chapter 8

Homeworks

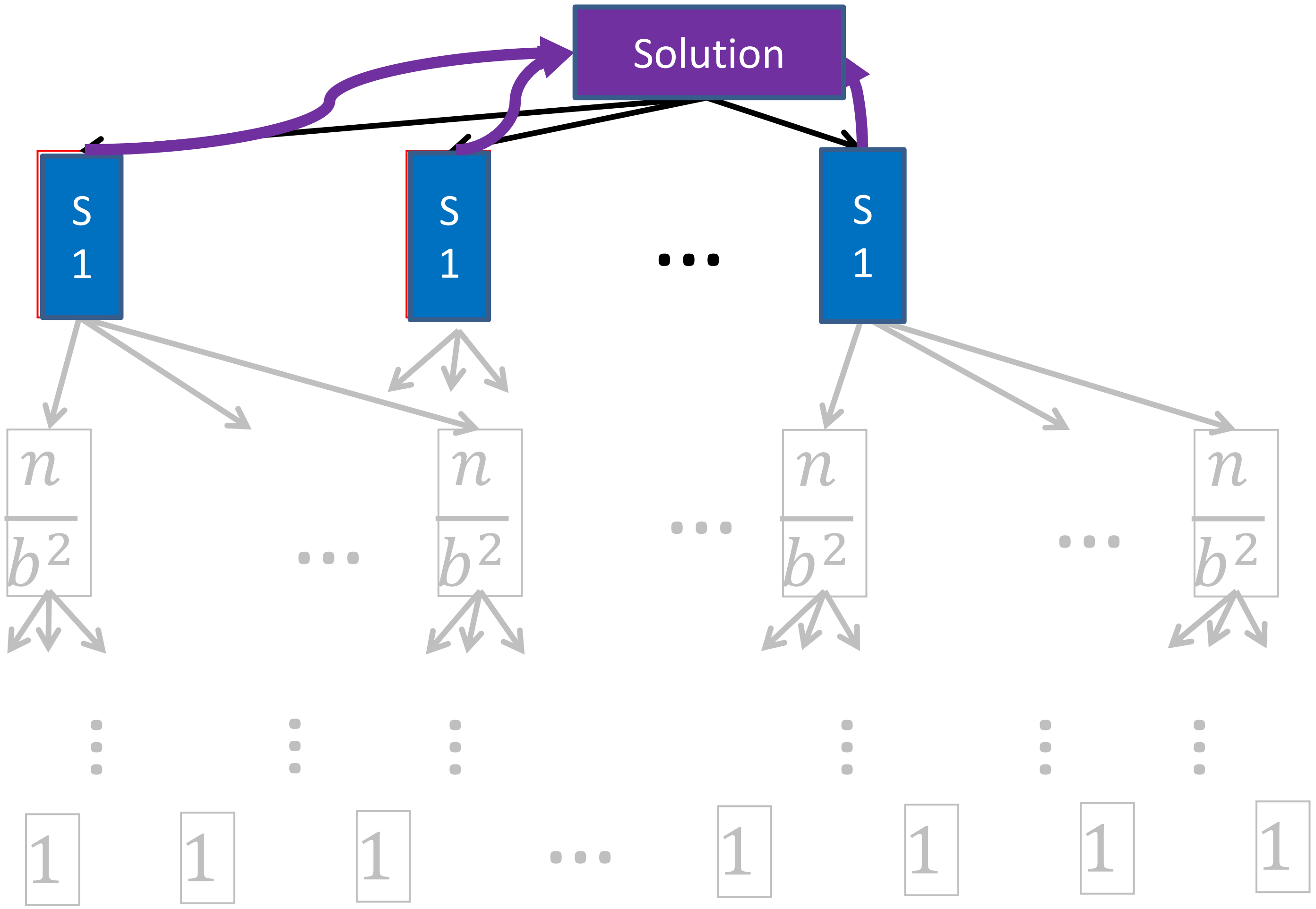
- HW3 due 11pm Tuesday, October 1
  - Divide and conquer
  - Written (use LaTeX!)
  - Submit BOTH a pdf and a zip file (2 separate attachments)
- Regrade Office Hours
  - Thursdays 11am-12pm
  - Thursdays 4pm-5pm

Aside: Divide and Conquer

Generic Divide and Conquer Solution

```
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) #brute force if necessary
        return solution
    subproblems[] = Divide(problem)
    for subproblem in subproblems:
        subsolutions.append(myDCalgo(subproblem))
    solution = Combine(subsolutions)
    return solution
```

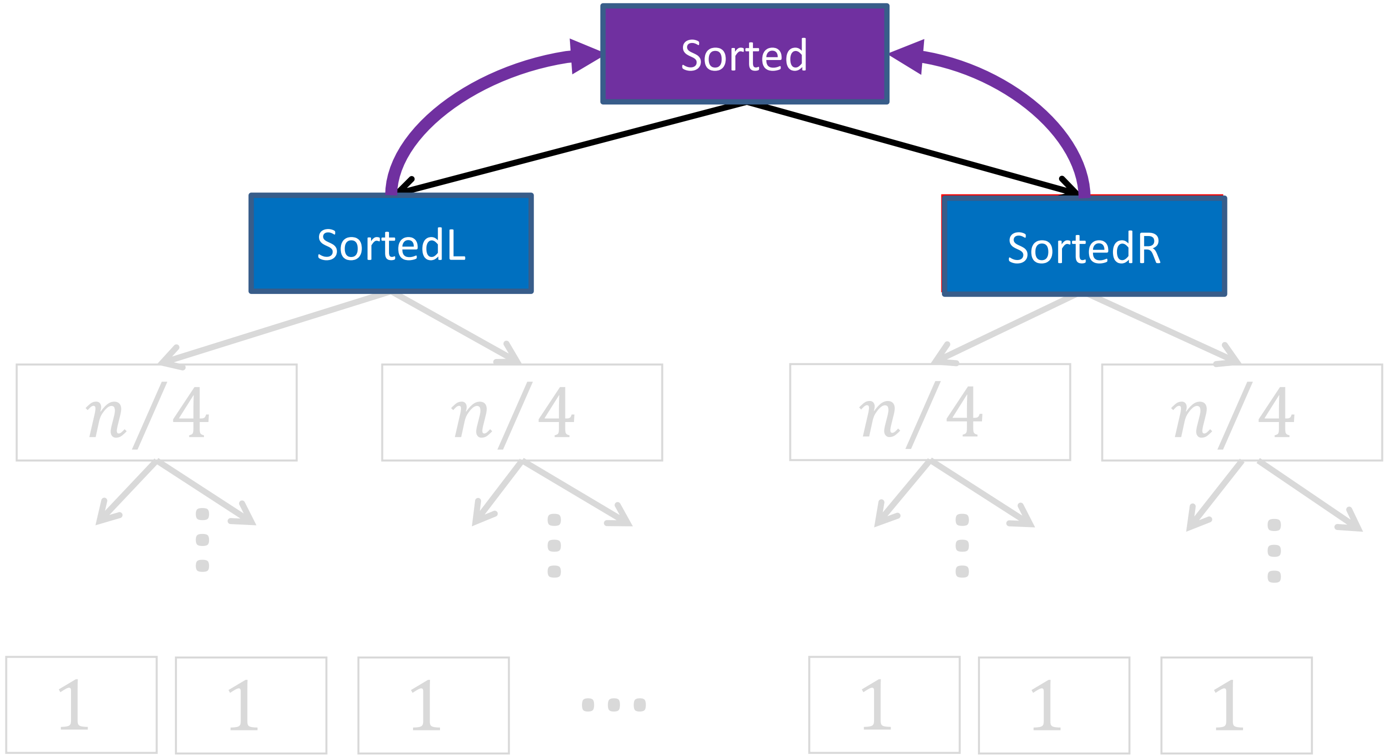
Generic Divide and Conquer Solution



MergeSort Divide and Conquer Solution

```
def mergesort(list):
    if list.length < 2:
        return list  #list of size 1 is sorted!
    {listL, listR} = Divide_by_median(list)
    for list in {listL, listR}:
        sortedSubLists.append(mergesort(list))
    solution = merge(sortedL, sortedR)
    return solution
```

MergeSort Divide and Conquer Solution



Back to Sorting!

Quicksort

- Idea: pick a pivot element, recursively sort two sublists around that element
- Divide: select an element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

Random Pivot

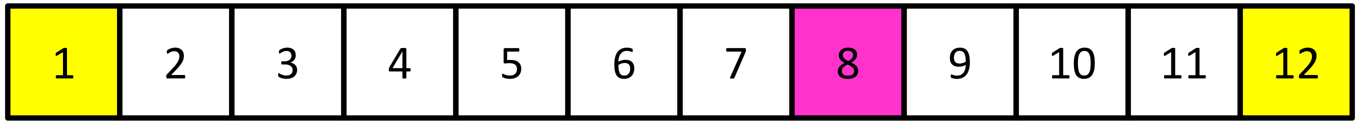
- Using Quickselect to pick median guarantees $\Theta(n \log n)$ run time
  - Approach has very large constants
  - If you really want $\Theta(n \log n)$, better off using MergeSort
- Better approach: Random pivot
  - Very small constant (very fast algorithm)
  - Expected to run in $\Theta(n \log n)$ time
    - Why? Unbalanced partitions are very unlikely

Formal Argument for $n \log n$ Average

- Remember, run time counts comparisons!
- Quicksort only compares against a pivot
  - Element i only compared to element j if one of them was the pivot

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

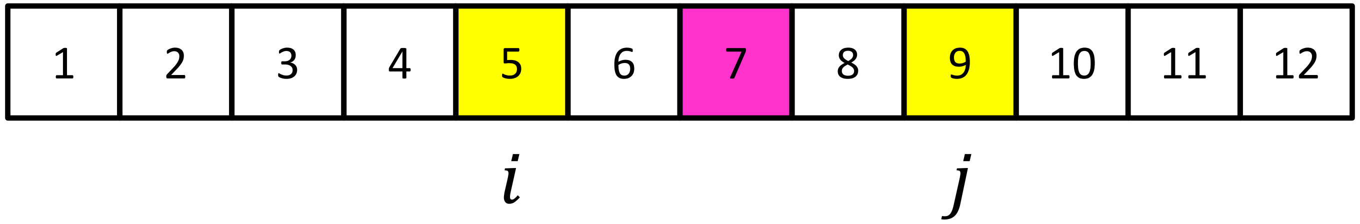
$$\Pr[\text{we compare 1 and 12}] = \frac{2}{12}$$

Assuming pivot is chosen uniformly at random

Only compared if 1 or 12 was chosen as the first pivot since otherwise they are in <u>different</u> 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			

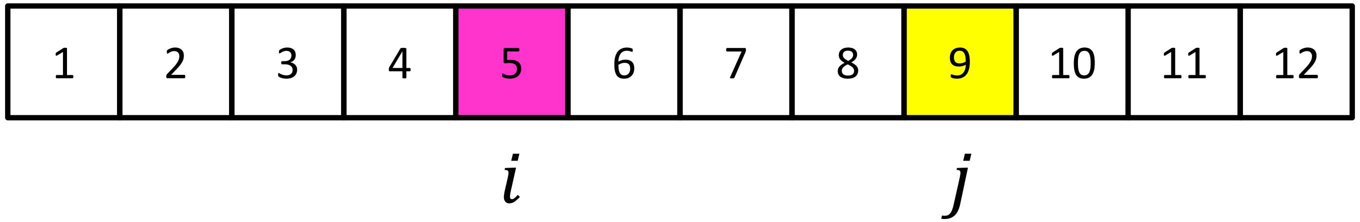
Case 3.1: Pivot contained in $[i+1, \ldots, j-1]$

Then i and j are in different sublists and will never be compared

$$\Pr[\text{we compare } i \text{ and } j] = 0$$

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 3.2: Pivot is either i or j

Then we will always compare i and j

$$\Pr[\text{we compare } i \text{ and } j] = 1$$

Formal Argument for $n \log n$ Average

- Probability of comparing element i with element j:
- $\Pr[\text{we compare } i \text{ and } j] = \frac{2}{j-i+1}$

- Expected 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

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

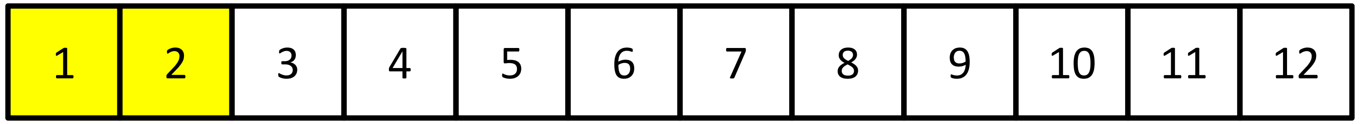
$$= 2 \sum_{i=1}^{n-1} \Theta(\log n) = \Theta(n \log n)$$

Quicksort overall: expected $\Theta(n \log n)$

Expected number of Comparisons

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

Consider when $i = 1$



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

Compared if 1 or 2 are chosen as pivot
(these will always be compared)

Sum so far: $\frac{2}{2}$

Expected number of Comparisons

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

Consider when $i = 1$

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

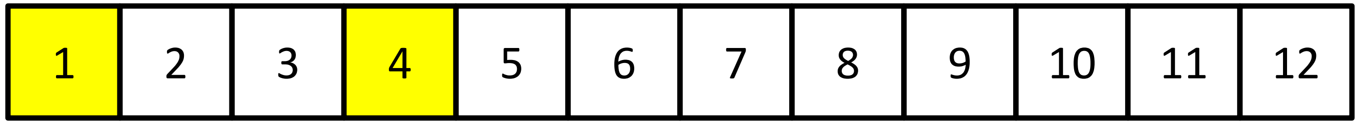
Compared if 1 or 3 are chosen as pivot
(but never if 2 is ever chosen)

Sum so far: $\frac{2}{2} + \frac{2}{3}$

Expected number of Comparisons

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

Consider when $i = 1$



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

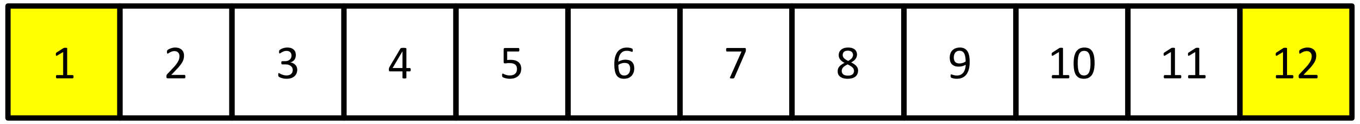
Compared if 1 or 4 are chosen as pivot
(but never if 2 or 3 are chosen)

Sum so far: $\frac{2}{2} + \frac{2}{3} + \frac{2}{4}$

Expected number of Comparisons

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

Consider when $i = 1$



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

Compared if 1 or 12 are chosen as pivot
(but never if 2 -> 11 are chosen)

Overall sum: $\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n}$

Expected number of Comparisons

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

When $i = 1$:

$$2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right) < 2\boxed{\sum_{x=1}^{n} \frac{1}{x}} \qquad O(\log n)$$

n terms overall in the outer sum

Quicksort overall: expected $O(n \log n)$

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 $O(n \log n)$

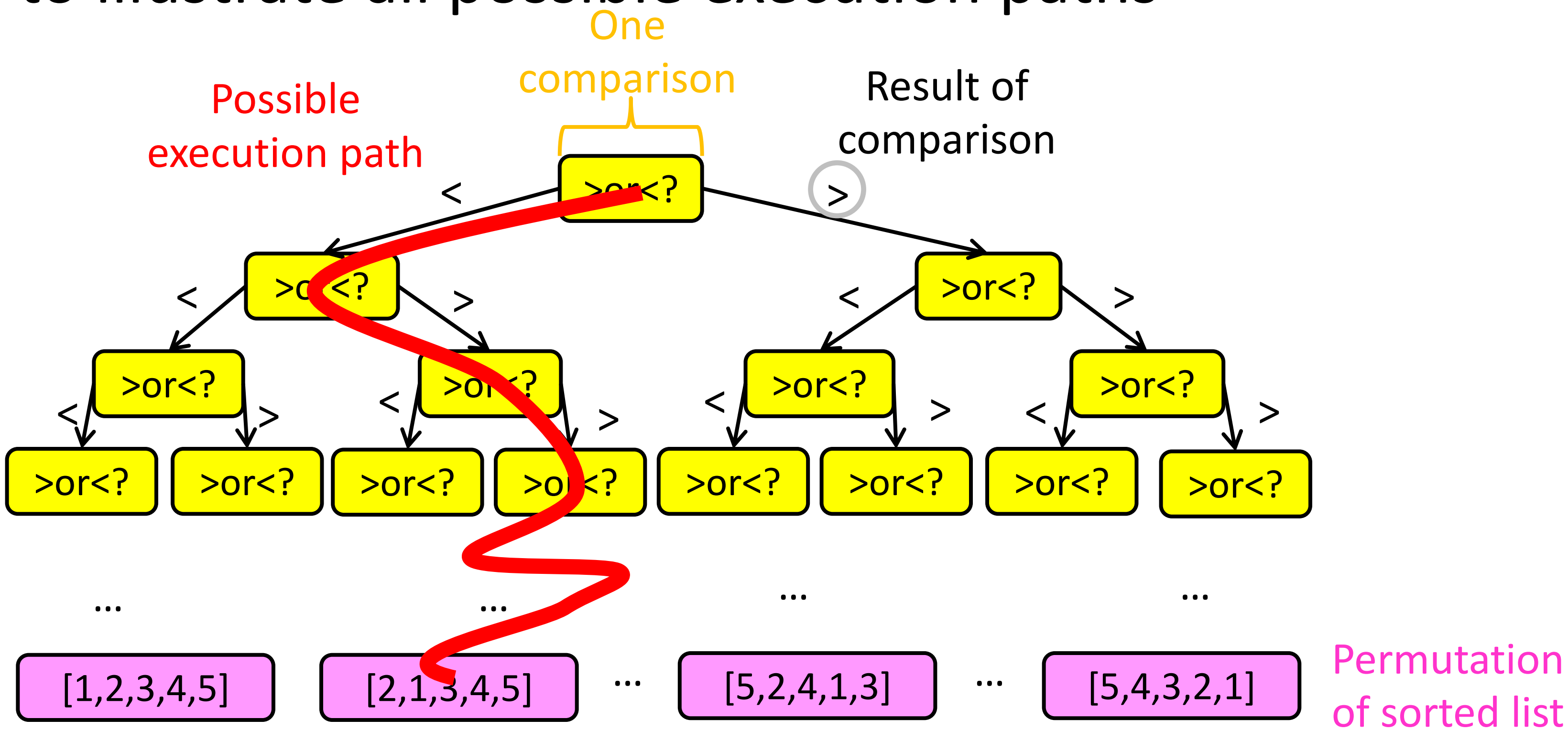
Can we do better than $O(n \log n)$?

Worst Case Lower Bounds

- Prove that there is no algorithm which can sort faster than $O(n \log n)$
  - Every algorithm, in the worst case, must have a certain lower bound
- Non-existence proof!
  - Very hard to do

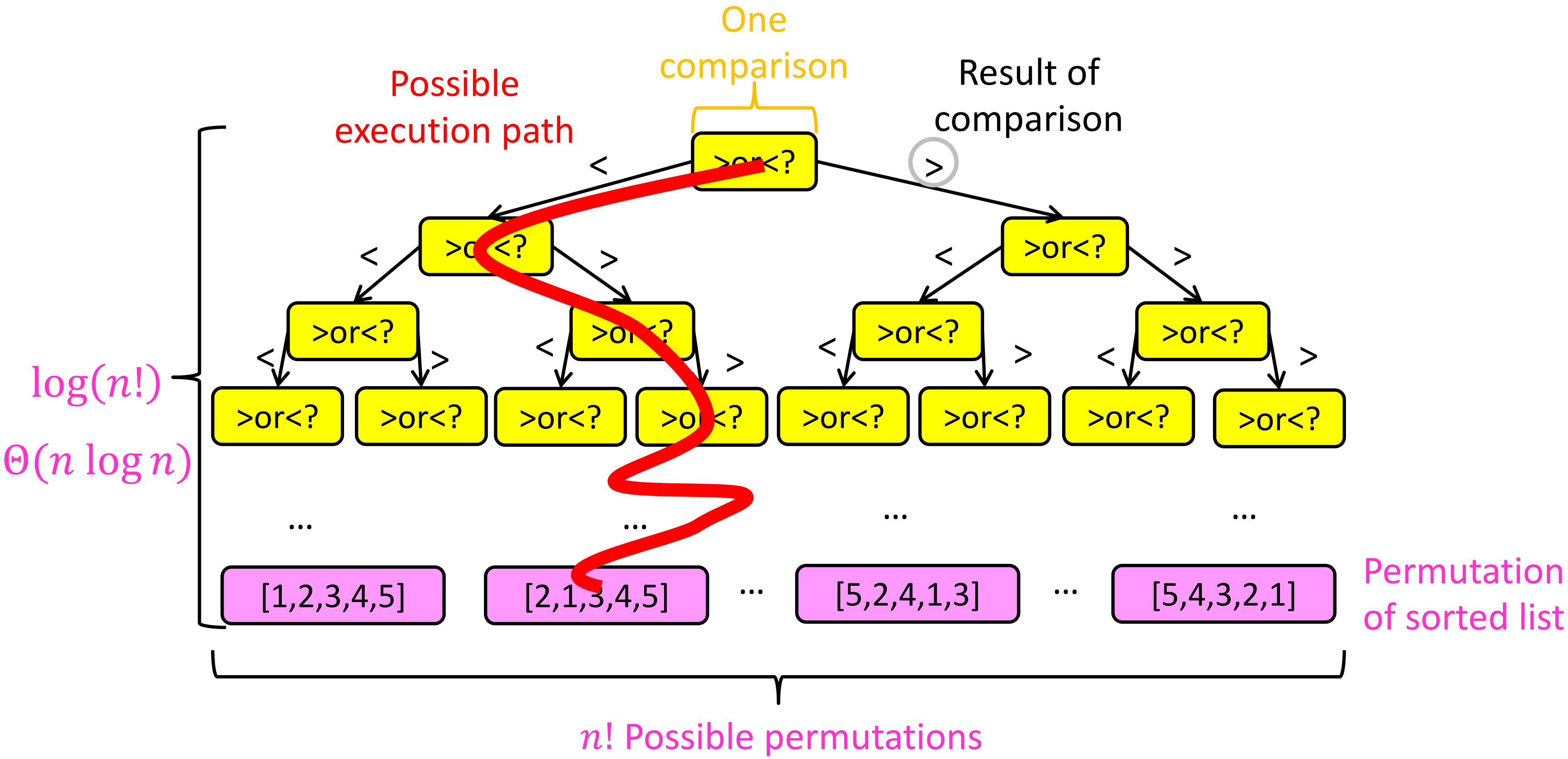
Strategy: Decision Tree

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



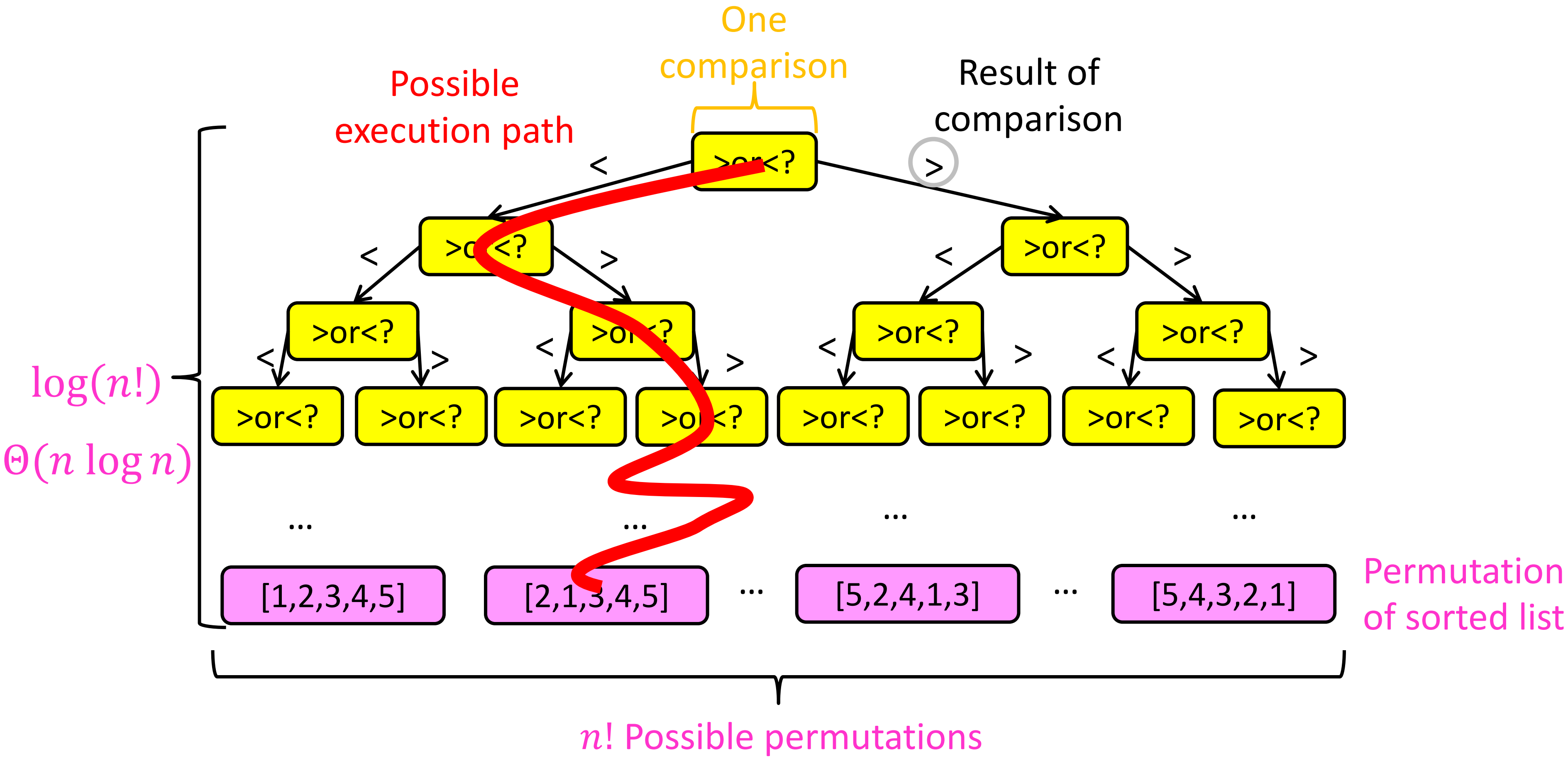
Strategy: Decision Tree

- 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 subl ist 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!

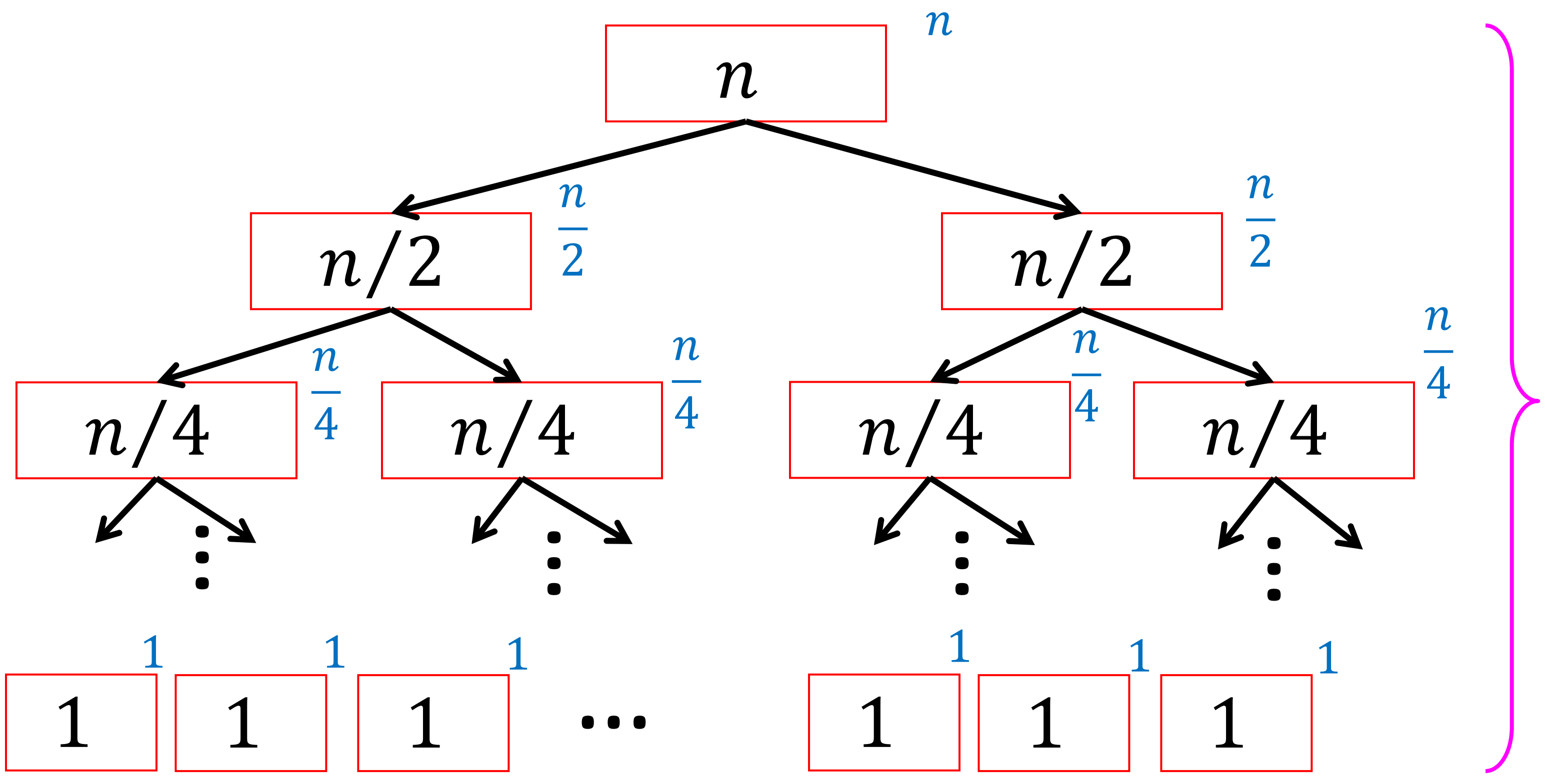
Mergesort

- **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)

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



n total / level

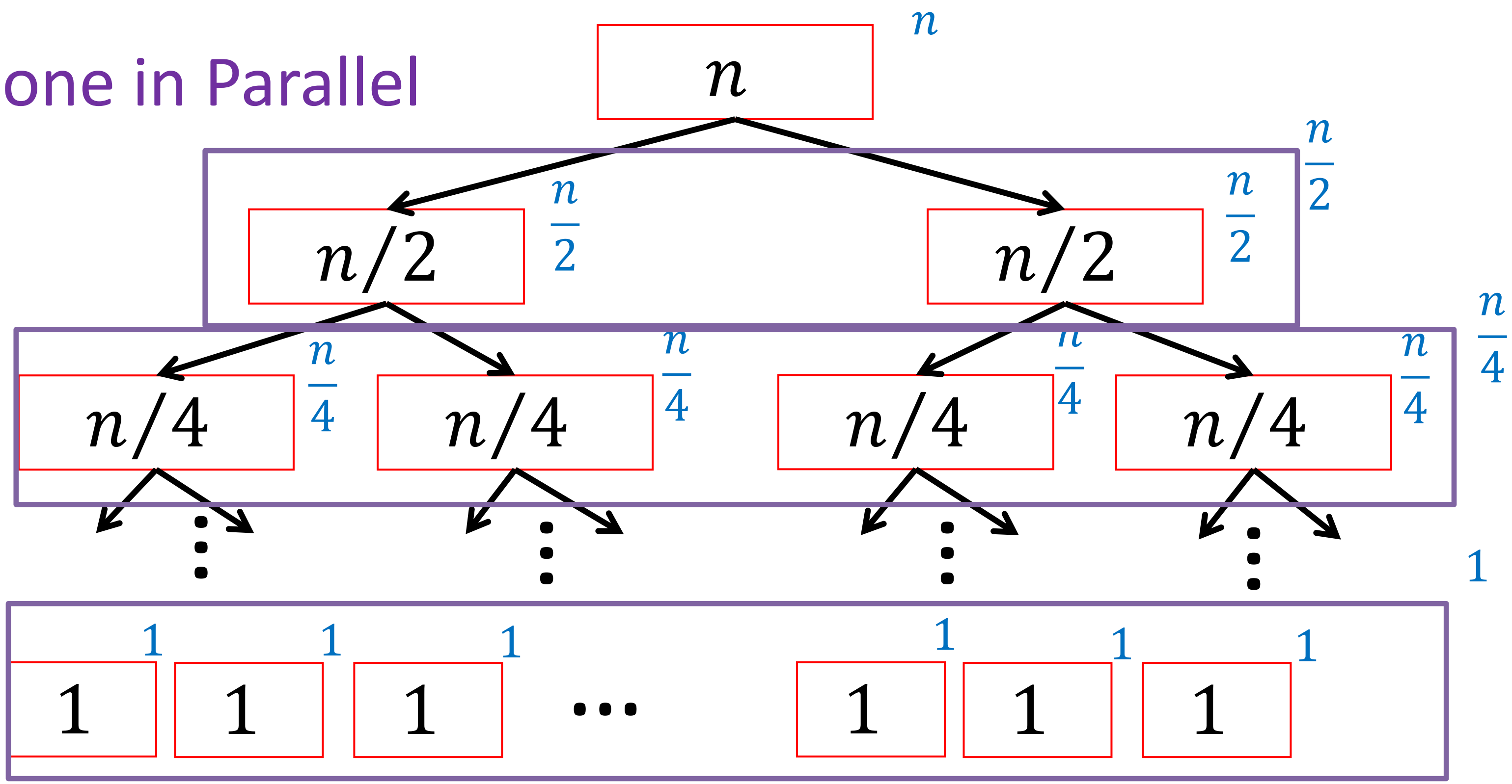
$\log_2 n$ levels of recursion

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

Mergesort (Parallel)

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

Done in Parallel



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?

$\Theta(n \log n)$

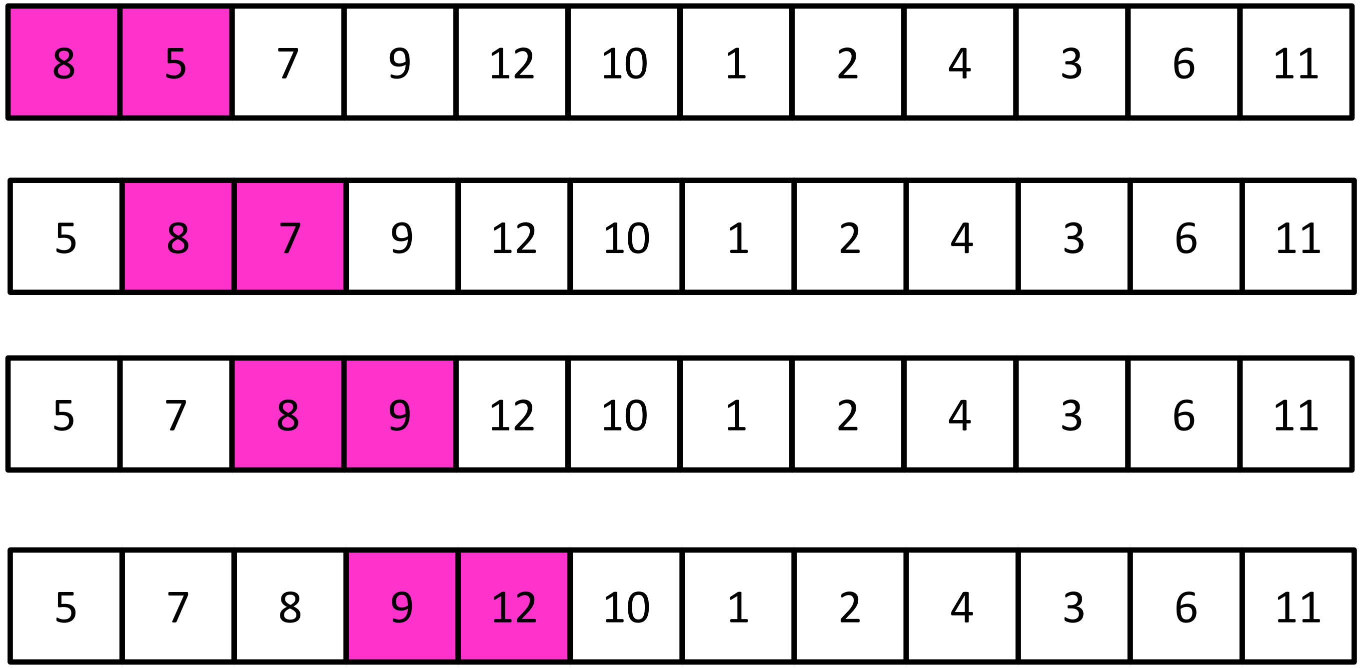
(almost always)

Better constants than Mergesort

In Place?	Adaptive?	Stable?	Parallelizable?
kinda Uses stack for recursive calls	No!	No	Yes!

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

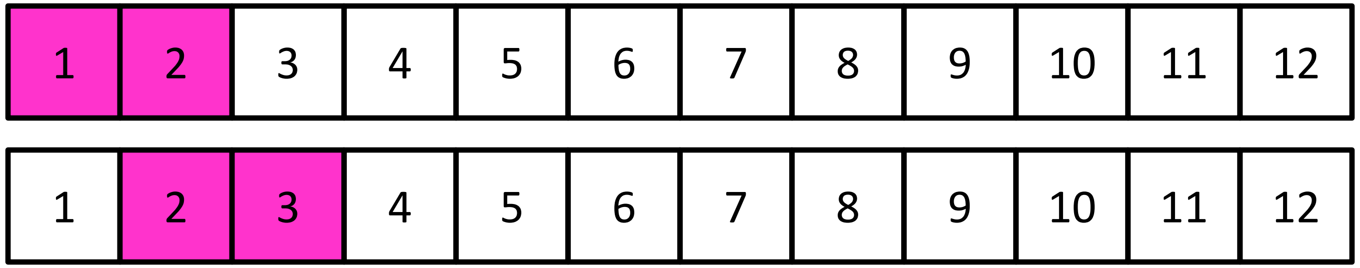
Adaptive?

Kinda

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

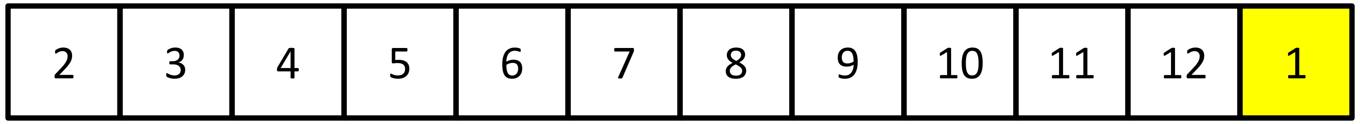
Bubble Sort is “almost” Adaptive

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



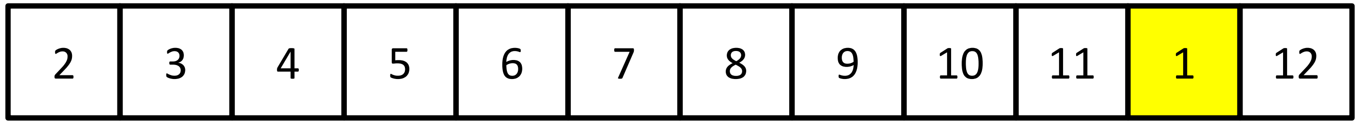
1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12

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?

$\Theta(n^2)$

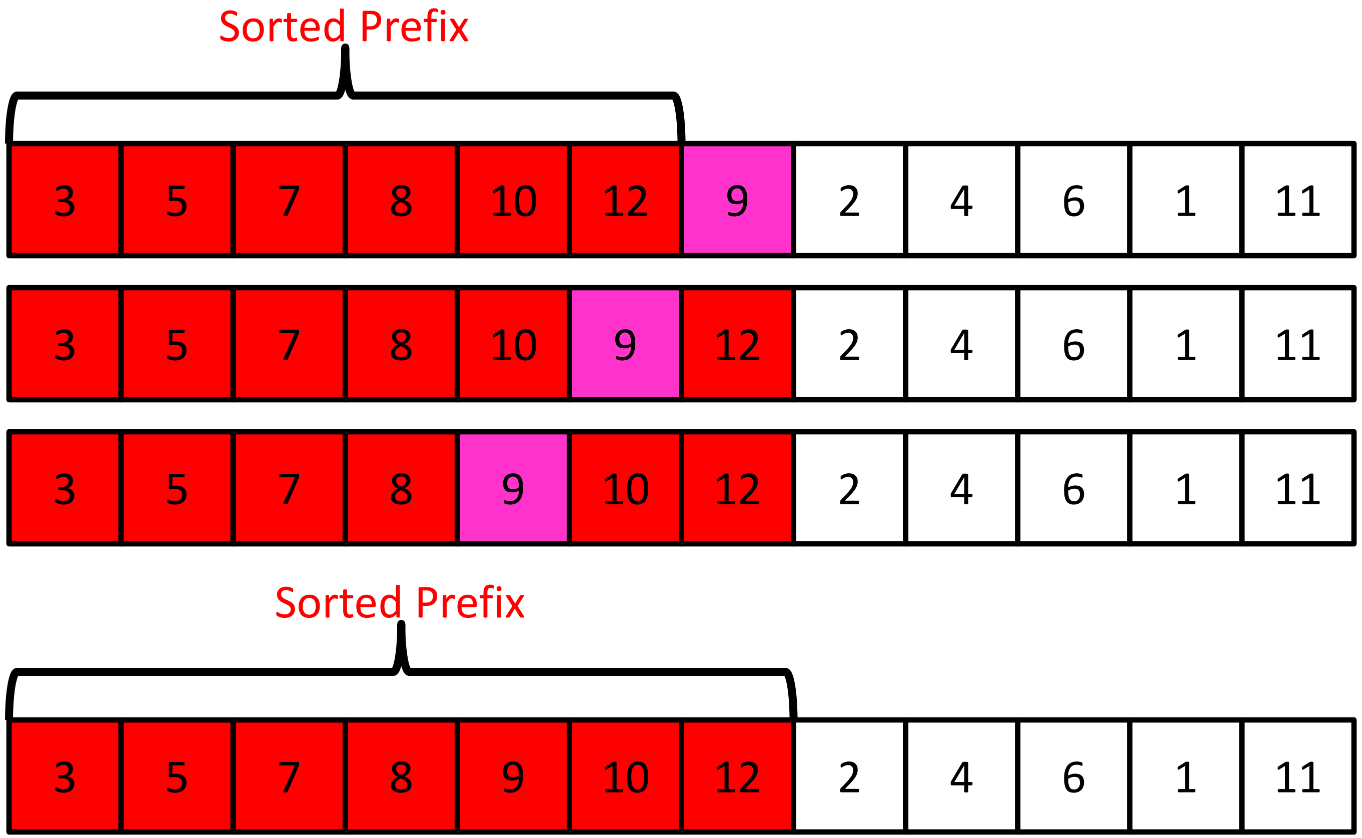
Constants worse than Insertion Sort

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



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

Run Time?

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

In Place?

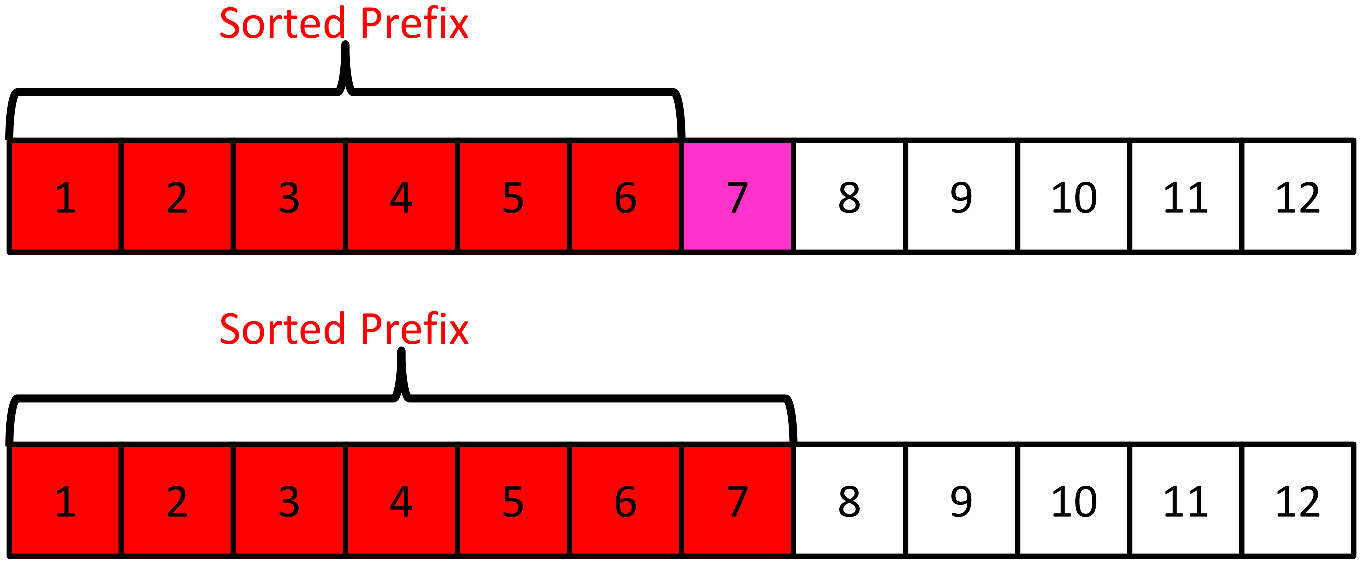
Yes!

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?

$\Theta(n^2)$

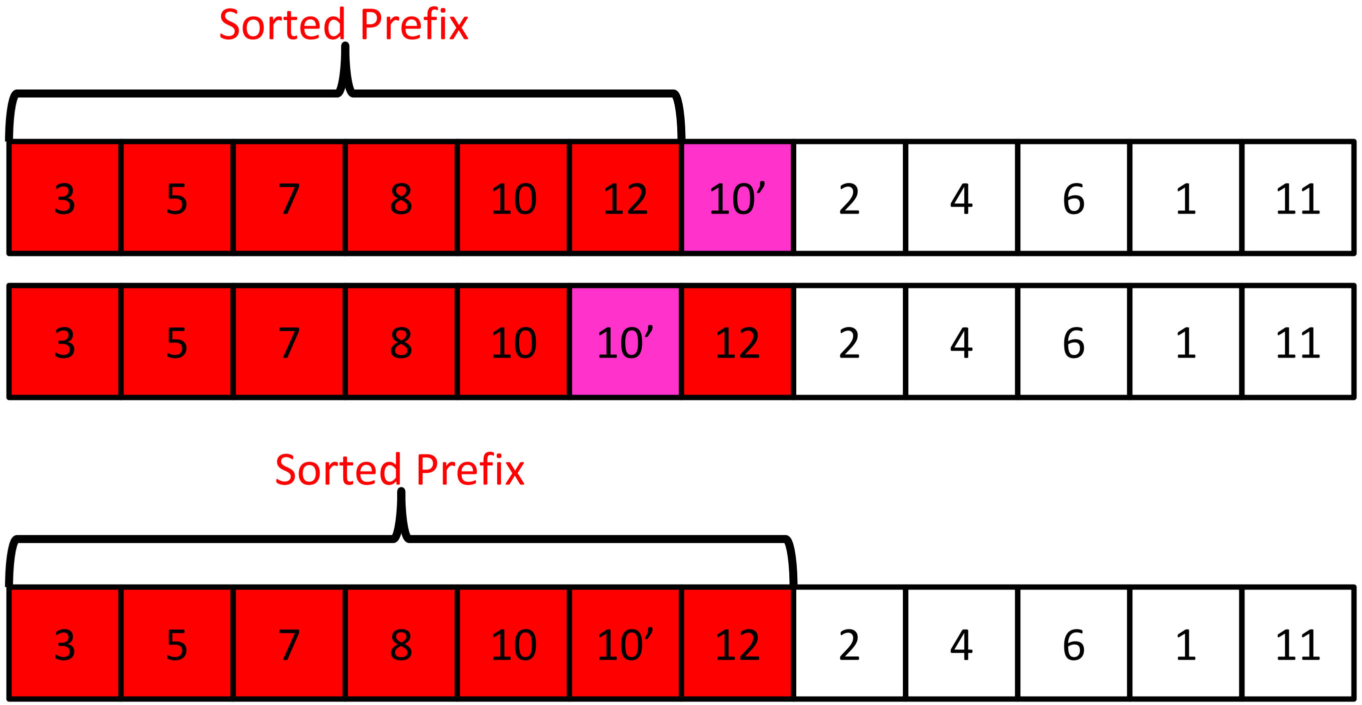
(but with very small constants)

Great for short lists!

In Place?	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

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

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

Online?
Yes

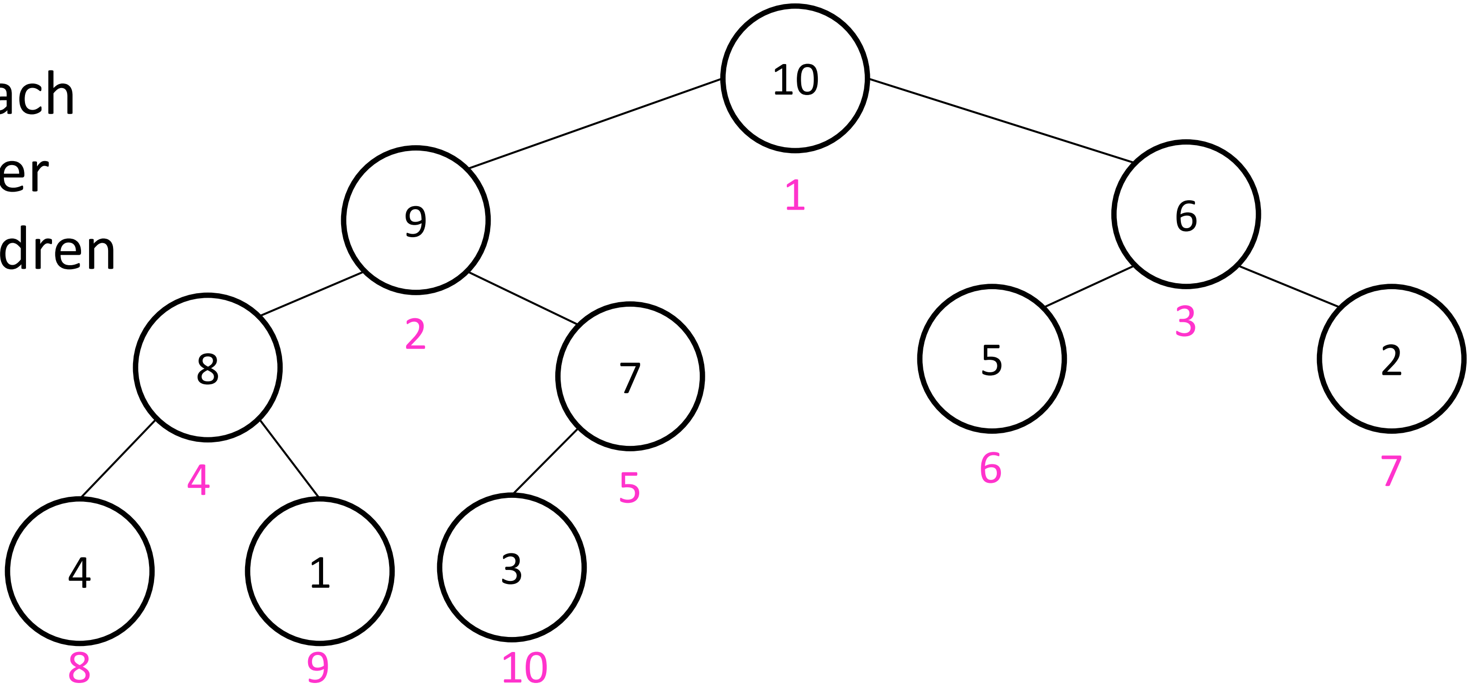
“All things considered, it’s actually a pretty good sorting algorithm!” –Nate Brunelle

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

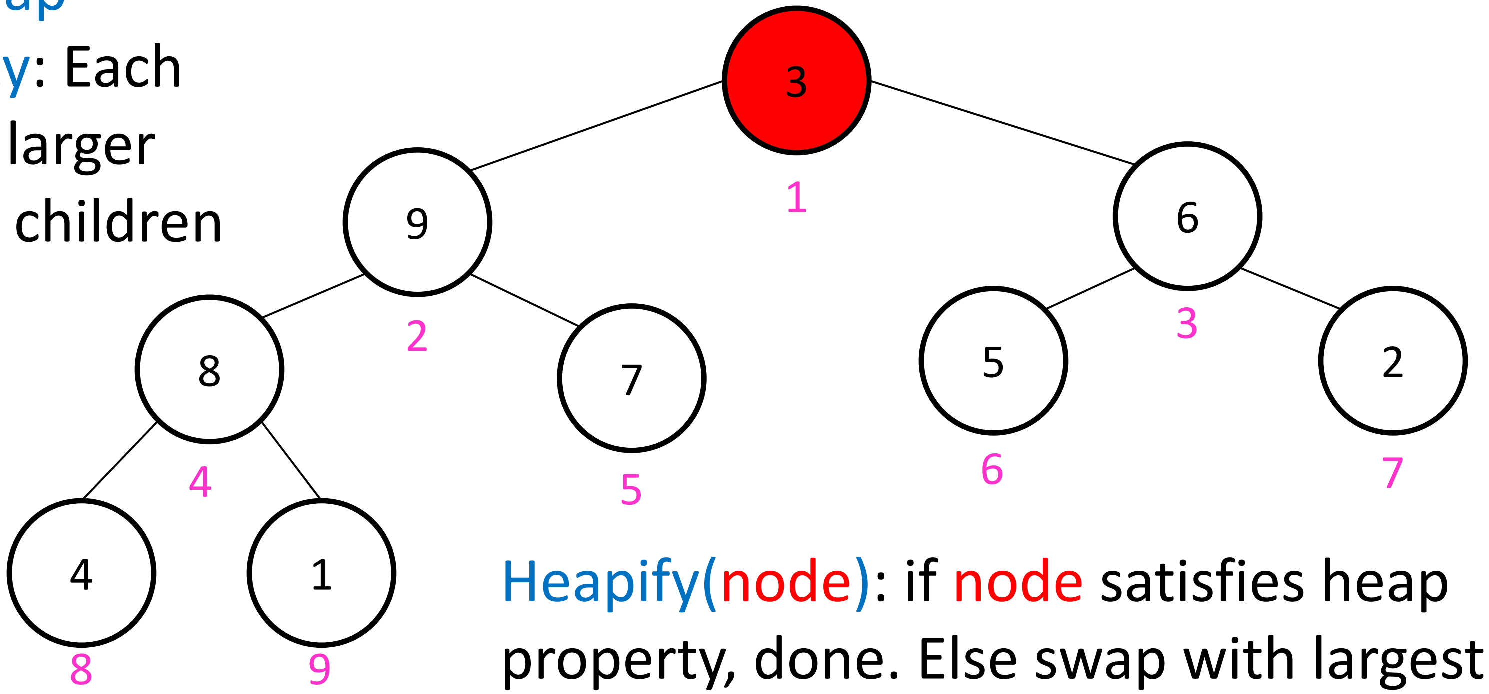


Heap Sort

- 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



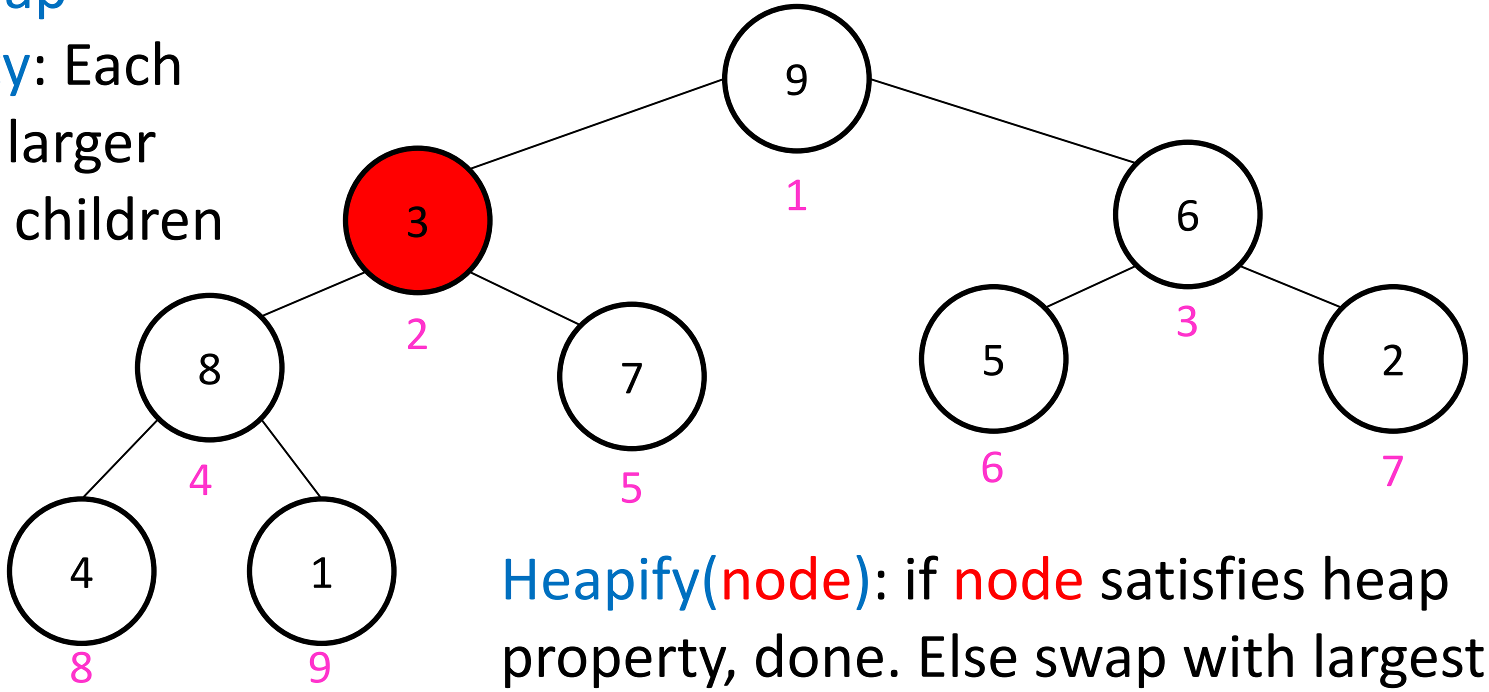
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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

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

Max Heap Property: Each node is larger than its children



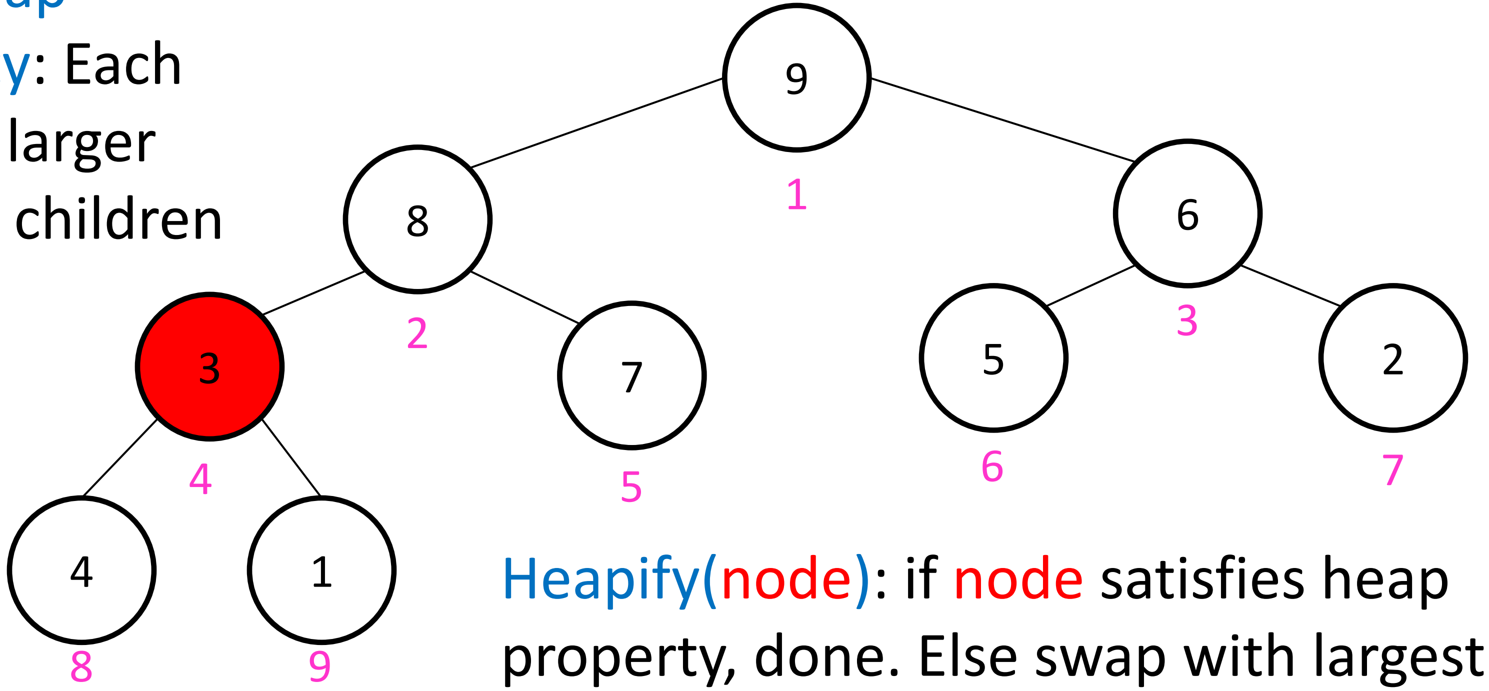
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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

	9	8	6	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



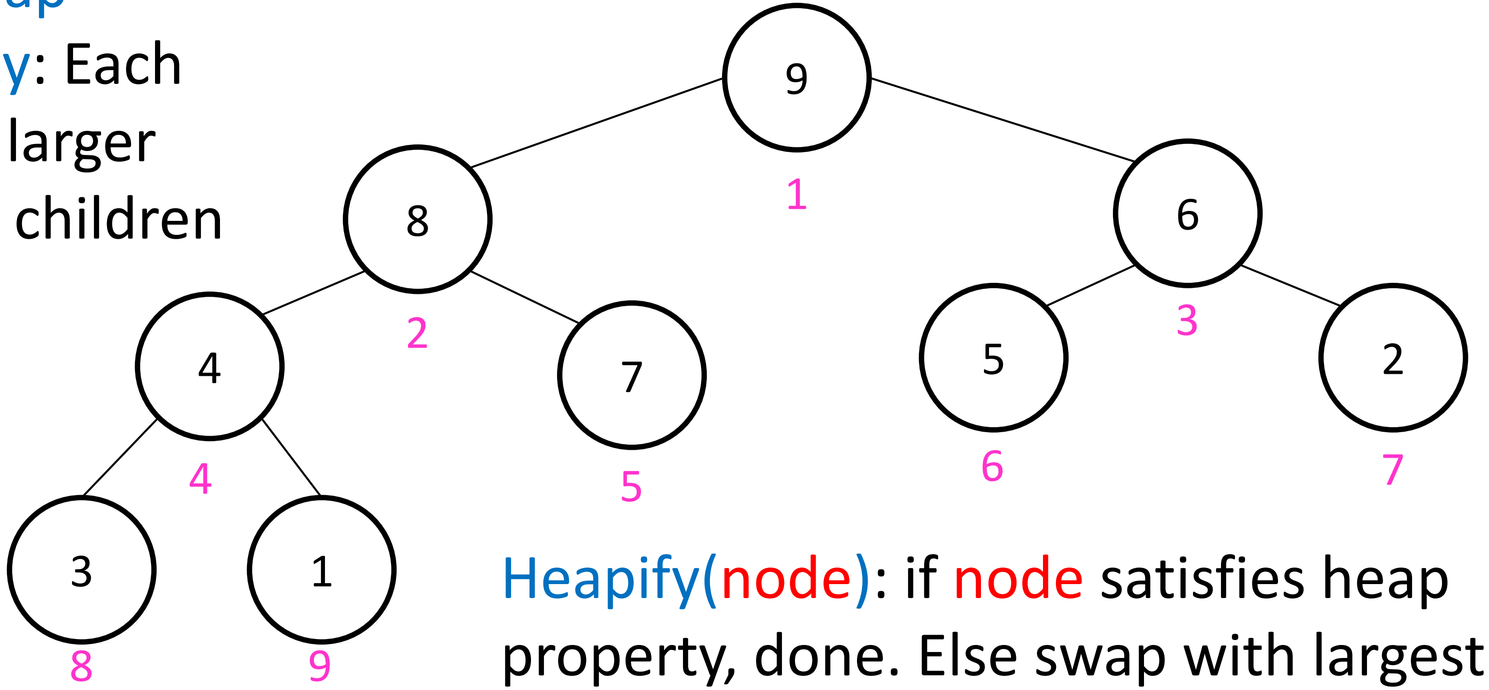
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

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!

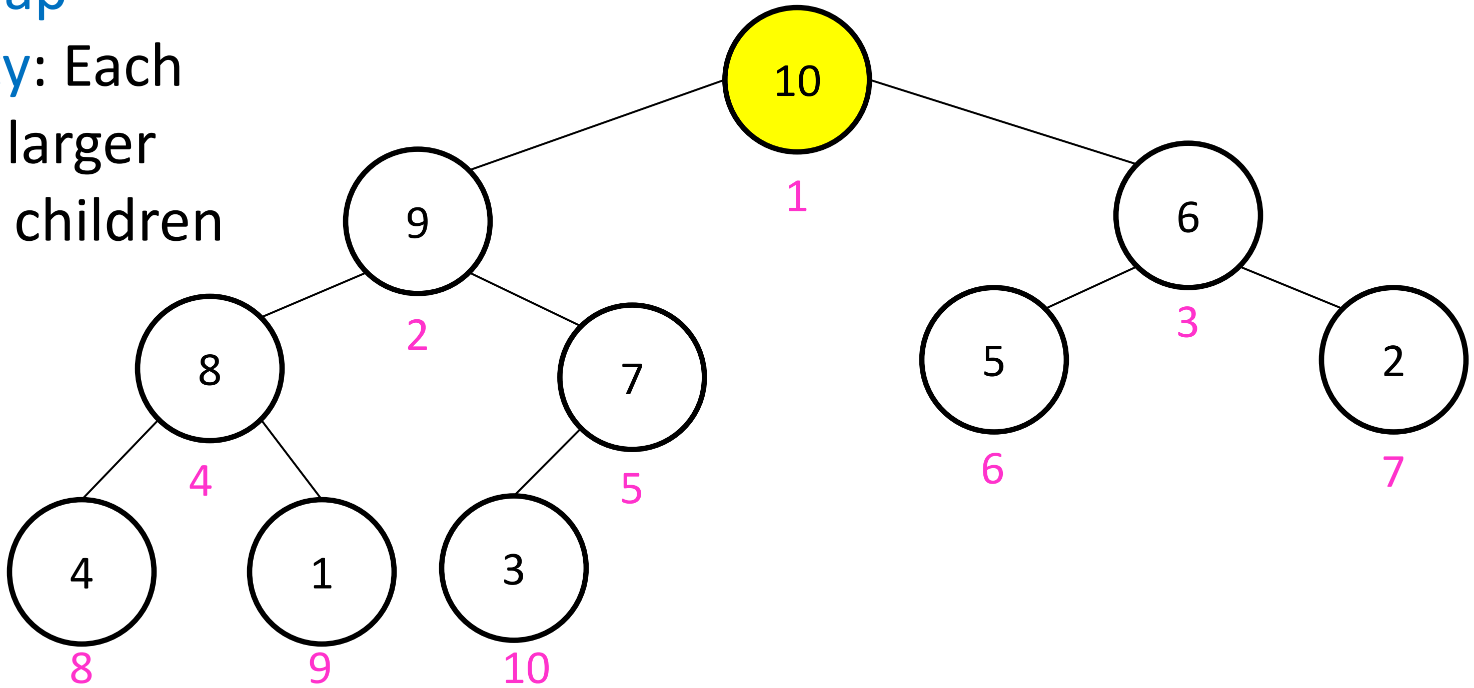
When removing an element from the heap, move it to the (now unoccupied) end of the list

In Place Heap Sort

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

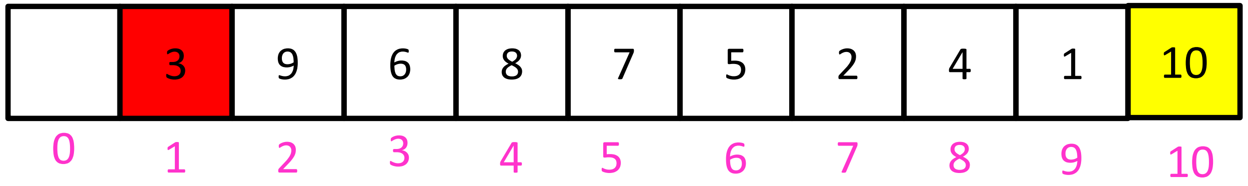
	10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9	10

Max Heap Property: Each node is larger than its children



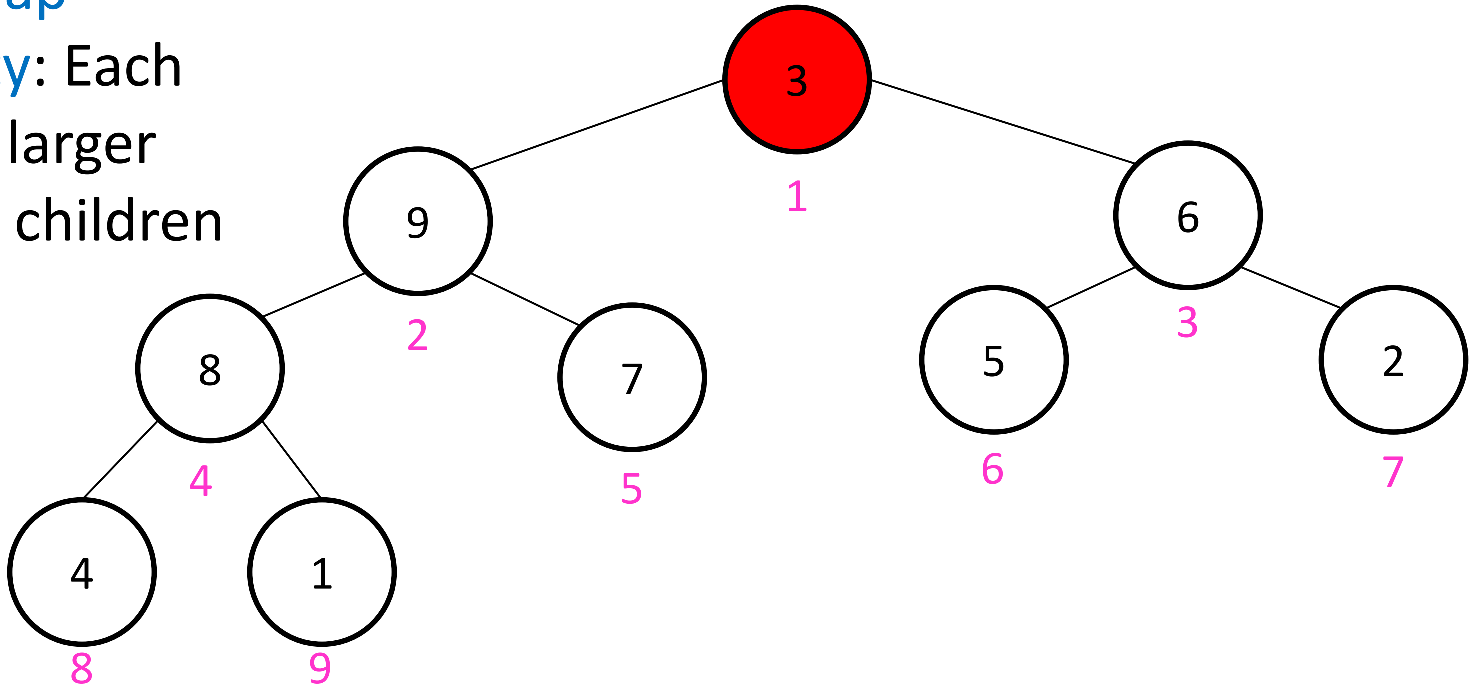
In Place Heap Sort

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



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

Max Heap Property: Each node is larger than its children

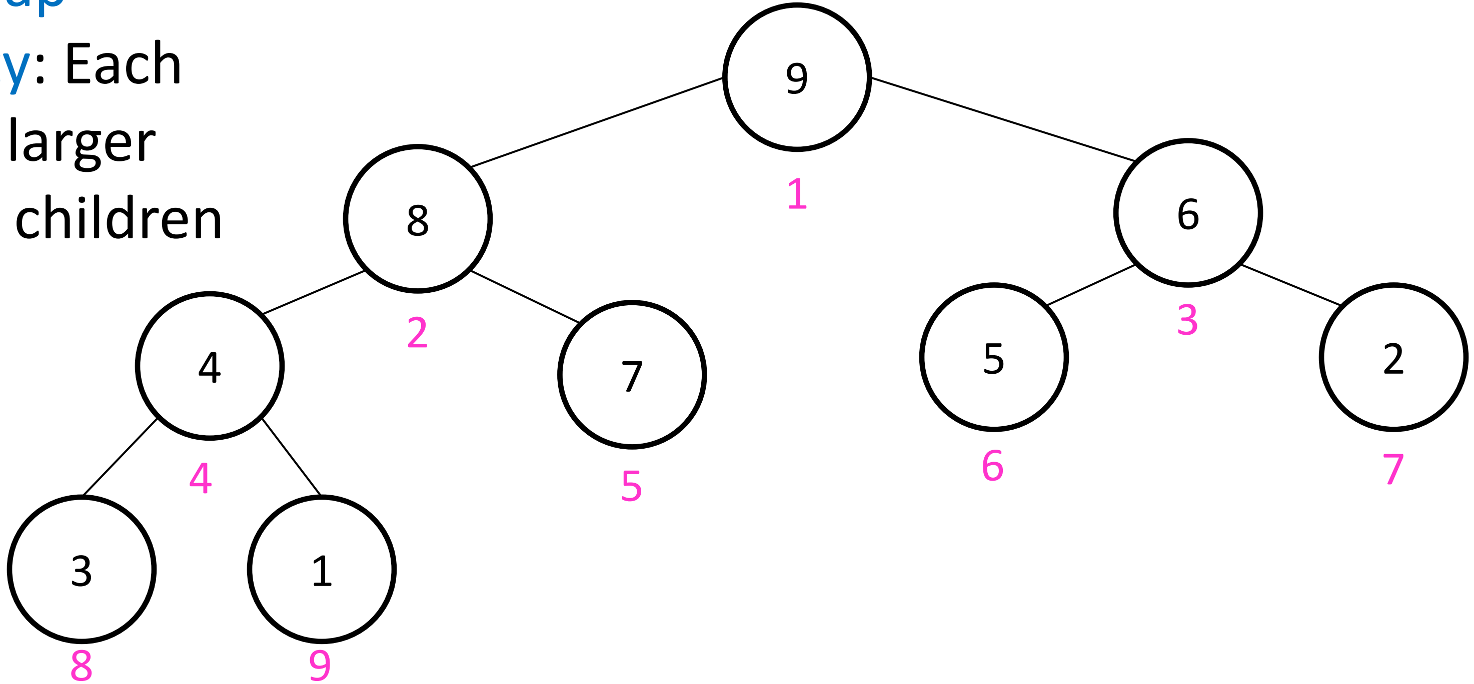


In Place Heap Sort

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

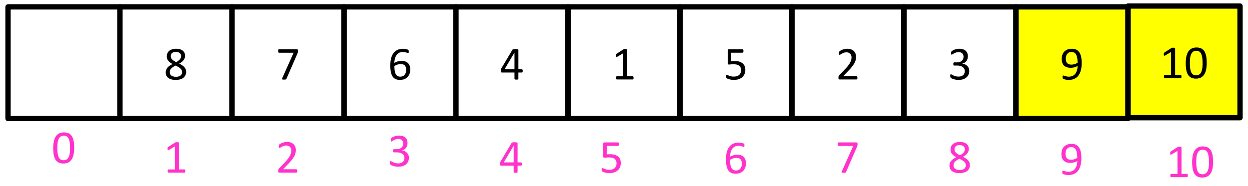
	9	8	6	4	7	5	2	3	1	10
0	1	2	3	4	5	6	7	8	9	10

Max Heap Property: Each node is larger than its children



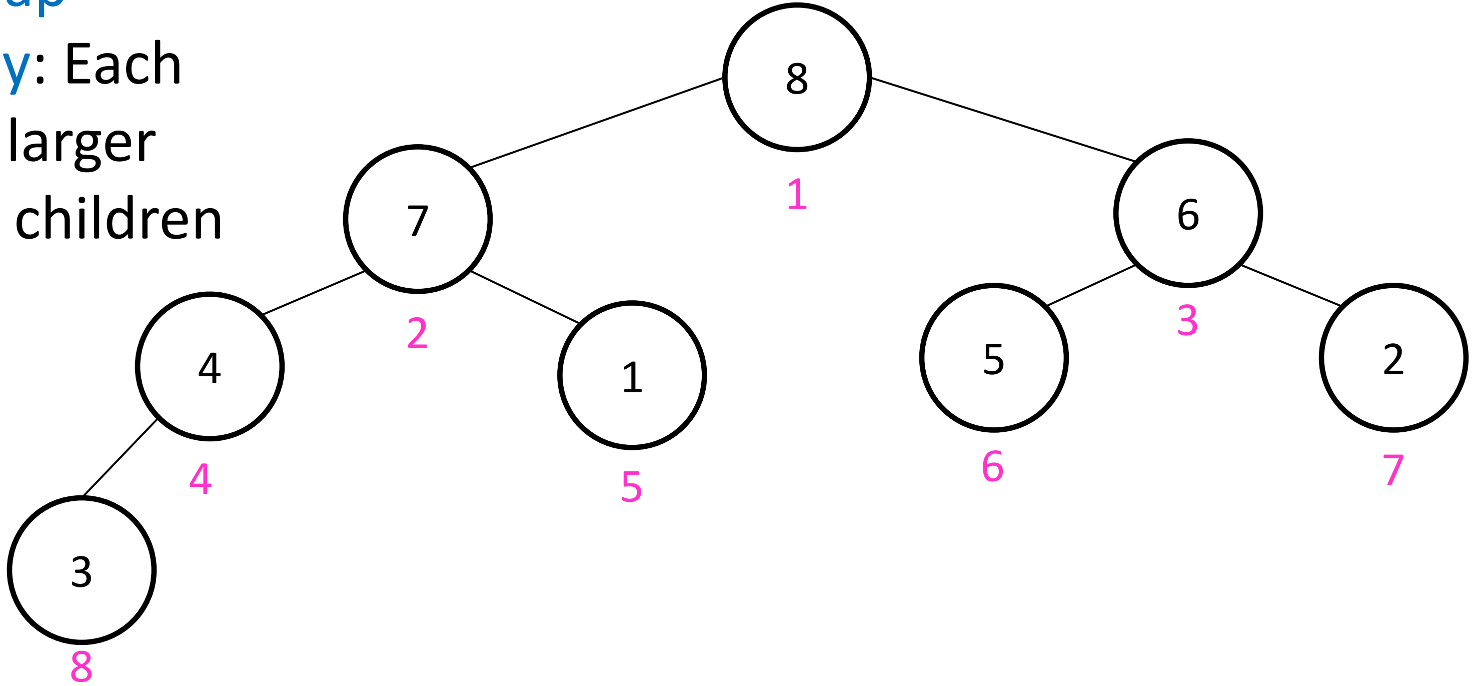
In Place Heap Sort

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



	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

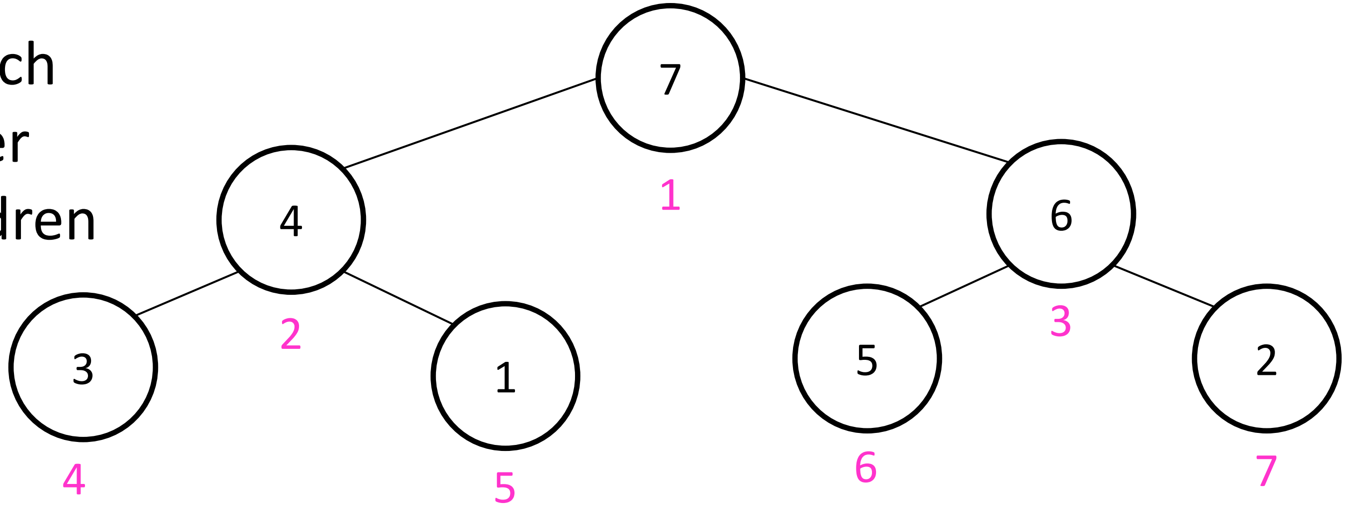


In Place Heap Sort

- **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 Right-to-Left

Run Time?

$\Theta(n \log n)$

Constants worse than Quick Sort

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