CS4102 Algorithms

Spring 2020 – Horton's Section

Warm up

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

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

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

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

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

$$\begin{array}{lccccccc} & \| & \wedge & & \wedge & & \wedge & \wedge \\ n^n = & n \cdot & n & \cdot & n & \cdot \ldots \cdot & n & \cdot n \end{array}$$

$n! \leq n^n$

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

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

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

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

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

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

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

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

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

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

Today's Keywords

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

CLRS Readings

- Chapter 7
- Chapter 8

Homeworks

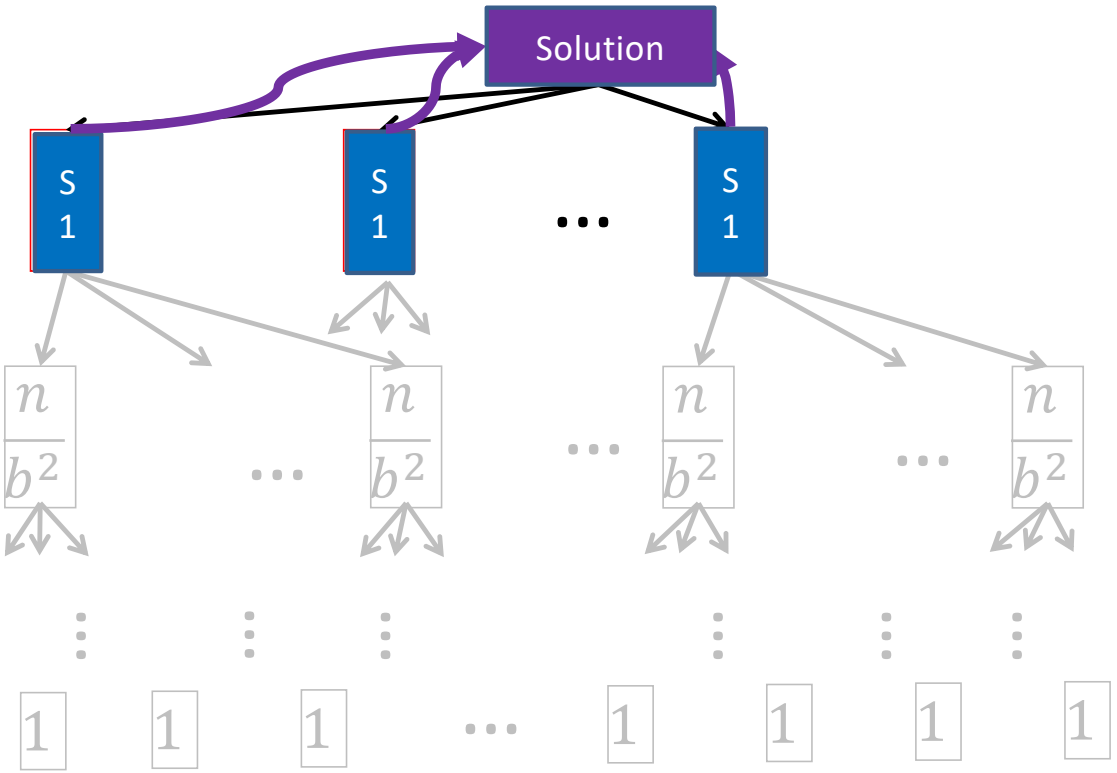
- HW4 due 11pm Thursday, February 27, 2020
  - Divide and Conquer and Sorting
  - Written (use LaTeX!)
  - Submit BOTH a pdf and a zip file (2 separate attachments)
- Regrade Office Hours
  - Fridays 2:30pm-3:30pm (Rice 210)
  - 2 weeks for HW0 regrades

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
```

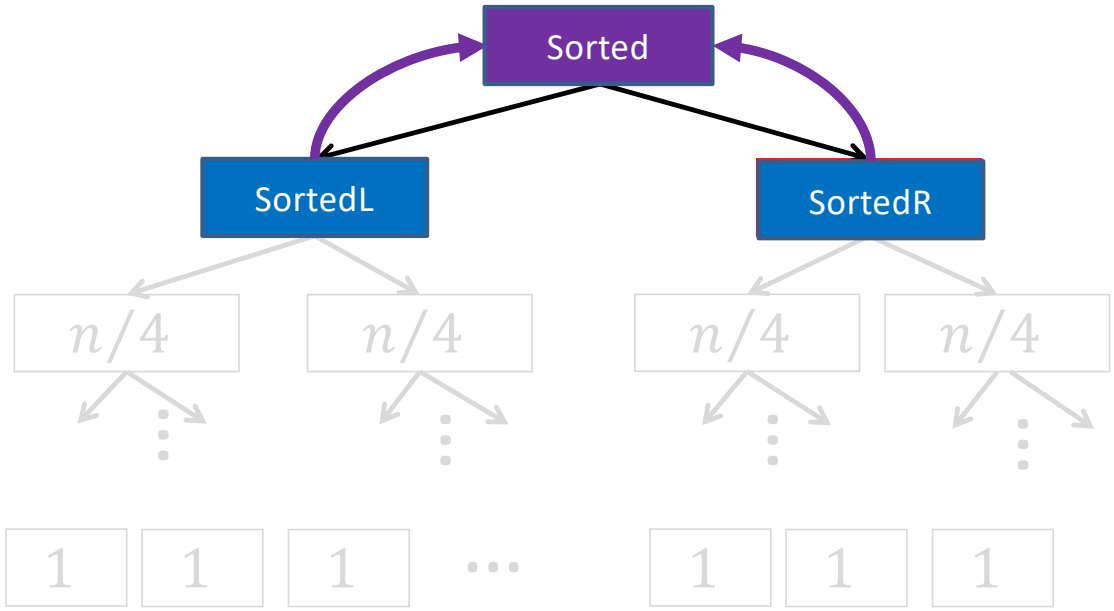
Generic Divide and Conquer Solution



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!

Review: What We've Learned

- Quicksort's time-complexity
  - Worst-case: $\theta(n^2)$ when always make most uneven partition
    - Odds of this happening are very very low
    - Can avoid with randomized partition, or median-of-three
  - Even a partition that divides 10%/90% is $\Theta(n \log n)$
- No surprise that Quicksort's average-case time-complexity is $\Theta(n \log n)$
  - Let's show that!

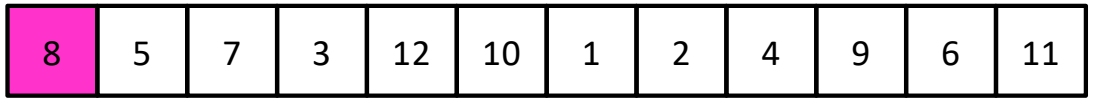
Formal Argument for $n \log n$ Average

- 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

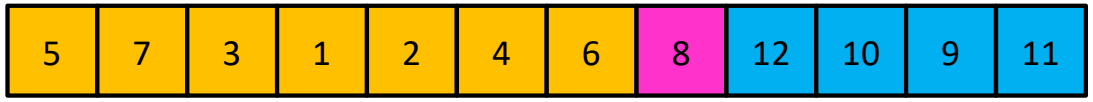
Partition (Divide step)

Given: a list, a pivot value p

Start: unordered list

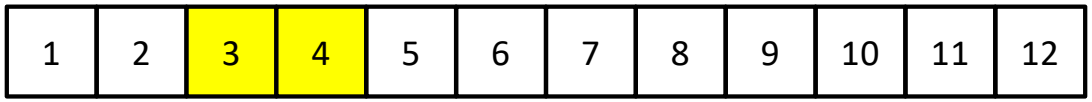


Goal: All elements $< p$ on left, all $> p$ on right



Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?



Consider the sorted version of the list.

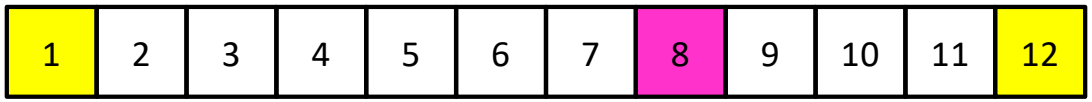
Observation: Adjacent elements must have been compared

- **Why?** Otherwise we would not know which came first
- **Every** sorting algorithm **must** compare elements that are adjacent in final sorted result.

In quicksort: adjacent elements always end up in same sublist, unless one is the pivot

Formal Argument for $n \log n$ Average

What if not adjacent? What is the probability of comparing two given elements?



Again, considering the sorted result.

Remember, for one Partition call, elements only compared to the pivot element.

Only compared if 1 or 12 was chosen as the first pivot.

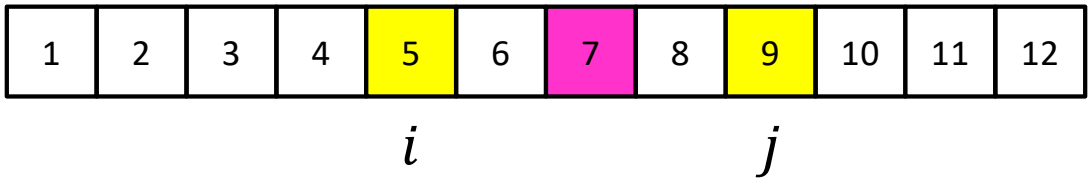
For later partitions, they are in different sublists

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

Assuming pivot is chosen uniformly at random. Twelve possible choices.

Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?



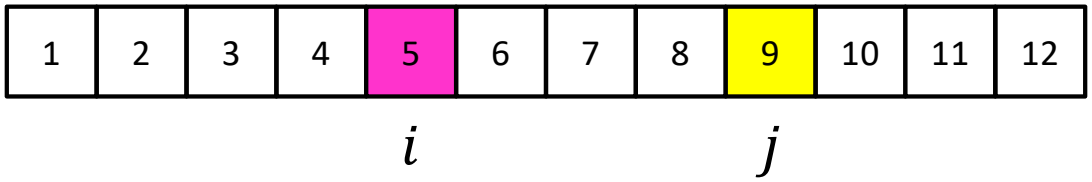
Case 1: Pivot contained in $[i+1, \ldots, j-1]$

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?



Case 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 i with j ($j > i$):

- Compare for 2 cases, when pivot is i or j
- Out of a possible $j - i + 1$ cases, the number of elements between (and including) i and j

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

Expected number of comparisons for Quicksort:

$$\sum_{i<j} \frac{2}{j - i + 1}$$

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

Formal Argument for $n \log n$ Average

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: $k = j - i$

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

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

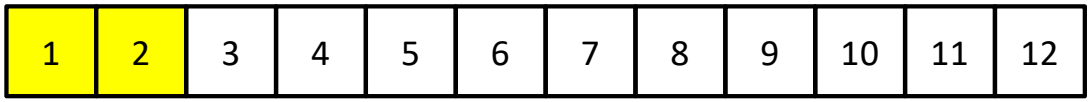
$$= 2 \sum_{i=1}^{n-1} \Theta(\log n) = \Theta(n \log n)$$

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 $i = 1$



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

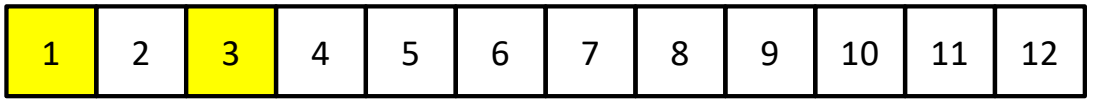
Remember $j - i + 1$ is the number of elements between (and including) i and j

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 $i = 1$



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

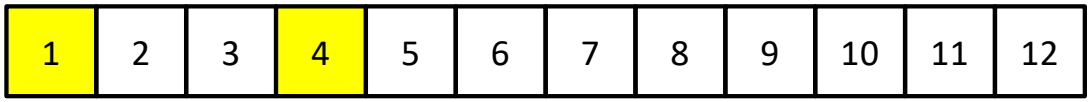
Remember $j - i + 1$ is the number of elements between (and including) i and j

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 $i = 1$



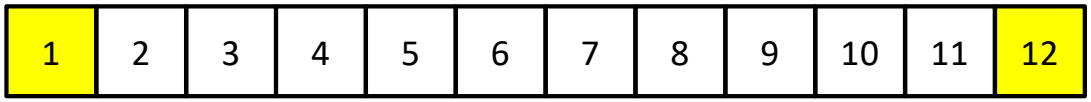
Compared if 1 or 4 are chosen as pivot
(but never if 2 or 3 are chosen)

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 $i = 1$



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

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

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

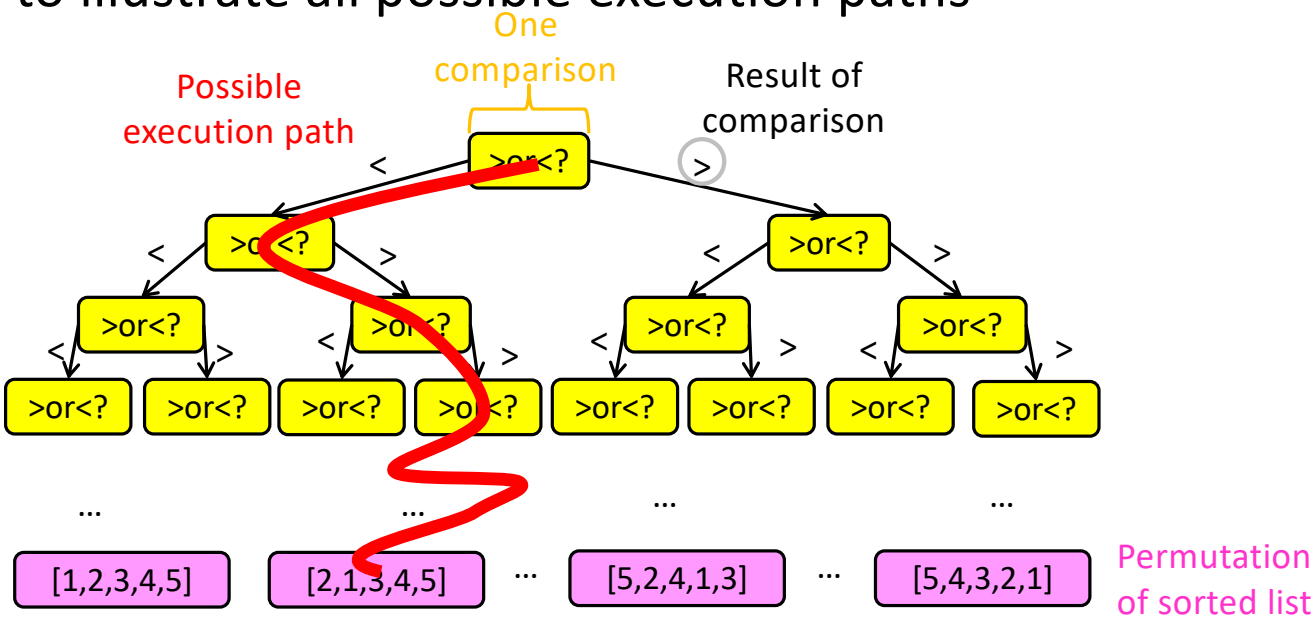
- Prove that there is no algorithm which can sort faster than $O(n \log n)$
  - Every algorithm, in the worst case, must have a certain lower bound
- Might seem very hard to prove something does not exist!
- **Lower bounds proofs**
  - Proof based on properties of the problem
  - Thus apply to any algorithm that solves the problem

Decision Tree for Sorting

- Let's use a decision tree to analyze the class of all sorting algorithms that compare keys
  - Each internal node represents one comparison for keys x_i and x_j
  - Leaf nodes represent a particular result. I.e. a permutation of the original sequence
  - The action of Sort on a particular input corresponds to following one path from the root to a leaf
- What can we say about such trees?
  - Since a correct sort must handle all permutations of n items, there must be at least n! leaves

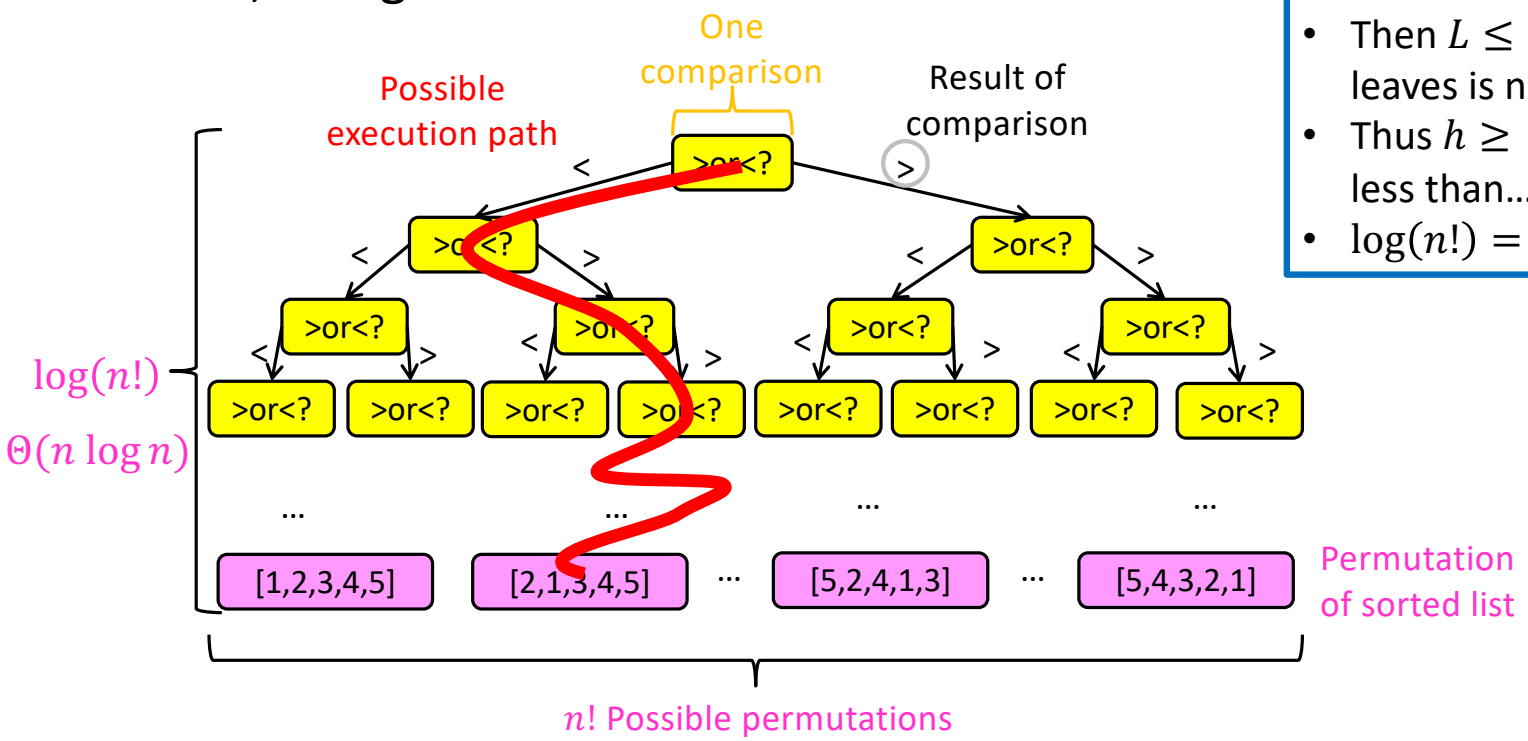
Strategy: Decision Tree

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



Strategy: Decision Tree

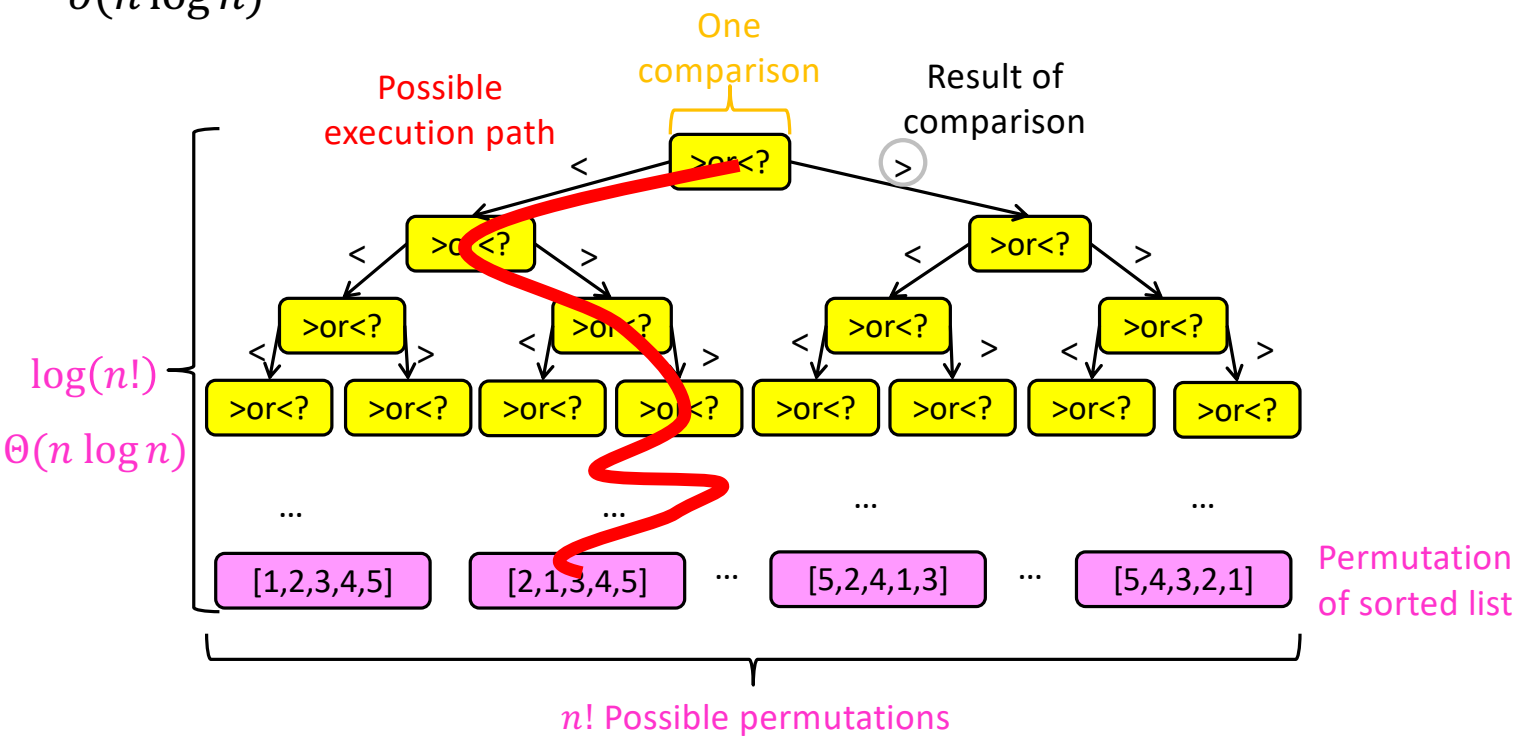
- Worst case run time is the longest execution path
- i.e., "height" of the decision tree



- Let L be number of leaves.
- Then $L \leq 2^h$. (Number of leaves is no more than 2^h.)
- Thus $h \geq \lceil \lg L \rceil$. (Height is not less than…)
- $\log(n!) = \Theta(n \log n)$

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!

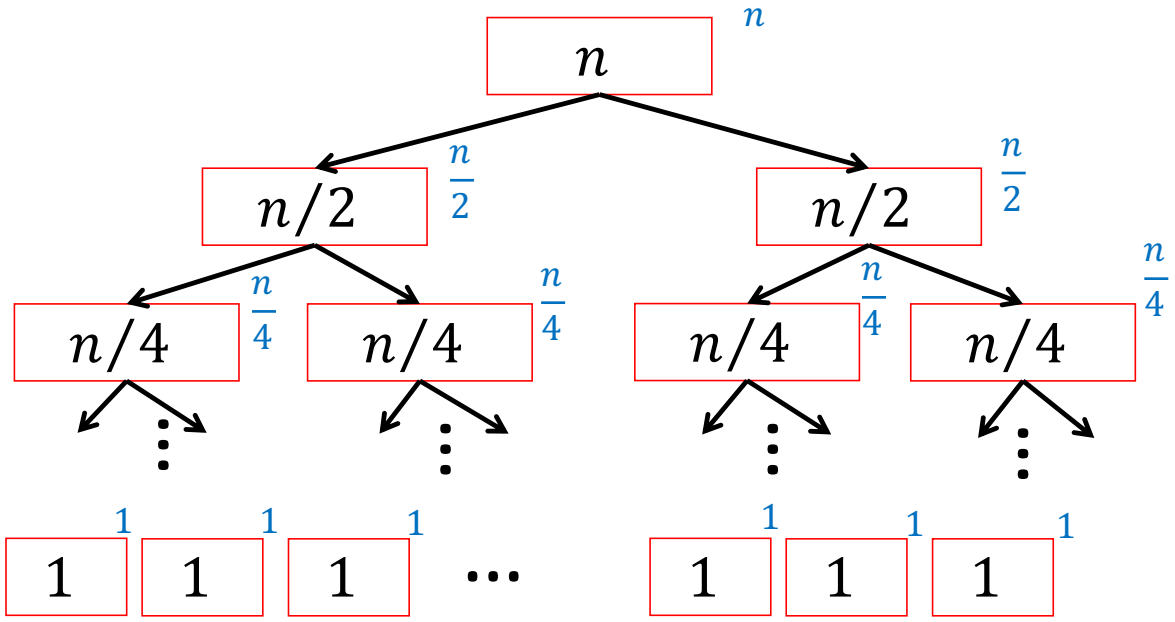
Mergesort

- **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 processors to work on each sublist

Mergesort (Sequential)

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



n total / level

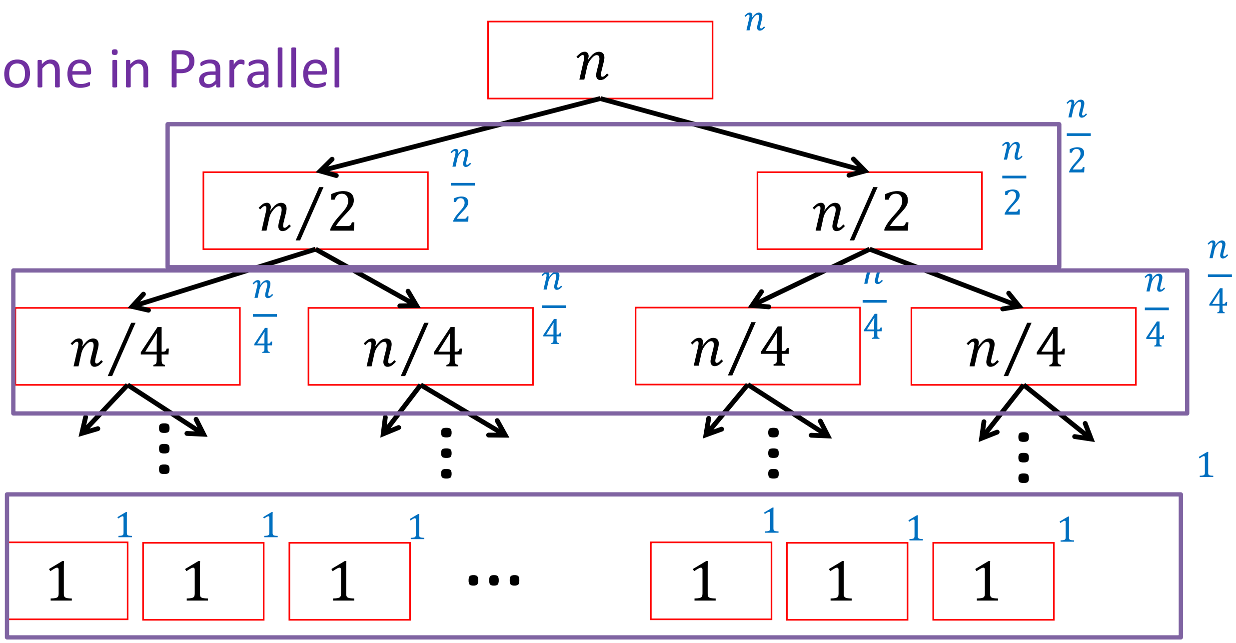
$\log_2 n$ levels of recursion

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

Mergesort (Parallel)

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

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?

$\Theta(n \log n)$

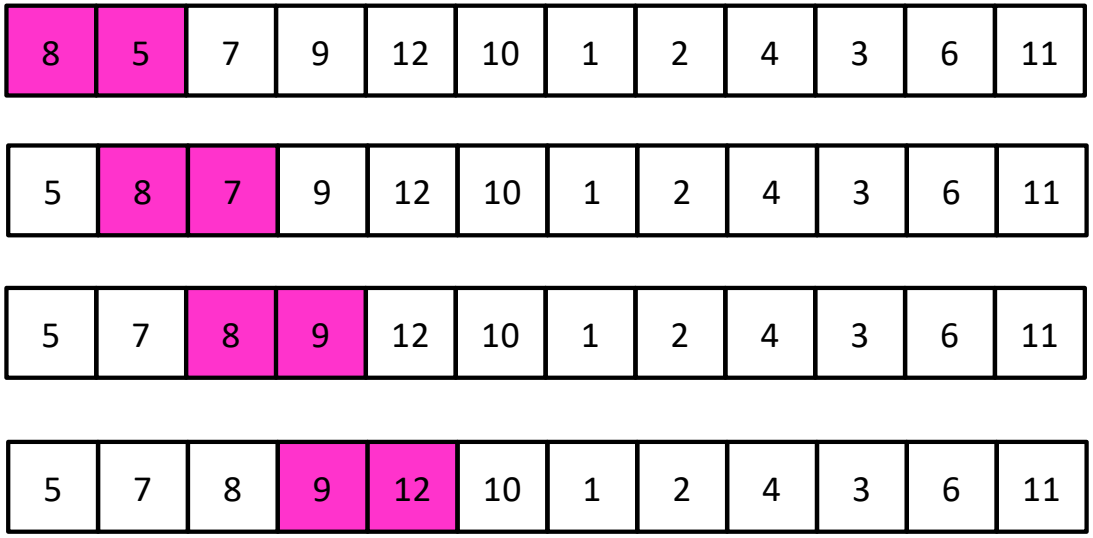
(almost always)

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



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

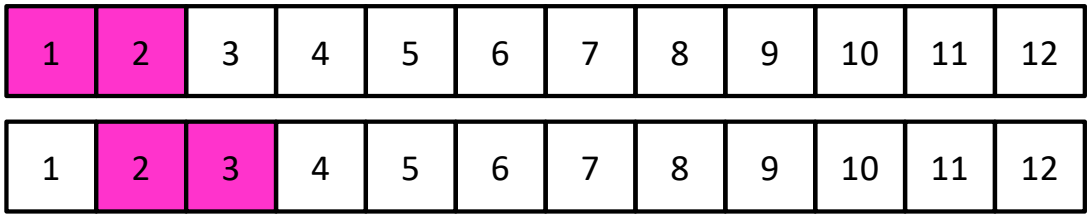
Adaptive?

Kinda

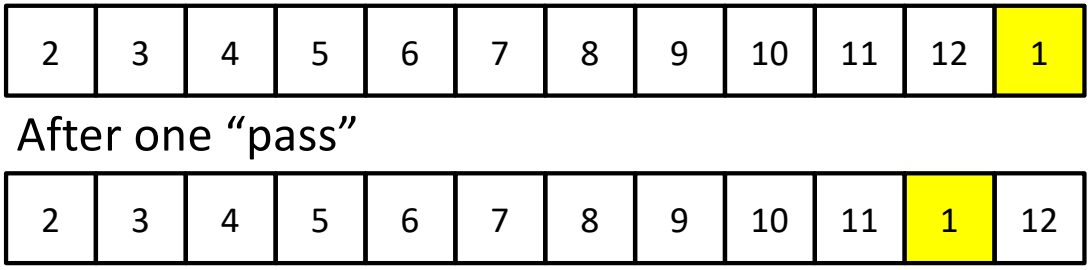
"Compared to straight insertion […], bubble sorting requires a more complicated program and takes about twice as long!"
–Donald Knuth

Bubble Sort is "almost" Adaptive

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



Only makes one "pass"



After one "pass"

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?

$\Theta(n^2)$

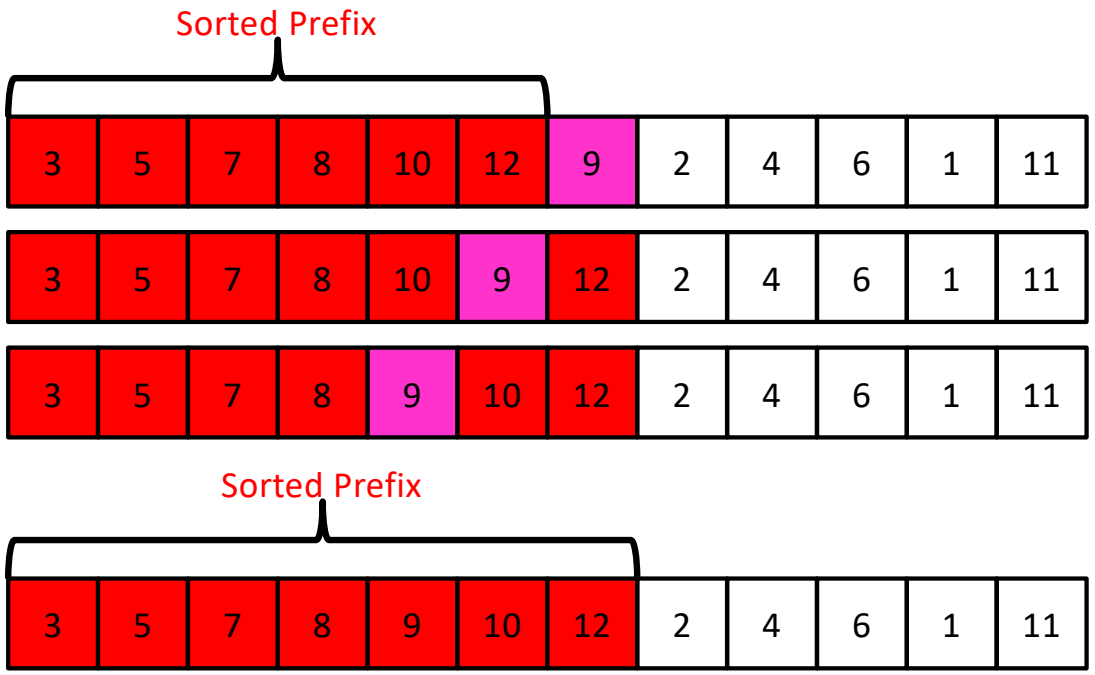
Constants worse than Insertion Sort

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?

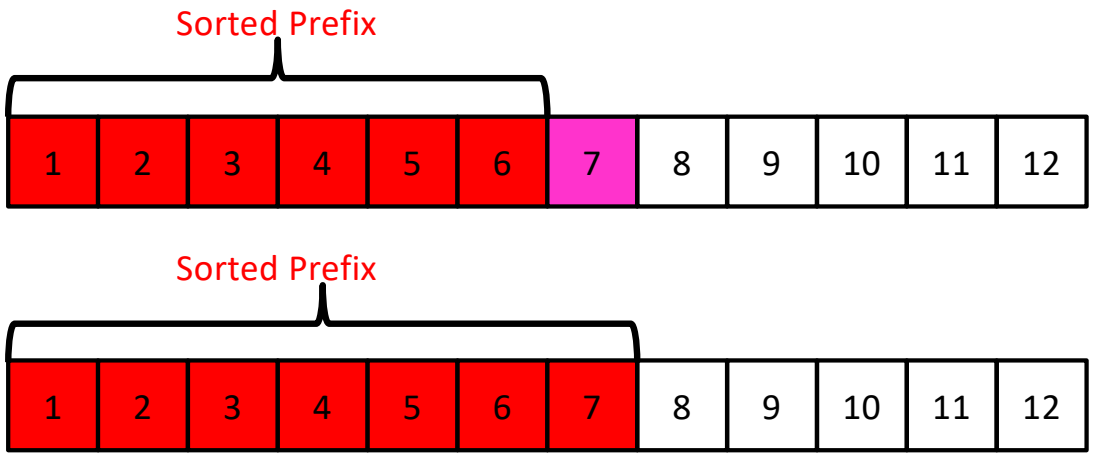
Yes!

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!

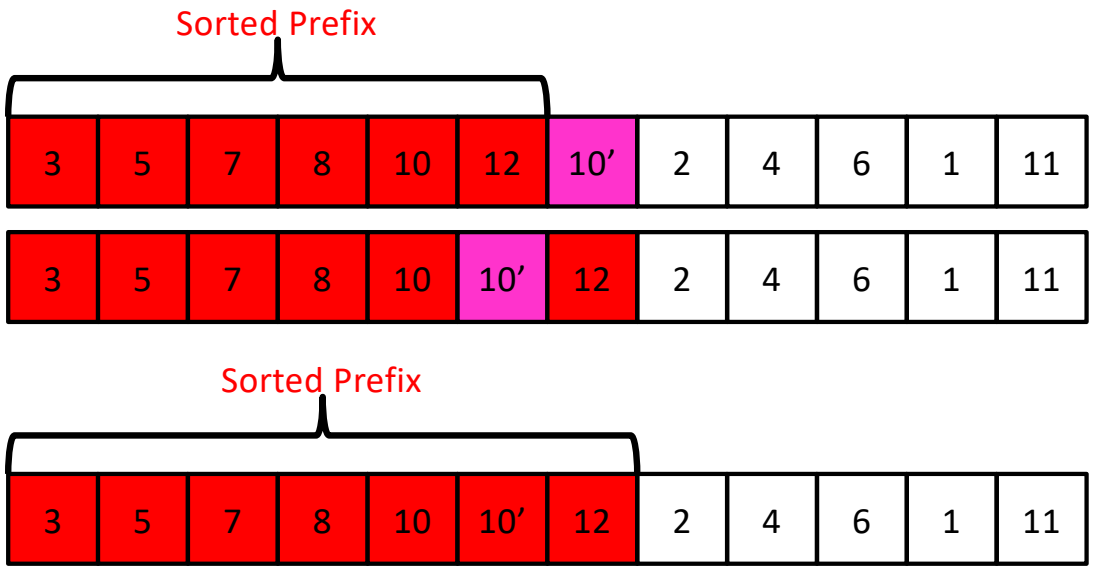
In Place? Yes!

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

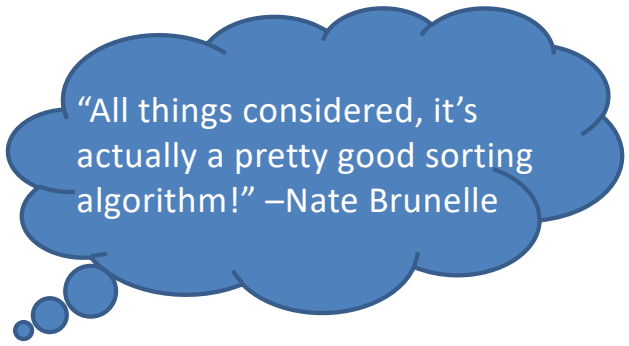
Run Time?

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

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



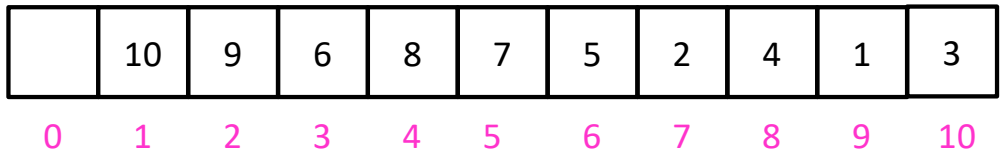
Can sort a list as it is received, i.e., don't need the entire list to begin sorting

Online?

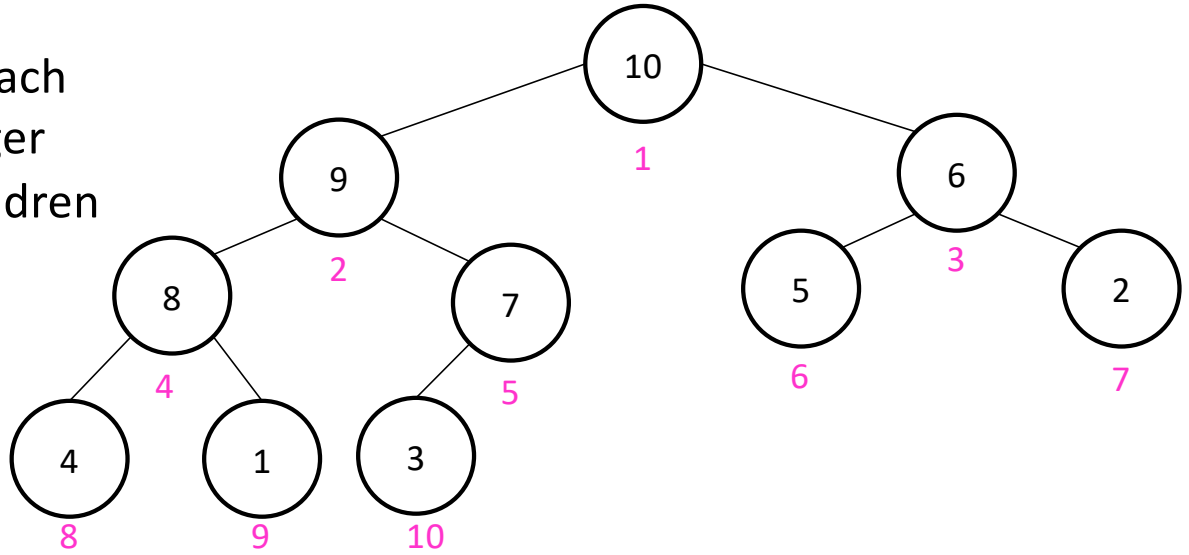
Yes

Heap Sort

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

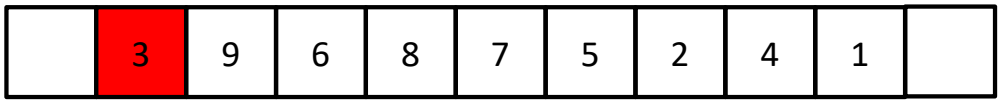


Max Heap Property: Each node is larger than its children

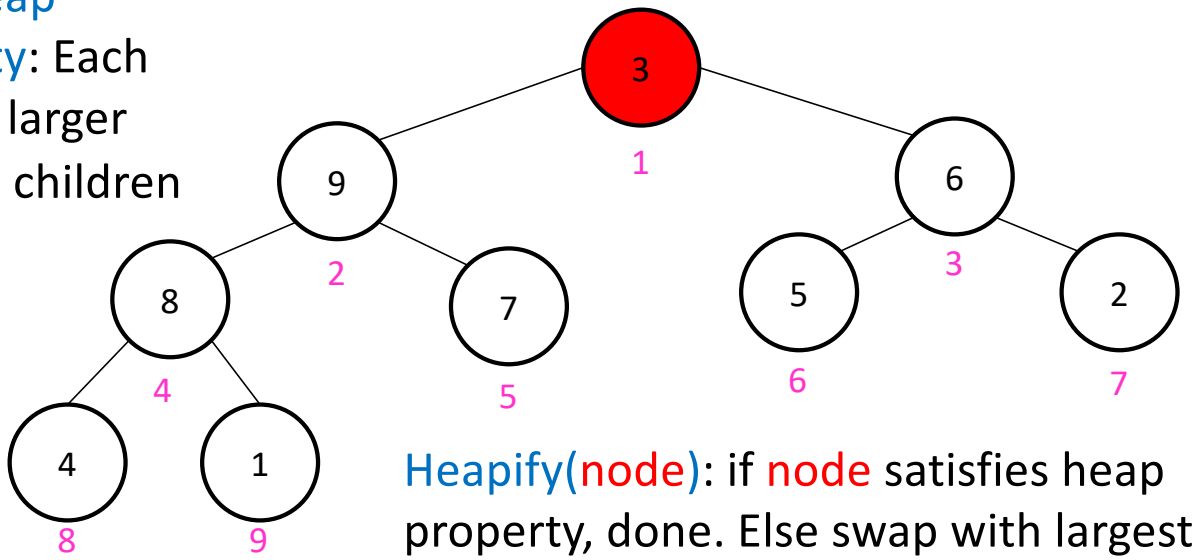


Heap Sort

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



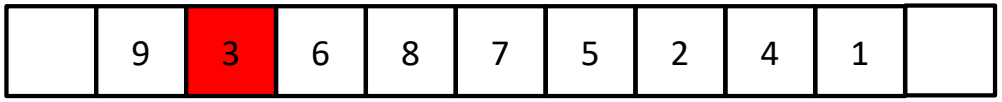
Max Heap Property: Each node is larger than its children



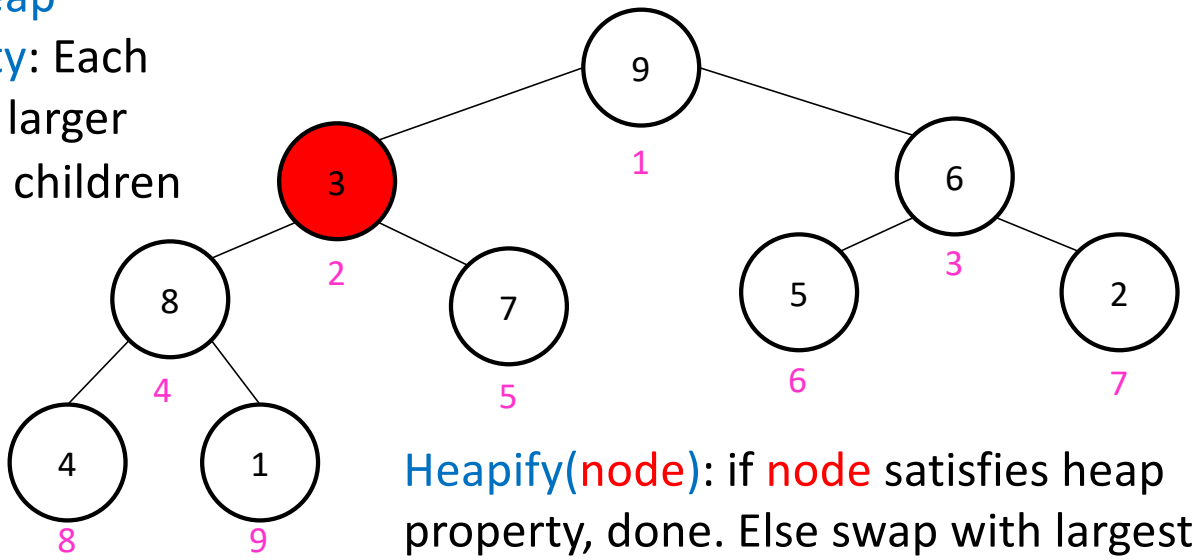
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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



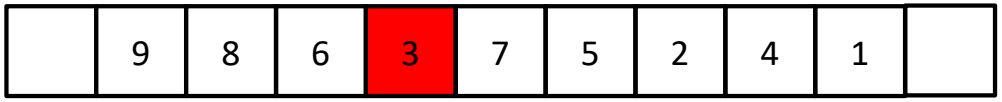
Max Heap Property: Each node is larger than its children



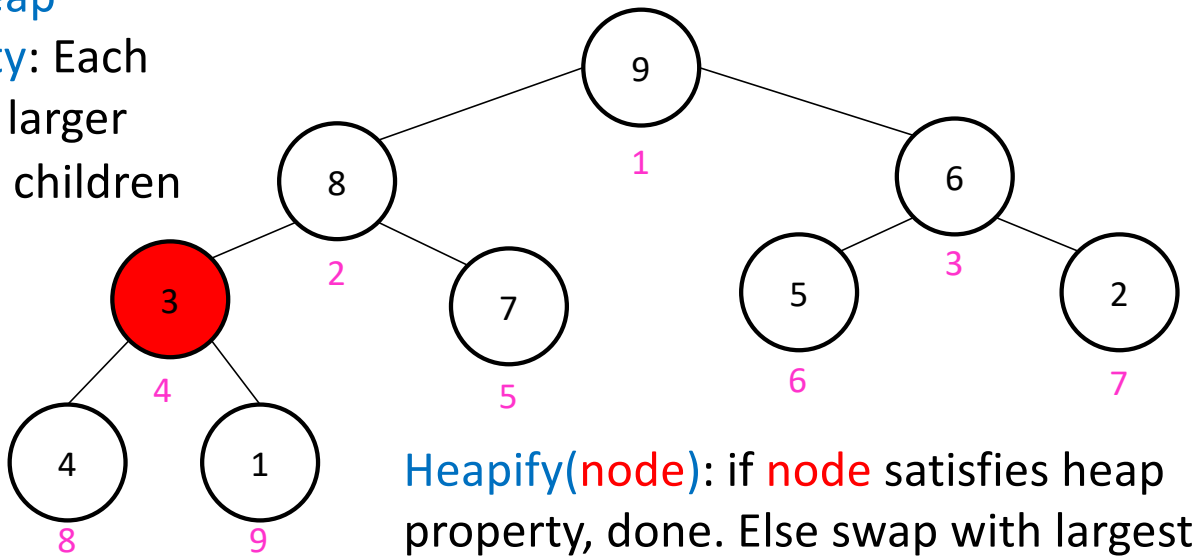
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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



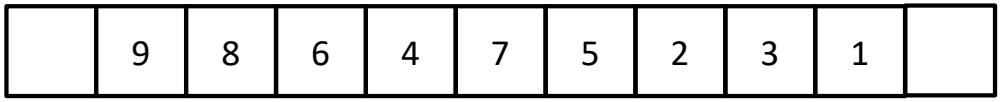
Max Heap Property: Each node is larger than its children



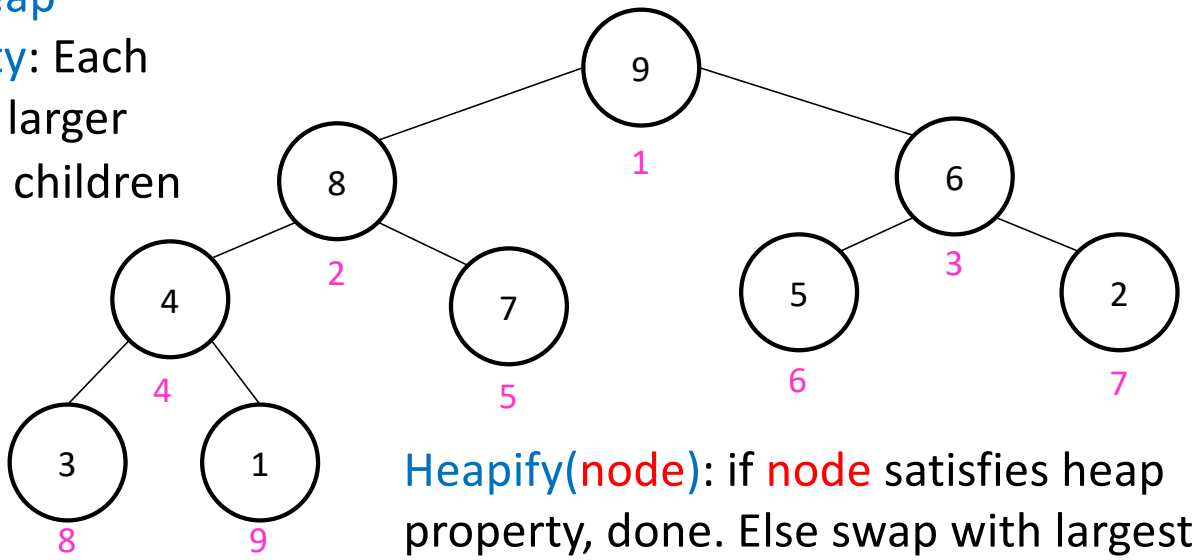
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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



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

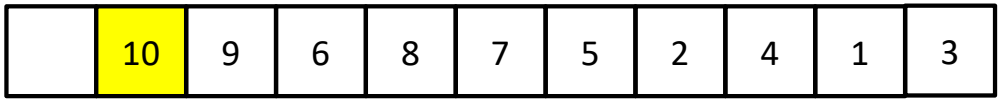
In Place?

Yes!

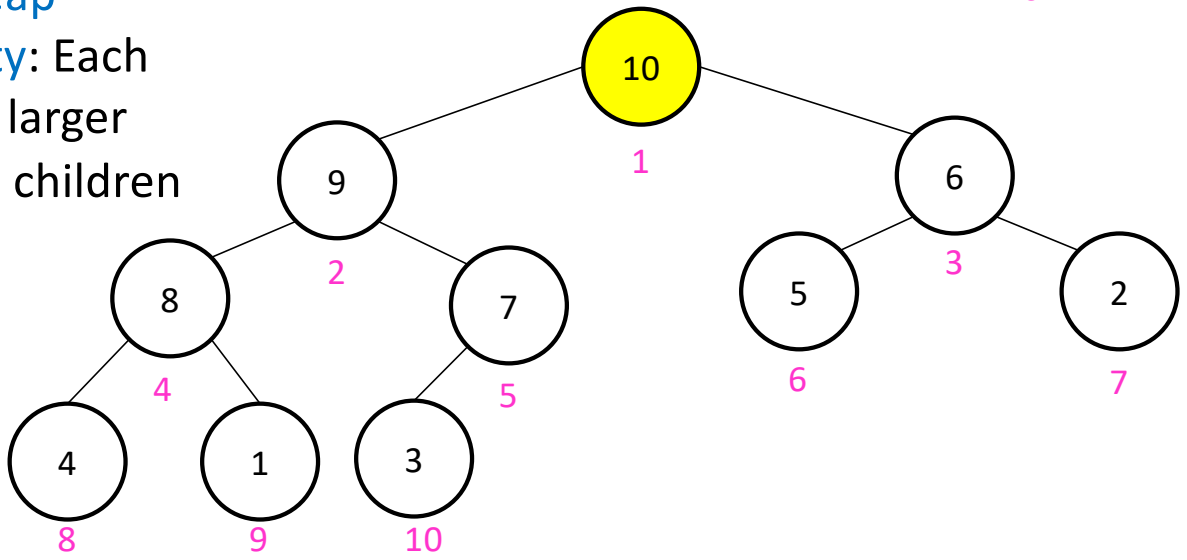
When removing an element from the heap, move it to the (now unoccupied) end of the list

In Place Heap Sort

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

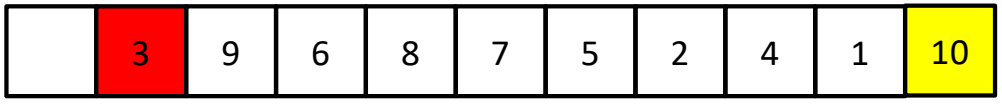


Max Heap Property: Each node is larger than its children

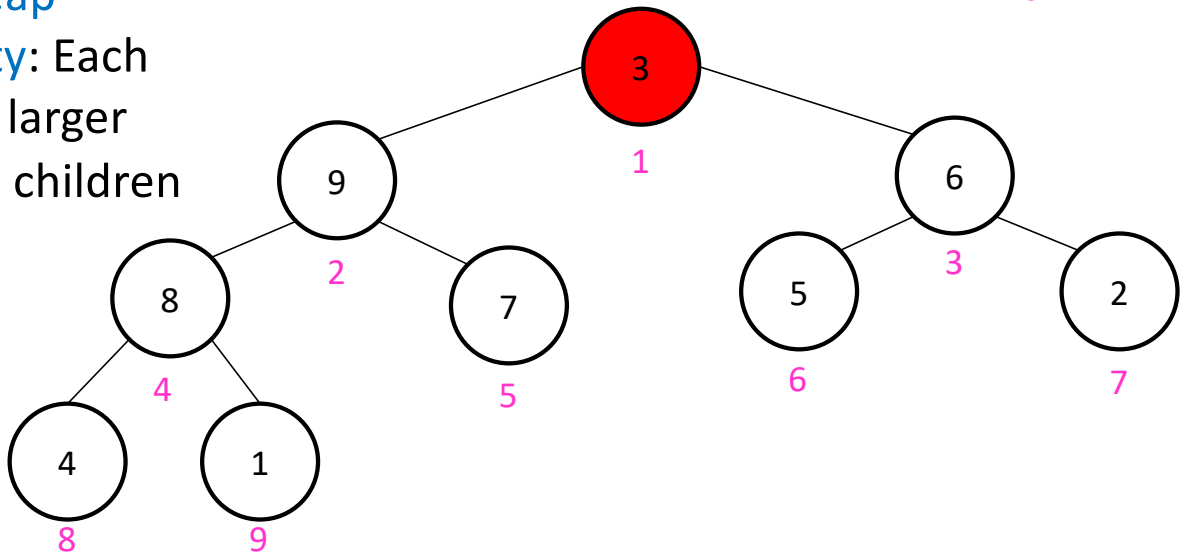


In Place Heap Sort

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

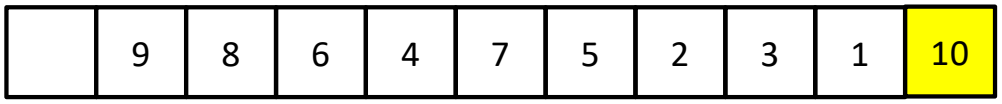


Max Heap Property: Each node is larger than its children

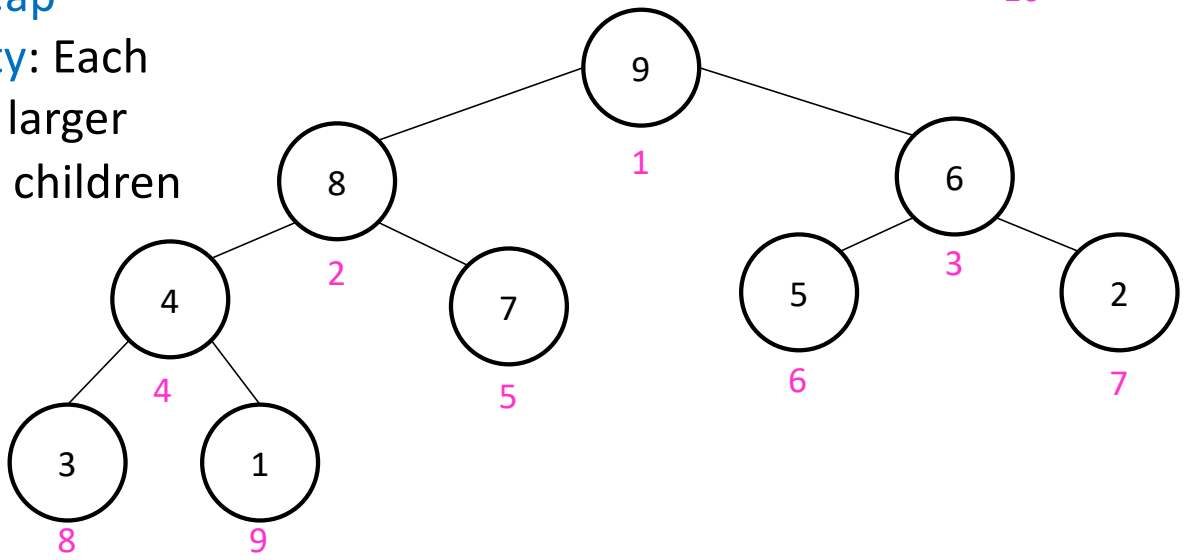


In Place Heap Sort

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

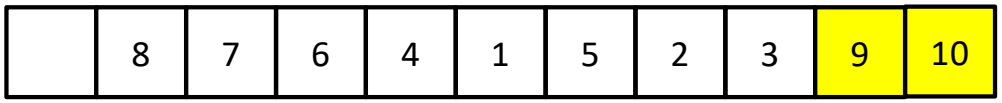


Max Heap Property: Each node is larger than its children

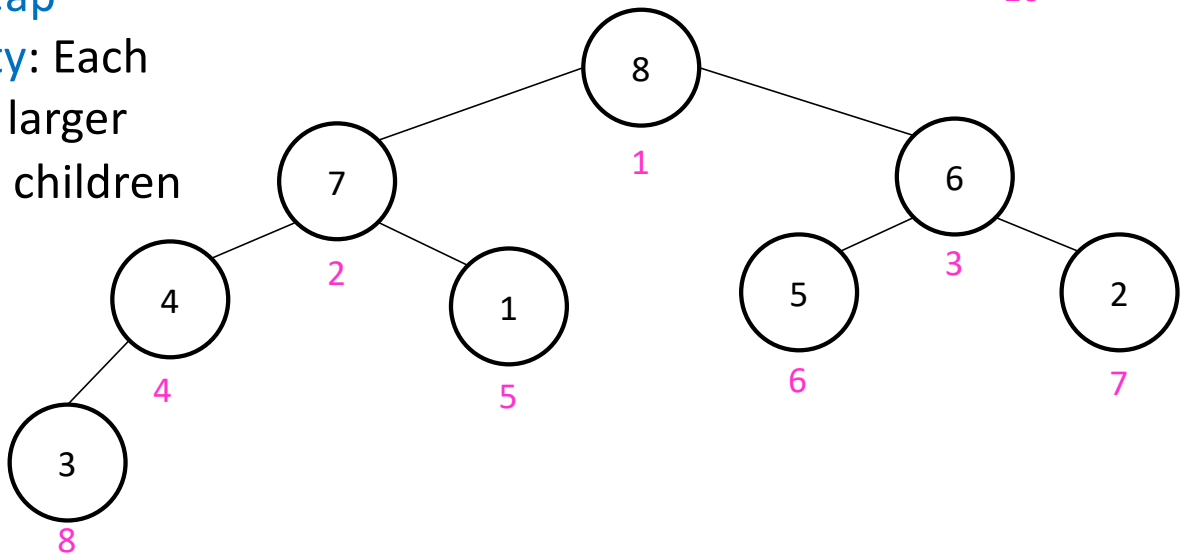


In Place Heap Sort

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

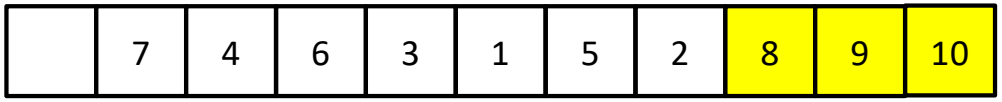


Max Heap Property: Each node is larger than its children

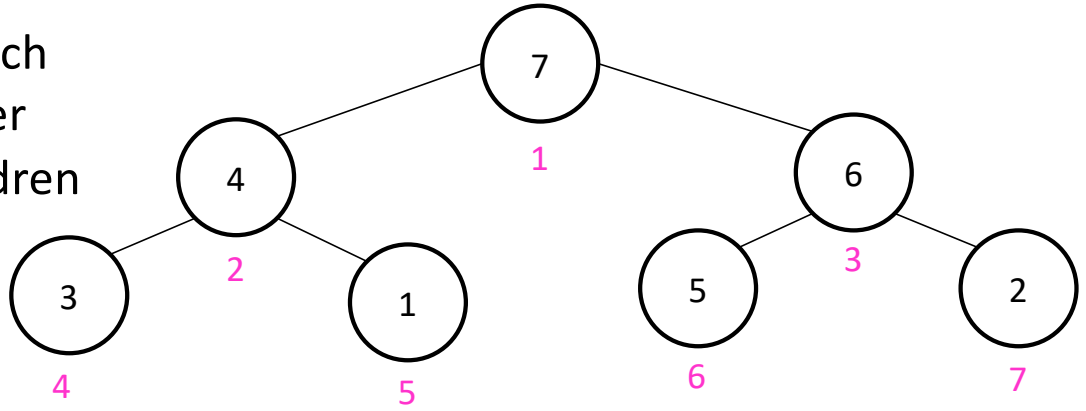


In Place Heap Sort

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



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