Warm up

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

Find Min, Lower Bound Proof

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

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 @ Rice 210 (starting next week!)
  - Thursdays 4pm-5pm @ Rice 501 (starting today!)

Today’s Keywords

- Sorting
- Linear time Sorting
- Counting Sort
- Radix Sort
- Maximum Sum Continuous Subarray

CLRS Readings

- Chapter 8

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

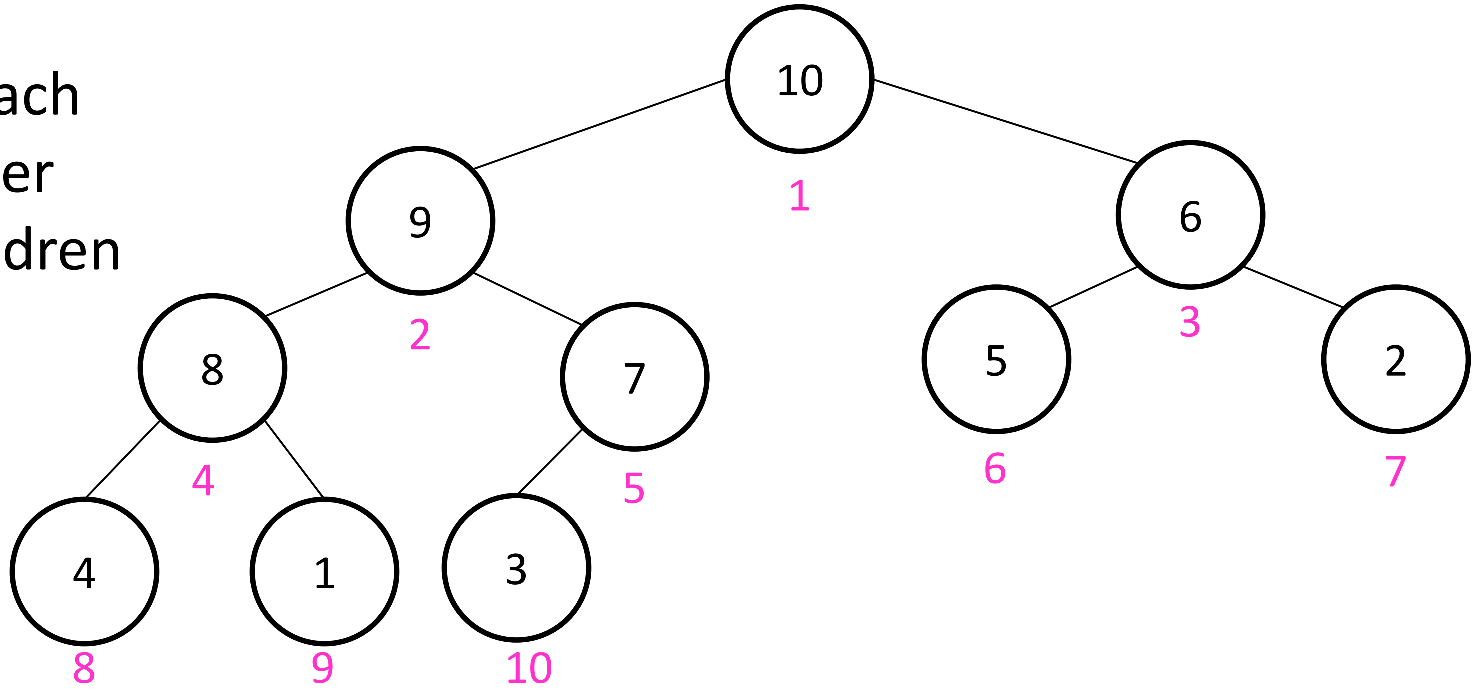
- Important properties of sorting algorithms:
- Run Time
  - Asymptotic Complexity
  - Constants
- In Place (or In-Situ)
  - Done with only constant additional space
- Adaptive
  - Faster if list is nearly sorted
- Stable
  - Equal elements remain in original order
- Parallelizable
  - Runs faster with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

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

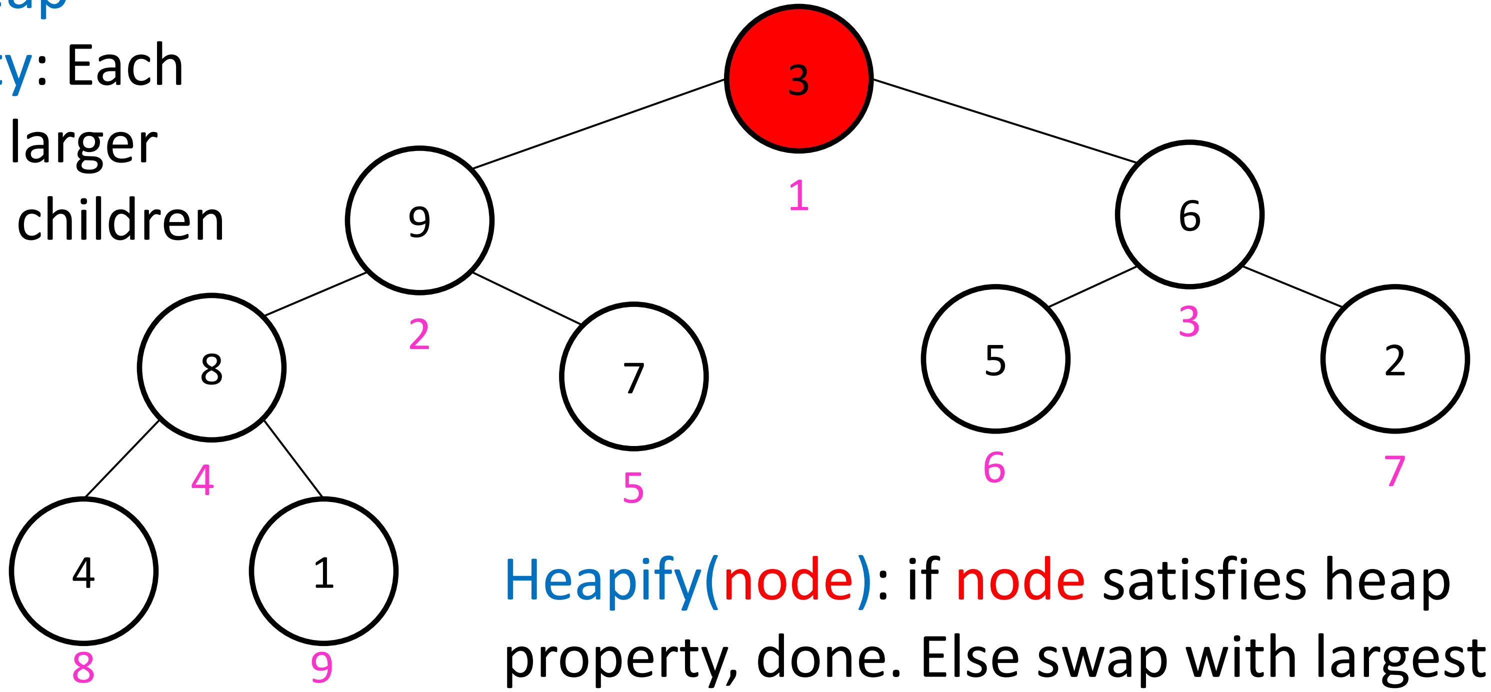


Heap Sort

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

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

Max Heap Property: Each node is larger than its children



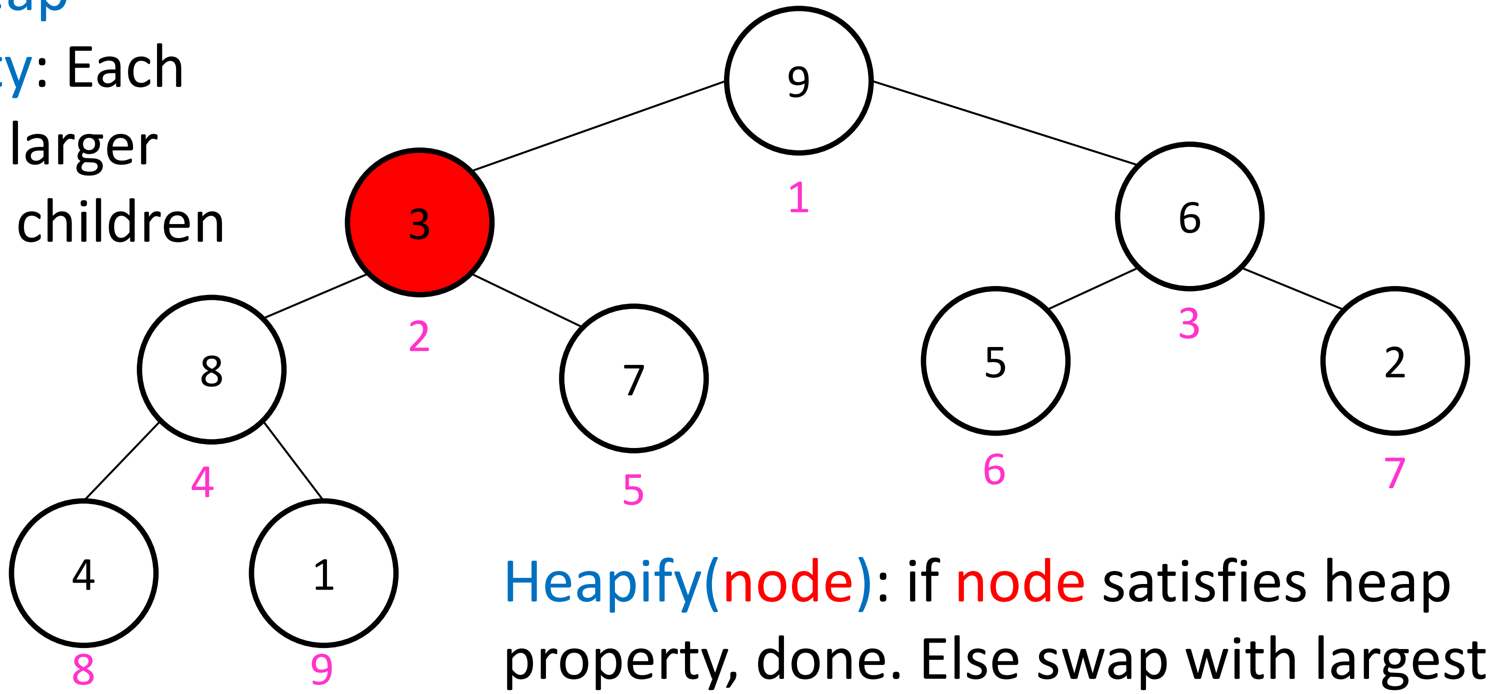
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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

	9	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

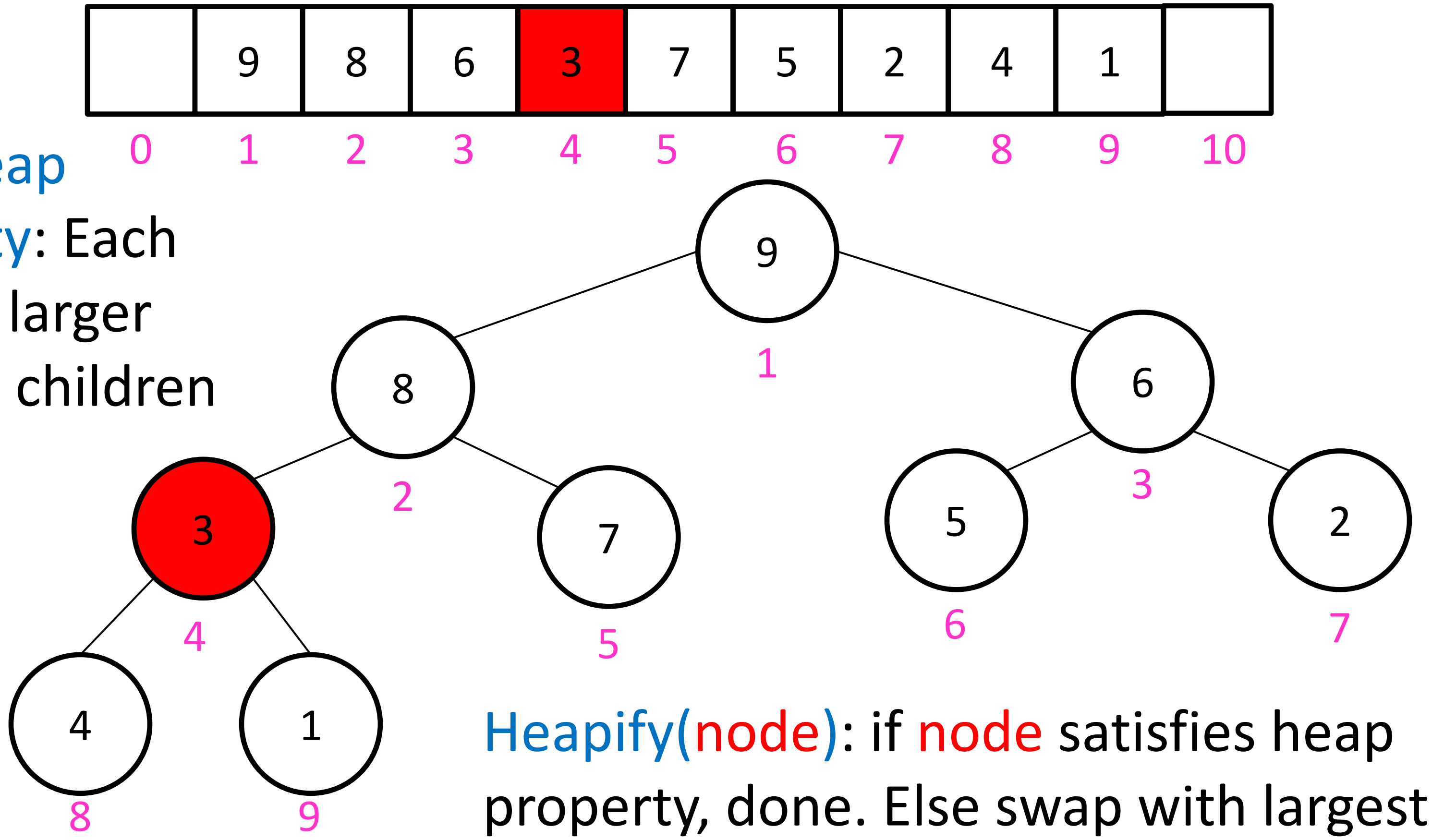


Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

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



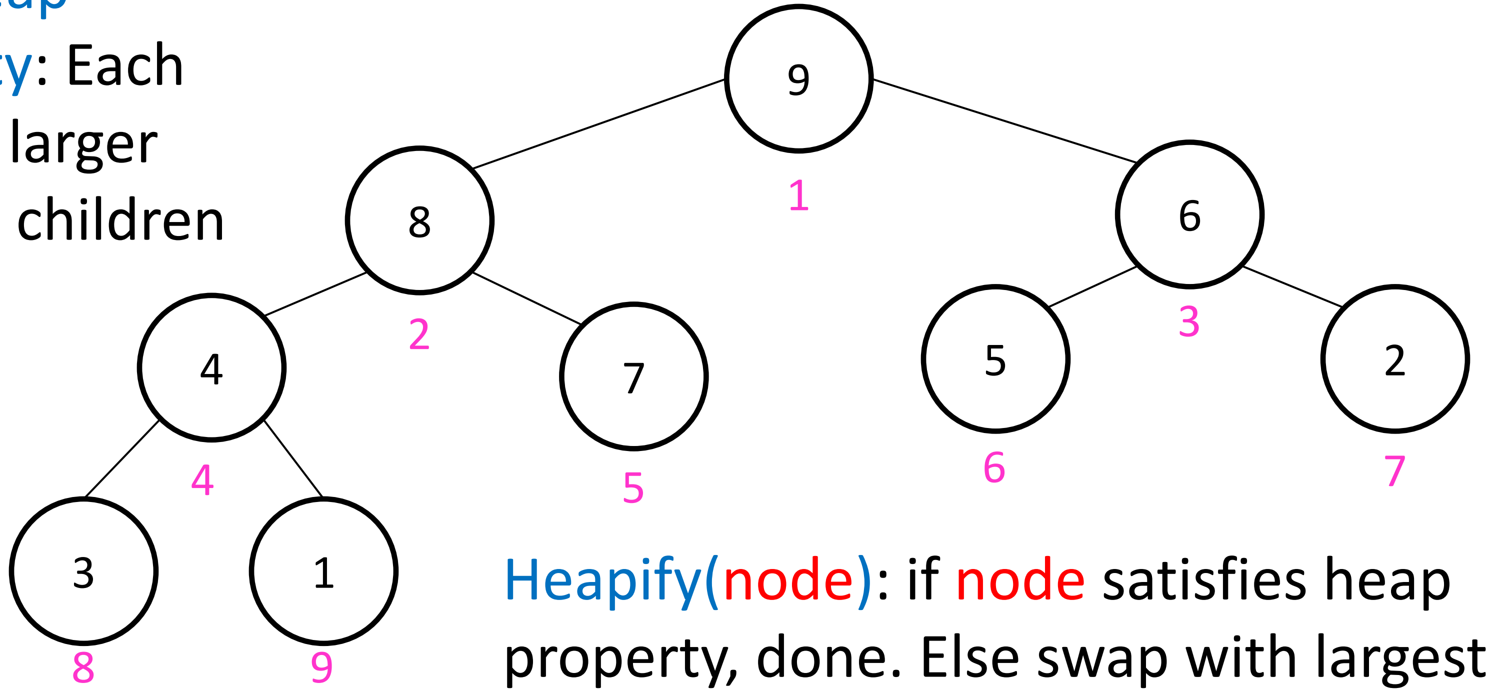
Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree

Heap Sort

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

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

- 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?

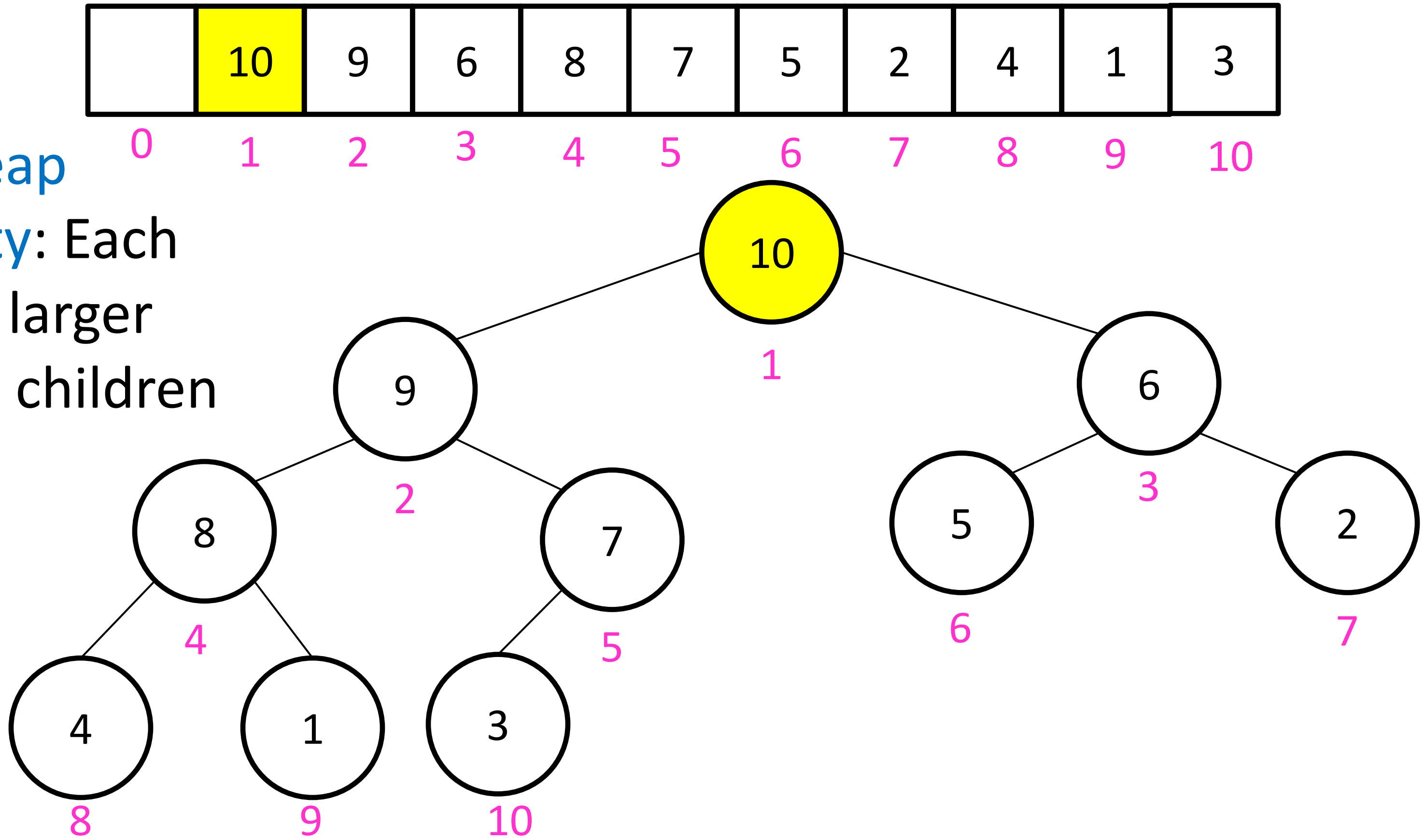
Yes!

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

In Place Heap Sort

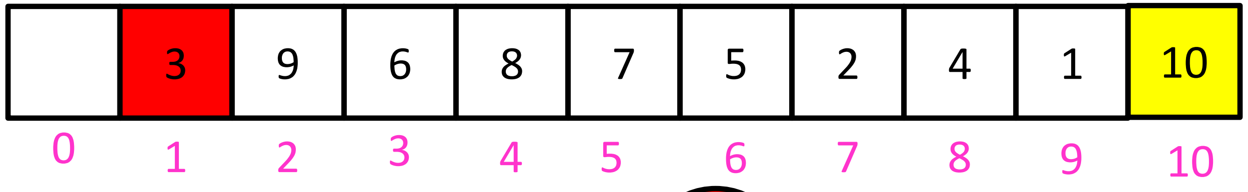
- Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

Max Heap Property: Each node is larger than its children

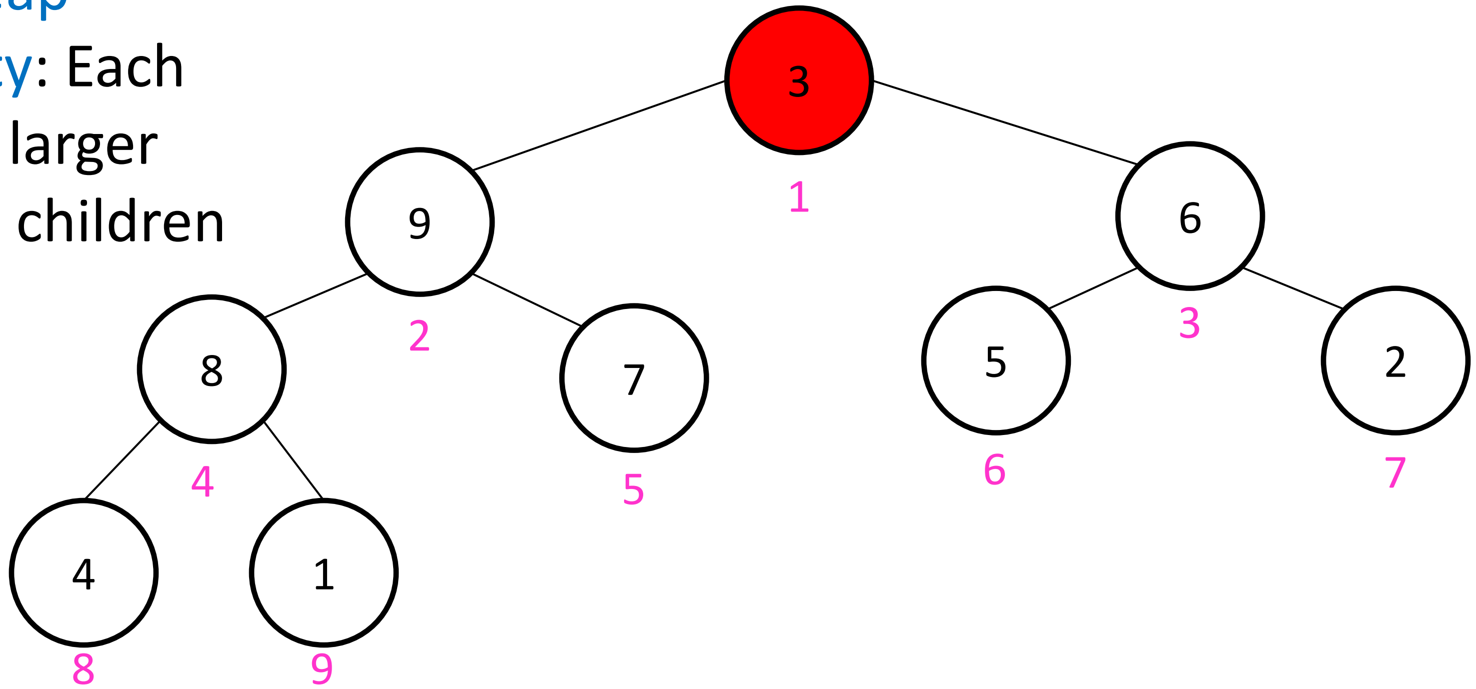


In Place Heap Sort

- 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

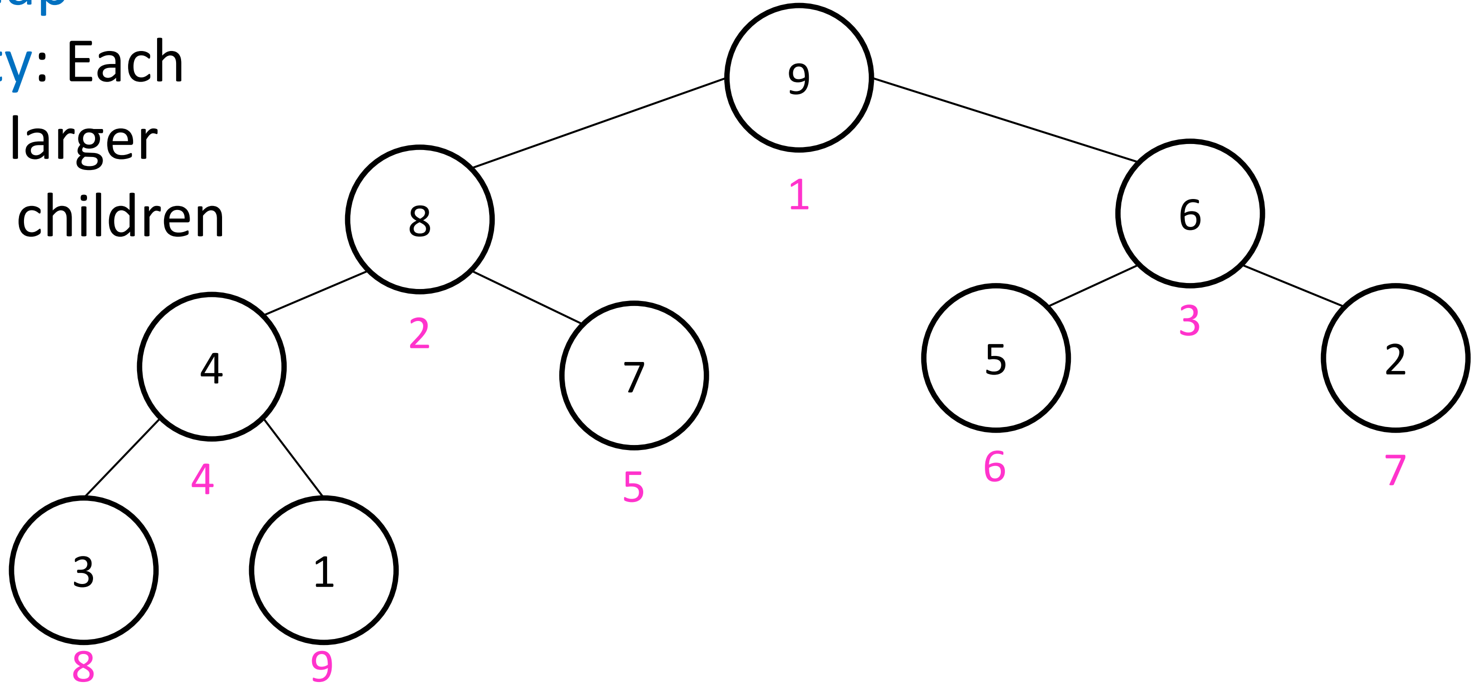


In Place Heap Sort

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

	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

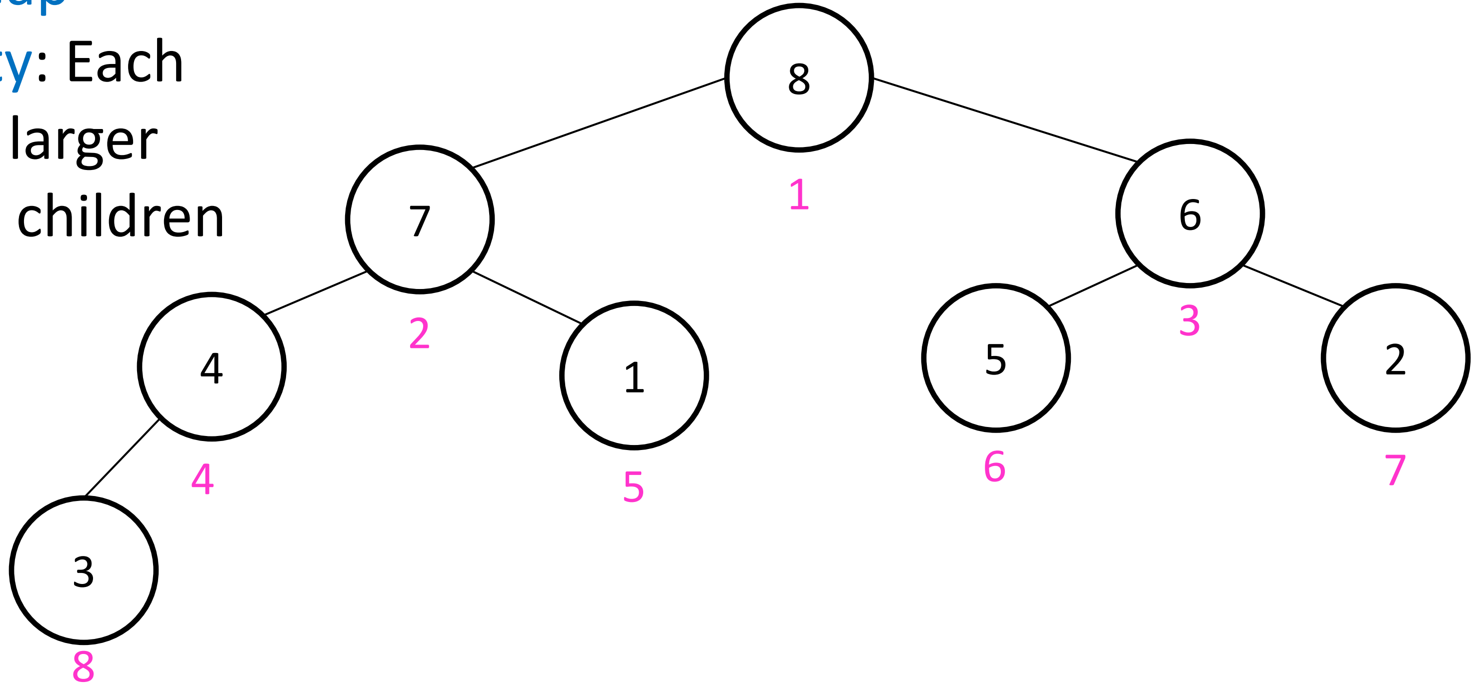


In Place Heap Sort

- 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

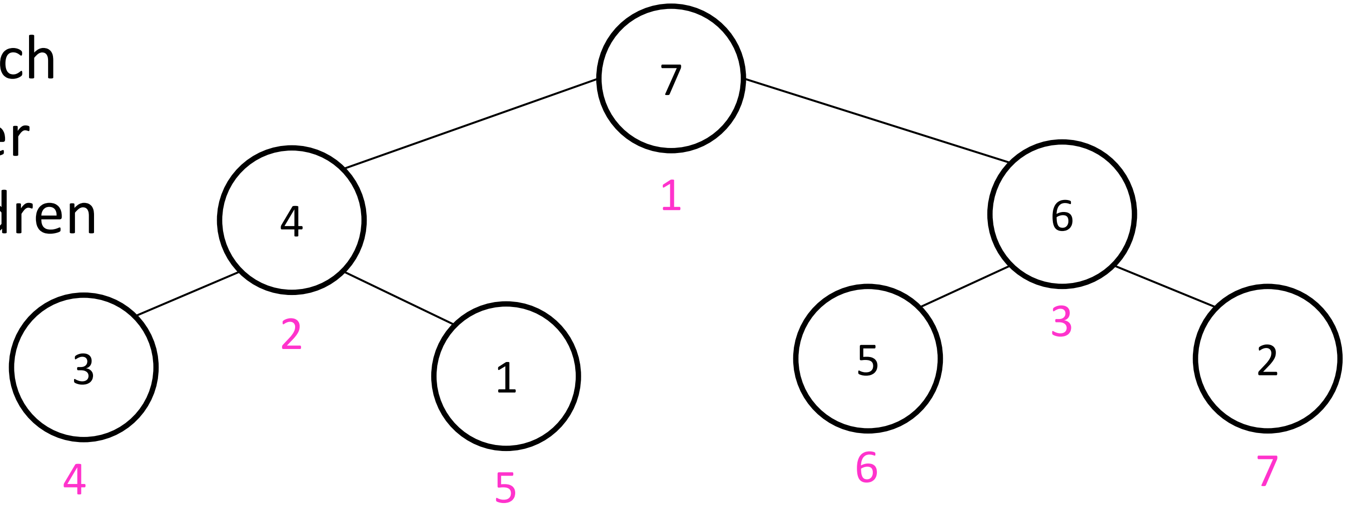


In Place Heap Sort

- 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

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!

Sorting in Linear Time

- Cannot be comparison-based
- Need to make some sort of assumption about the contents of the list
  - Small number of unique values
  - Small range of values
  - Etc.

Counting Sort

- Idea: Count how many things are less than each element

$L =$

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

1. Range is $[1, k]$ (here $[1,6]$)
   make an array C of size k
   populate with counts of each value

$C =$

2	0	2	1	0	3
1	2	3	4	5	6

For i in L:
$$++C[L[i]]$$

running sum

2. Take "running sum" of C
   to count things less than each value

$C =$

2	2	4	5	5	8
1	2	3	4	5	6

For $i = 1$ to $\text{len}(C)$:
$$C[i] = C[i-1] + C[i]$$

To sort: last item of value 3 goes at index 4

Counting Sort

- **Idea**: Count how many things are less than each element

$L =$

1	2	3	4	5	6	7	8
3	6	4	1	**3**	**6**	**1**	**6**

$C =$

1	2	3	4	5	6
2	2	4	5	5	7

Last item of value 6 goes at index 8

For each element of L (last to first):
Use C to find its proper place in B
Decrement that position of C

For $i = \text{len}(L)$ downto 1:

$$B\left[C\left[L[i]\right]\right] = L[i]$$

$$C\left[L[i]\right] = C\left[L[i]\right] - 1$$

$B =$

1	2	3	4	5	6	7	8
							6

Counting Sort

- Idea: Count how many things are less than each element

$L =$

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

$C =$

1	2	4	5	5	7
1	2	3	4	5	6

Last item of value 1 goes at index 2

For each element of L (last to first):
Use C to find its proper place in B
Decrement that position of C

For $i = \text{len}(L)$ downto 1:

$$B\left[C\left[L[i]\right]\right] = L[i]$$

$$C\left[L[i]\right] = C\left[L[i]\right] - 1$$

$B =$

	1						6
1	2	3	4	5	6	7	8

Counting Sort

- Idea: Count how many things are less than each element

$L =$

1	2	3	4	5	6	7	8
3	6	4	1	**3**	**6**	**1**	**6**

$C =$

1	2	3	4	5	6
1	2	4	5	5	6

Last item of value 6 goes at index 7

For each element of L (last to first):
Use C to find its proper place in B
Decrement that position of C

For $i = \text{len}(L)$ downto 1:

$$B\left[C\left[L[i]\right]\right] = L[i]$$

$$C\left[L[i]\right] = C\left[L[i]\right] - 1$$

$B =$

1	2	3	4	5	6	7	8
	1					**6**	**6**

Run Time: $O(n + k)$

Memory: $O(n + k)$

Counting Sort

- Why not always use counting sort?
- For 64-bit numbers, requires an array of length $2^{64} > 10^{19}$
  - 5 GHz CPU will require > 116 years to initialize the array
  - 18 Exabytes of data
    - Total amount of data that Google has

12 Exabytes

Radix Sort

- Idea: Stable sort on each digit, from least significant to most significant

103	801	401	323	255	823	999	101	113	901	555	512	245	800	018	121
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Place each element into a "bucket" according to its 1's place

800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999
0	1	2	3	4	5	6	7	8	9

Radix Sort

- Idea: Stable sort on each digit, from least significant to most significant

Place each element into a “bucket” according to its 10’s place

0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

Radix Sort

- Idea: Stable sort on each digit, from least significant to most significant

800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999
0	1	2	3	4	5	6	7	8	9

Place each element into a “bucket” according to its 100’s place

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

Run Time: $O\big(d(n+b)\big)$

d = digits in largest value

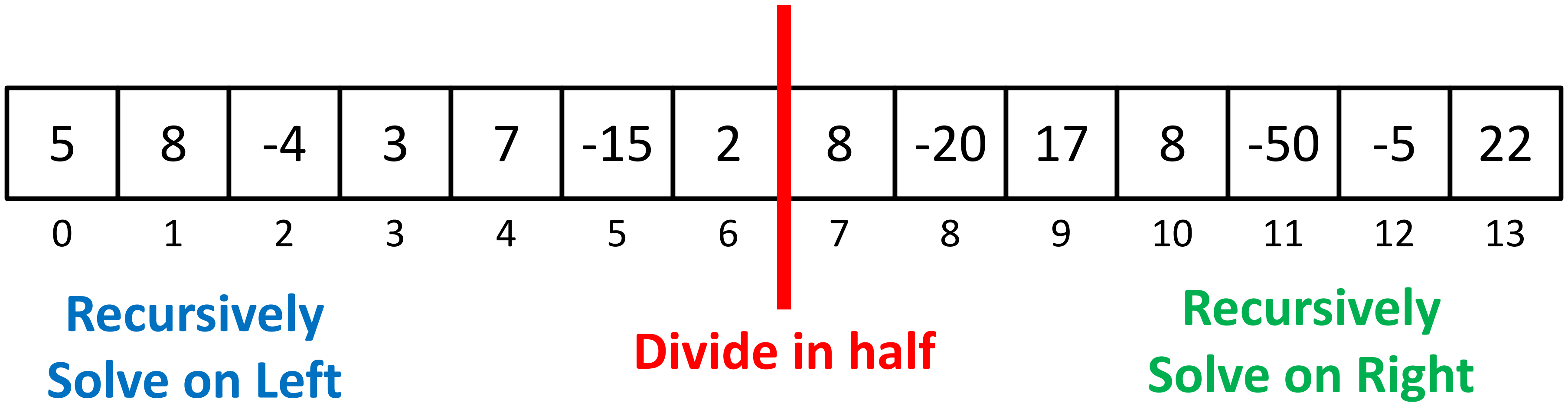
b = base of representation

Maximum Sum Contiguous Subarray Problem

The maximum-sum subarray of a given array of integers A is the interval $[a, b]$ such that the sum of all values in the array between a and b inclusive is maximal.

Given an array of n integers (may include both positive and negative values), give a $O(n \log n)$ algorithm for finding the maximum-sum subarray.

Divide and Conquer $\Theta(n \log n)$



Divide and Conquer $\Theta(n \log n)$

Largest sum that ends here + Largest sum that starts here

Largest sum	6	1	-7	-3	-6	-13	2	8	-12	5	13	-37	-42	-20
Value	5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively Solve on Left
19

Divide in half

Find Largest sum that spans the cut

Recursively Solve on Right
25

Divide and Conquer $\Theta(n \log n)$

Return the Max of

Left, Right, Center

6	1	-7	-3	-6	-13	2	8	-12	5	13	-37	-42	-20
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively Solve on Left
19

Divide in half

Find Largest sum that spans the cut
19

Recursively Solve on Right
25

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

Divide and Conquer Summary

Typically multiple subproblems.
Typically all roughly the same size.

- Divide
  - Break the list in half
- Conquer
  - Find the best subarrays on the left and right
- Combine
  - Find the best subarray that “spans the divide”
  - I.e. the best subarray that ends at the divide concatenated with the best that starts at the divide

Generic Divide and Conquer Solution

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

MSCS Divide and Conquer $\Theta(n \log n)$

```
def MSCS(list):
    if list.length < 2:
        return list[0]      #list of size 1 the sum is maximal
    {listL, listR} = Divide (list)
    for list in {listL, listR}:
        subSolutions.append(MSCS(list))
    solution = max(solnL, solnR, span(listL, listR))
    return solution
```

Types of "Divide and Conquer"

- Divide and Conquer
  - Break the problem up into several subproblems of roughly equal size, recursively solve
  - E.g. Karatsuba, Closest Pair of Points, Mergesort…
- Decrease and Conquer
  - Break the problem into a single smaller subproblem, recursively solve
  - E.g. Impossible Missions Force (Double Agents), Quickselect, Binary Search

Pattern So Far

- Typically looking to divide the problem by some fraction (½, ¼ the size)
- Not necessarily always the best!
  - Sometimes, we can write faster algorithms by finding unbalanced divides.