

CS4102 Algorithms

Spring 2020 – Horton's Slides

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

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)
- Midterm: March 4 (two weeks away!)
- Regrade Office Hours
 - Fridays 2:30pm-3:30pm (Rice 210)

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

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?

No

Adaptive?

No

Stable?

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?

No

Adaptive?

No

Stable?

Yes!
(usually)

Parallelizable?

Yes!

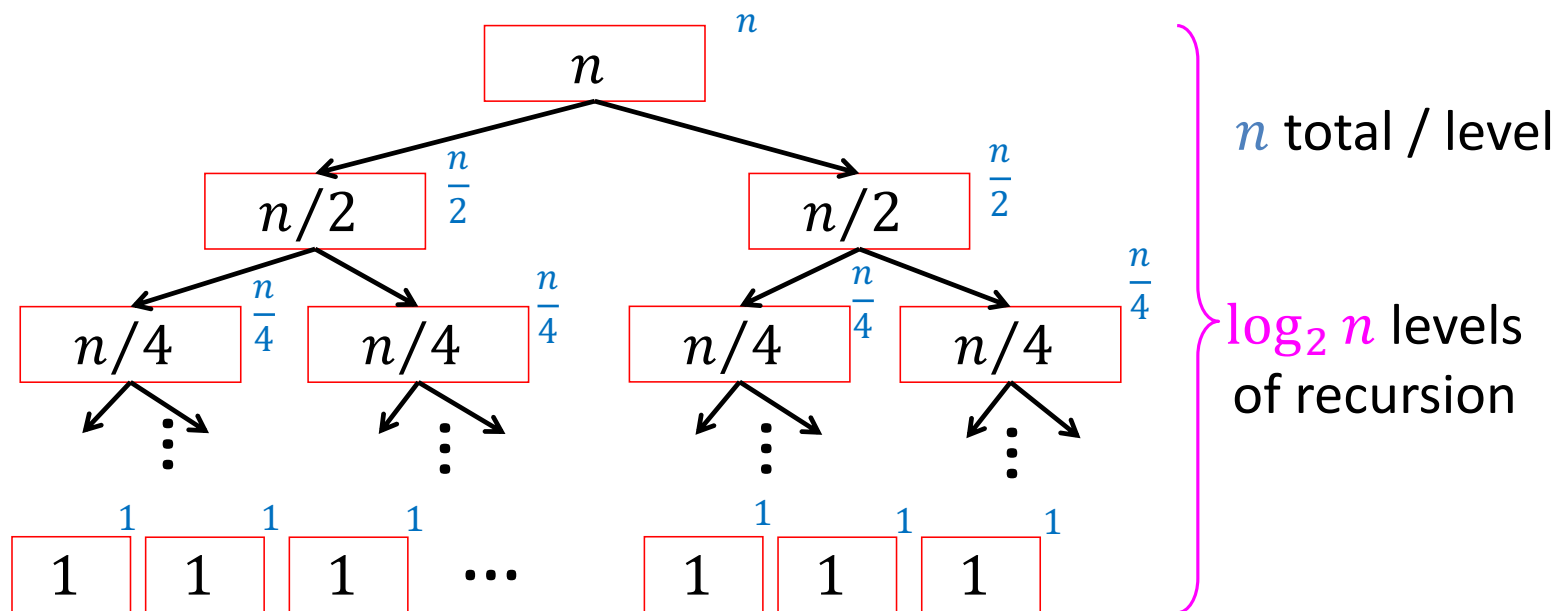
Mergesort

Parallelizable:
Allow different
machines to work
on each sublist

- **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

Mergesort (Sequential)

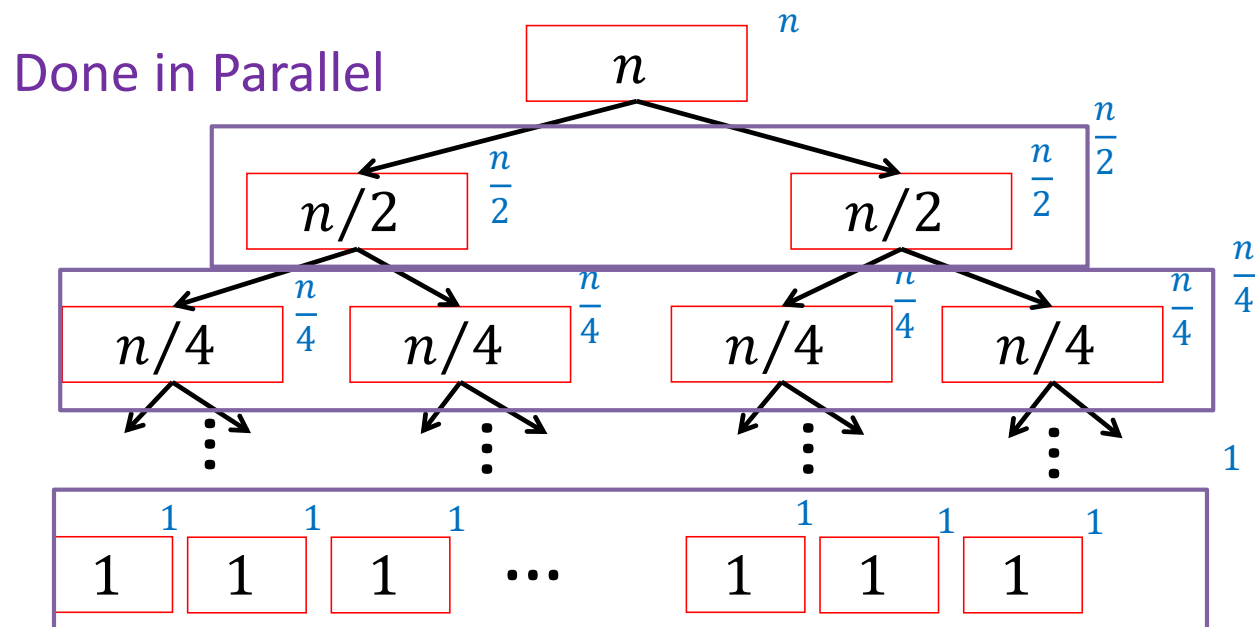
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



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

Mergesort (Parallel)

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



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

Adaptive?

No!

Stable?

No

Parallelizable?

Yes!

Uses stack for
recursive calls

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?

Yes!

Adaptive?

~~Kinda~~

Not really

Stable?

Yes

Parallelizable?

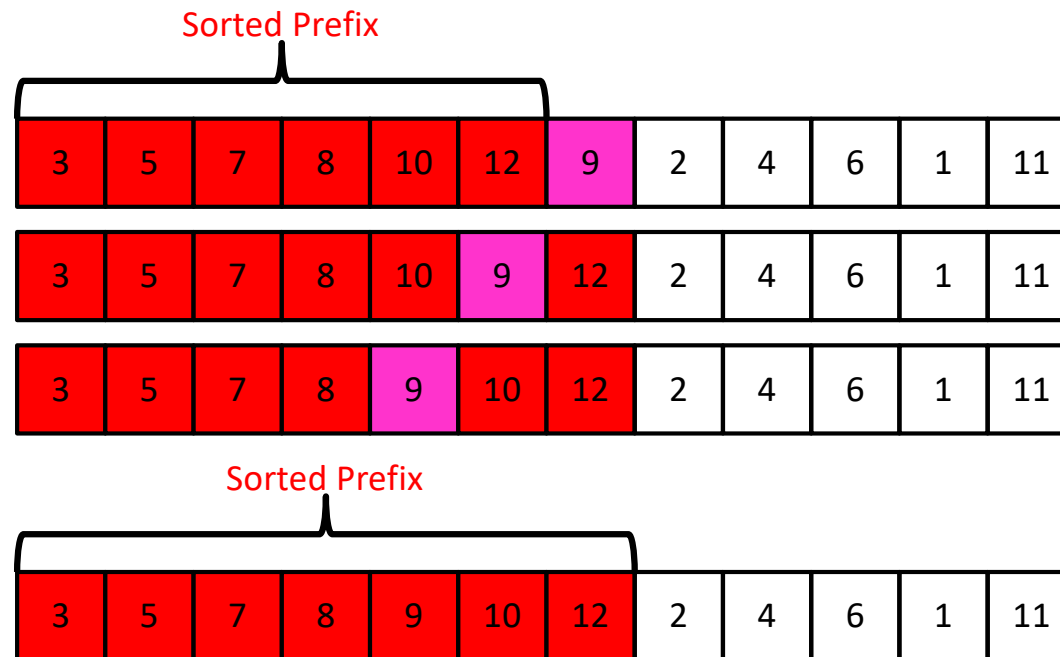
No

"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**

In Place?

Yes!

Adaptive?

Yes

Run Time?

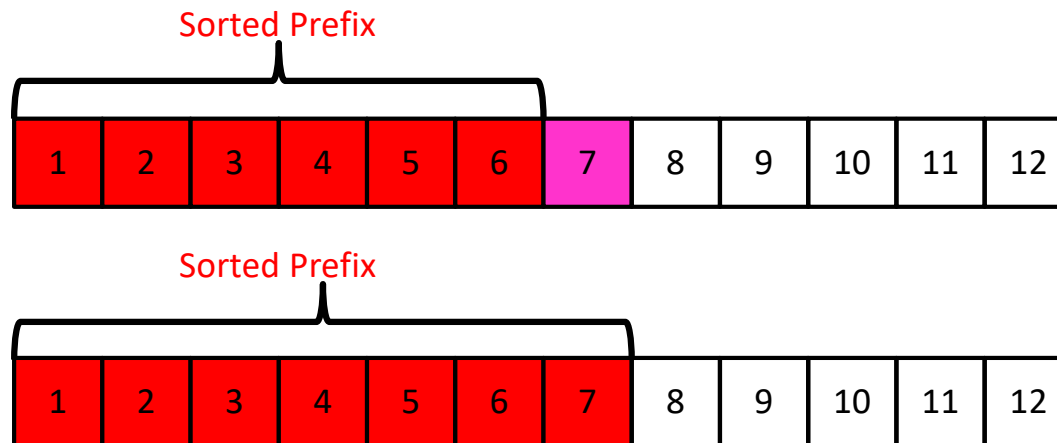
$\Theta(n^2)$

(but with very small constants)

Great for short lists!

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?

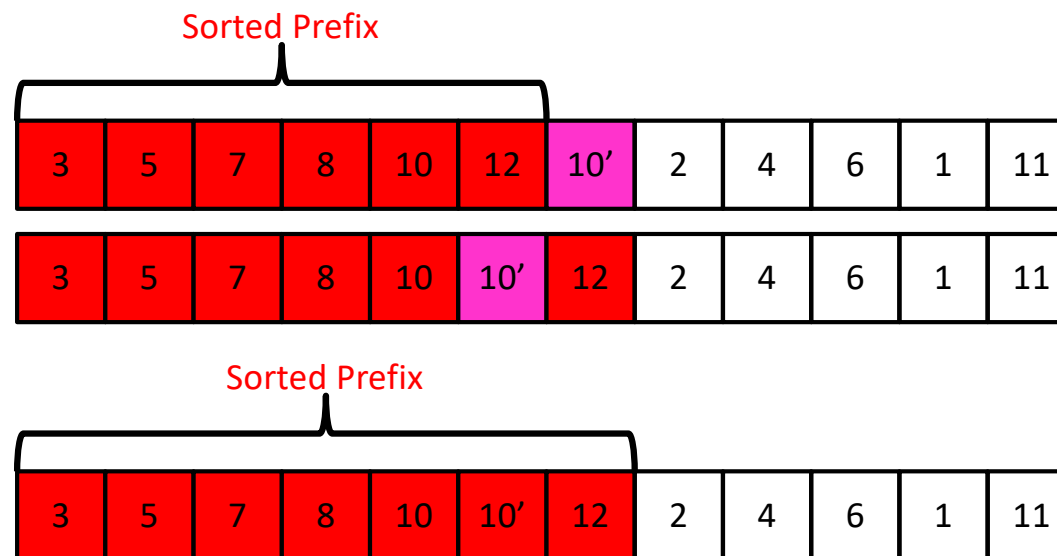
Yes

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?

Yes!

Adaptive?

Yes

Stable?

Yes

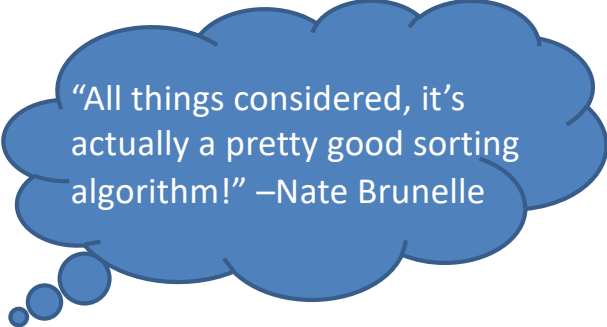
Parallelizable?

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

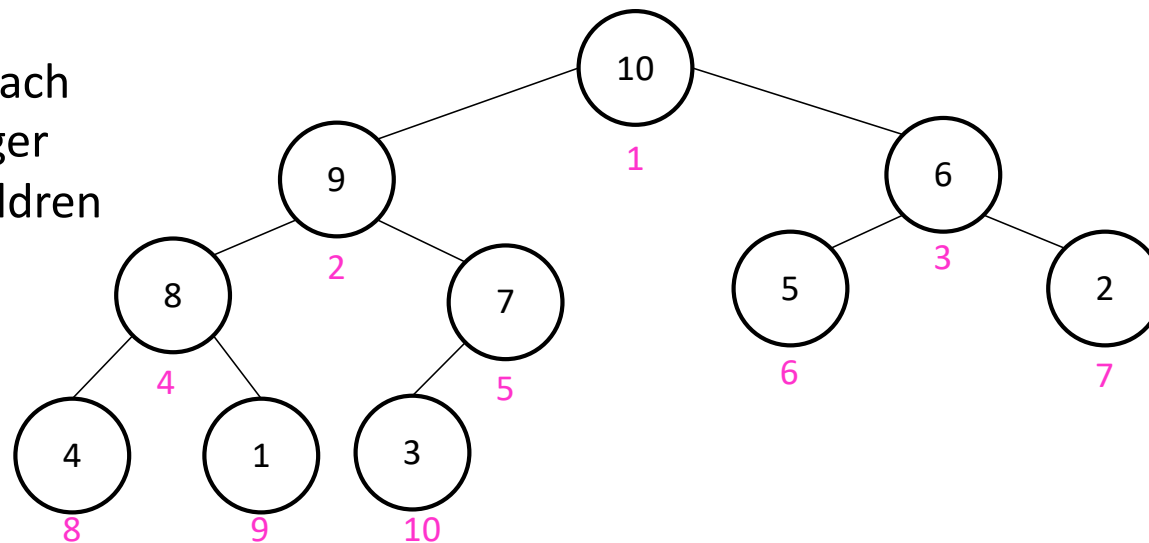
Heap Sort

- **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 |

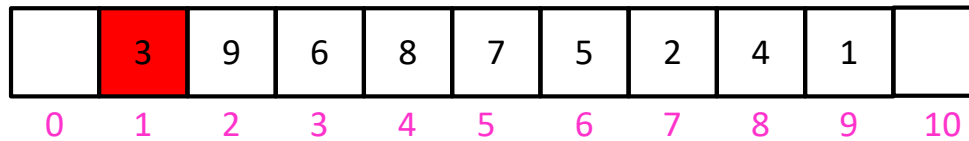
Max Heap

Property: Each node is larger than its children



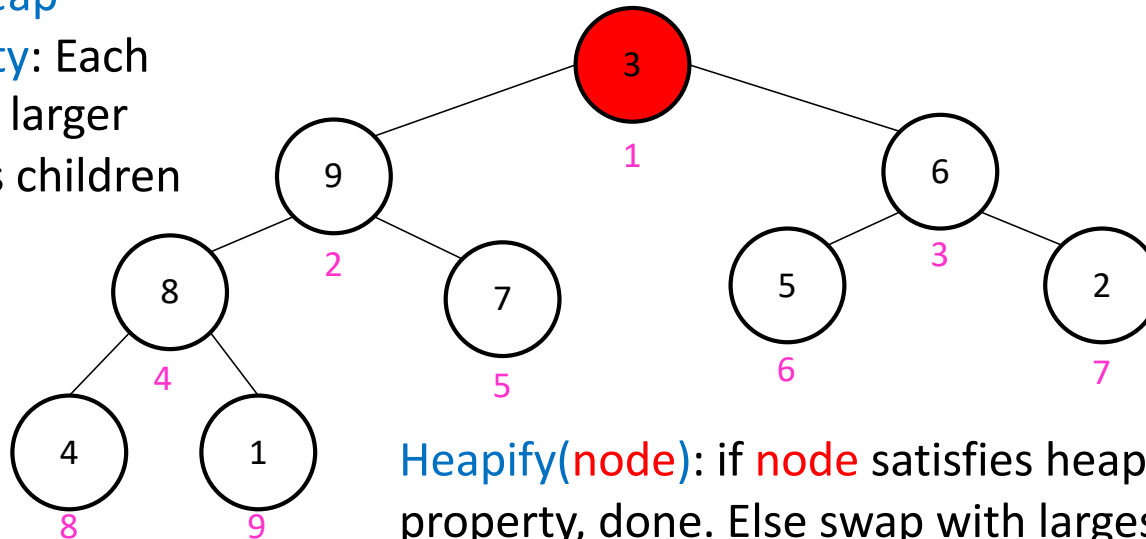
Heap Sort

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



Max Heap

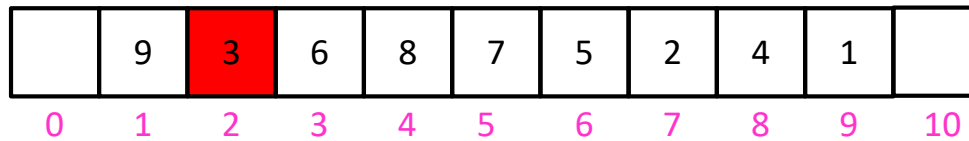
Property: Each node is larger than its children



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

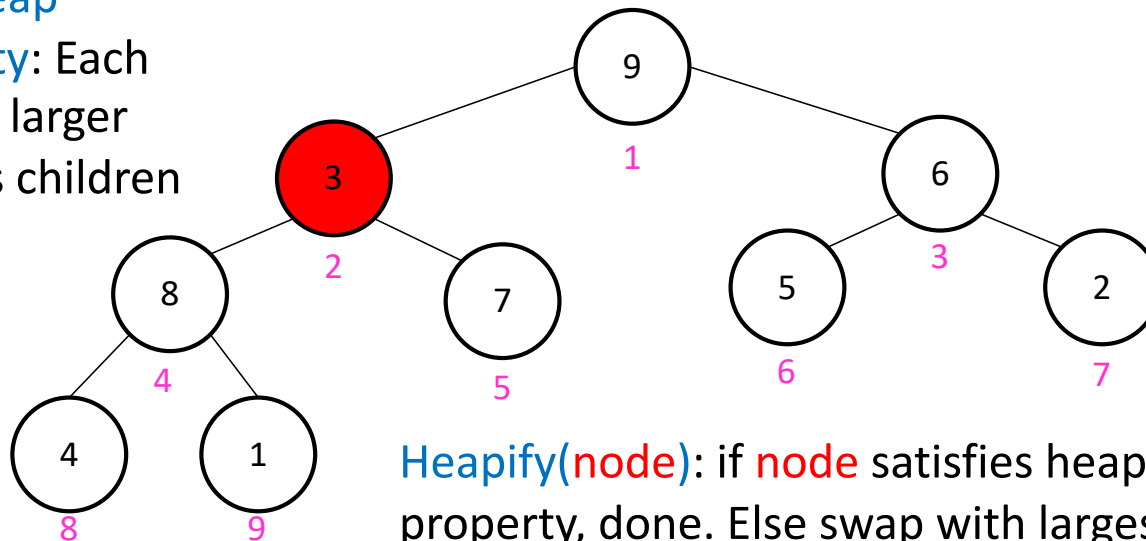
Heap Sort

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



Max Heap

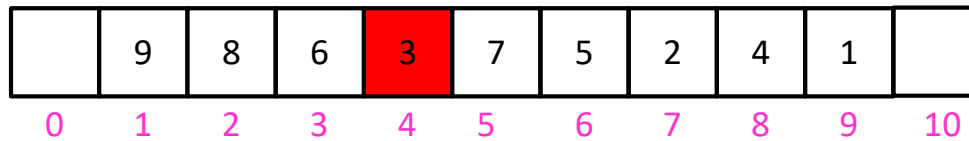
Property: Each node is larger than its children



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

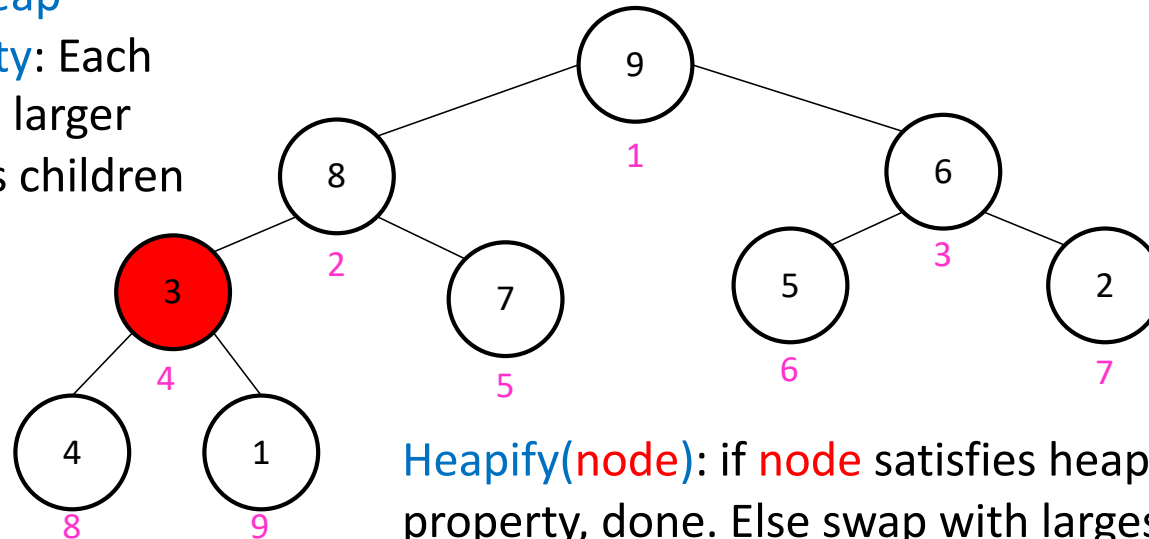
Heap Sort

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



Max Heap

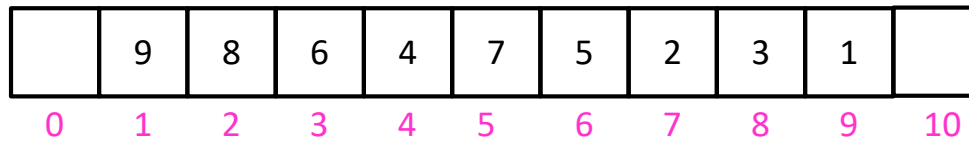
Property: Each node is larger than its children



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

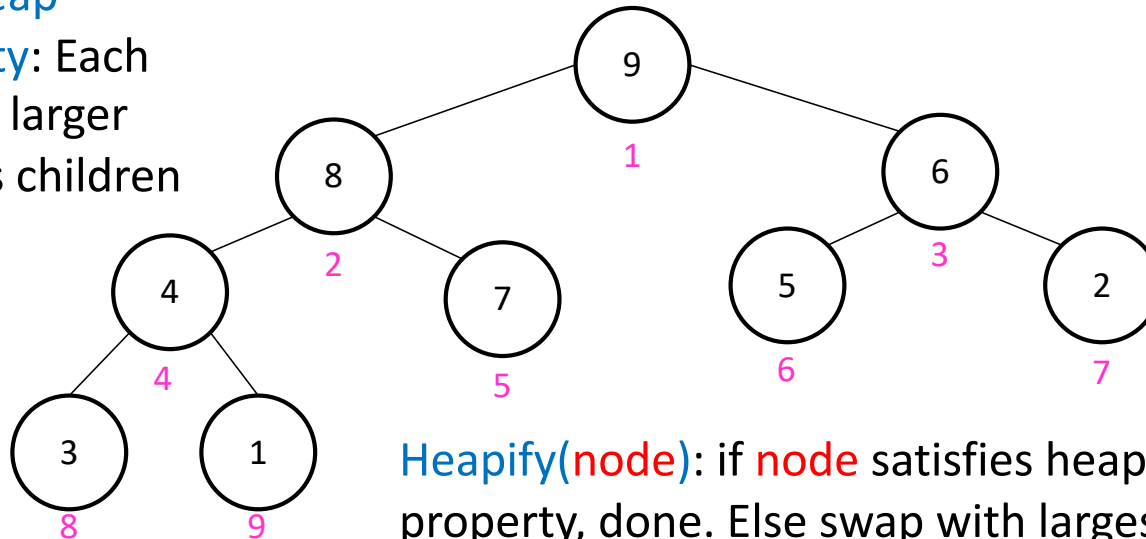
Heap Sort

- 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

In Place?

Yes!

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

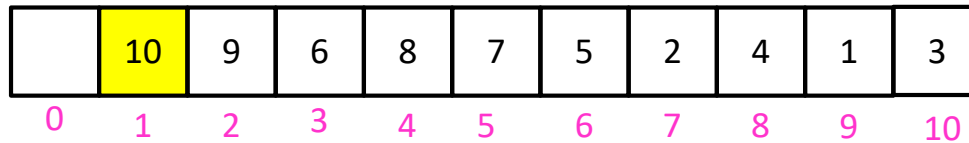
Run Time?

$\Theta(n \log n)$

Constants worse than Quick Sort

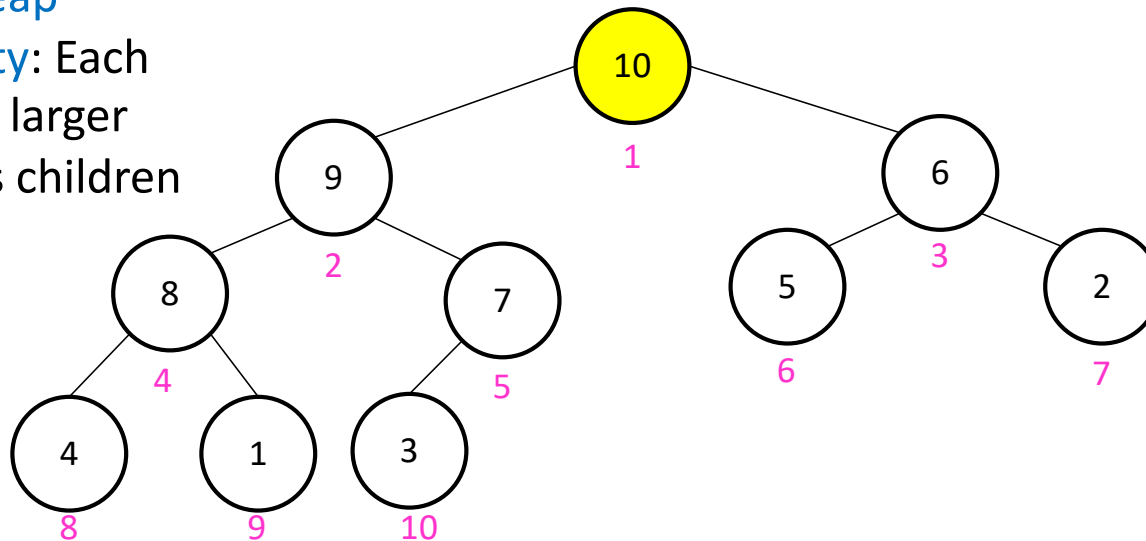
In Place Heap Sort

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



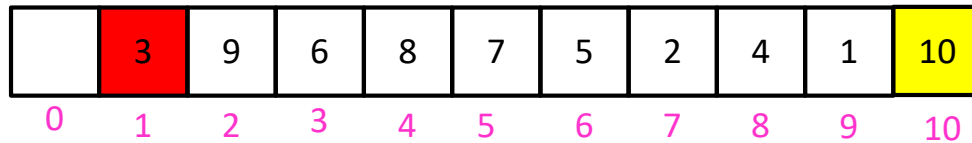
Max Heap

Property: Each node is larger than its children



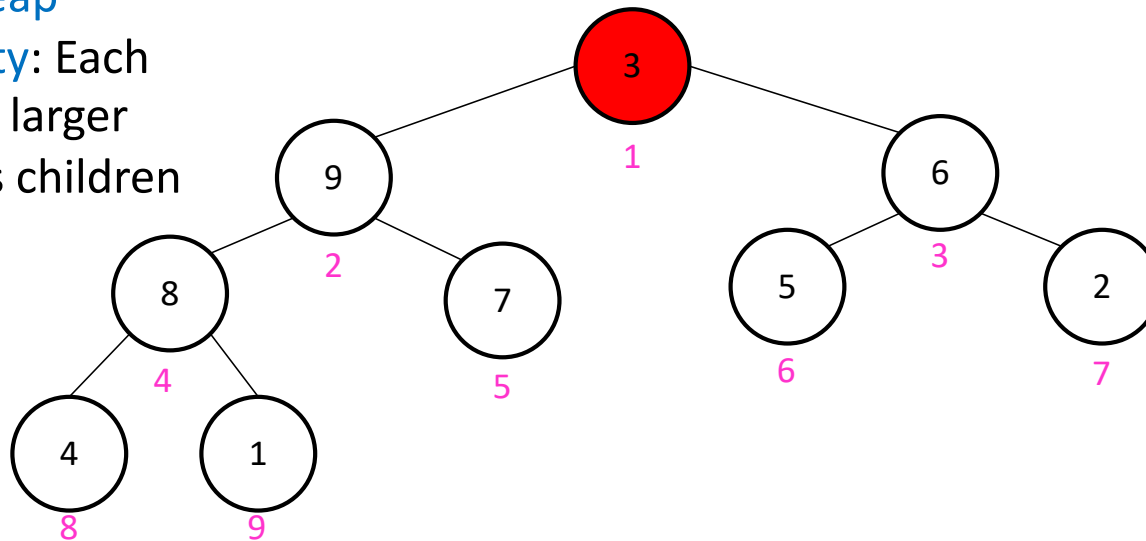
In Place Heap Sort

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



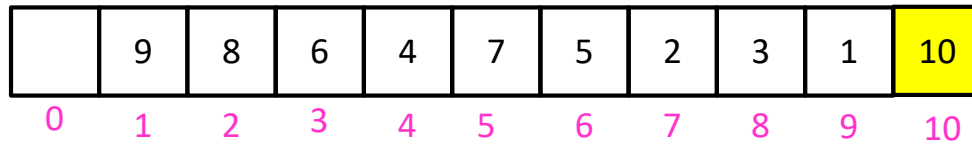
Max Heap

Property: Each node is larger than its children



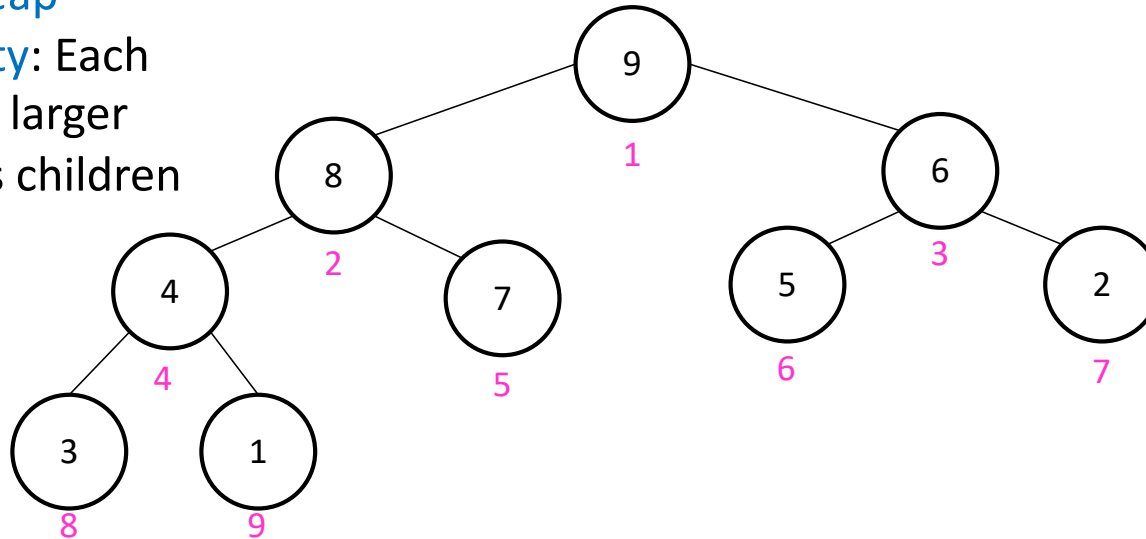
In Place Heap Sort

- **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



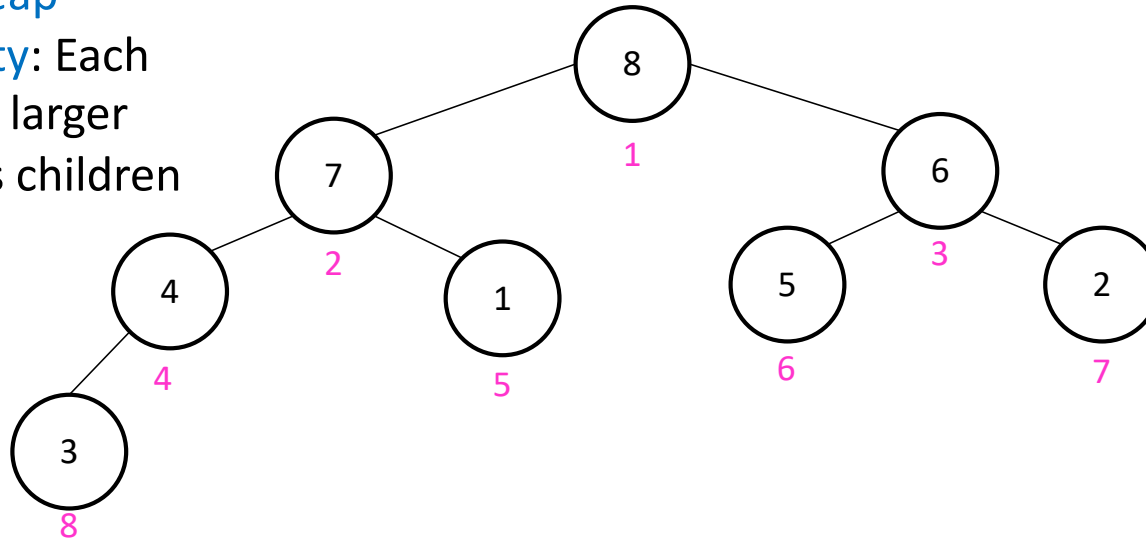
In Place Heap Sort

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|
| | 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



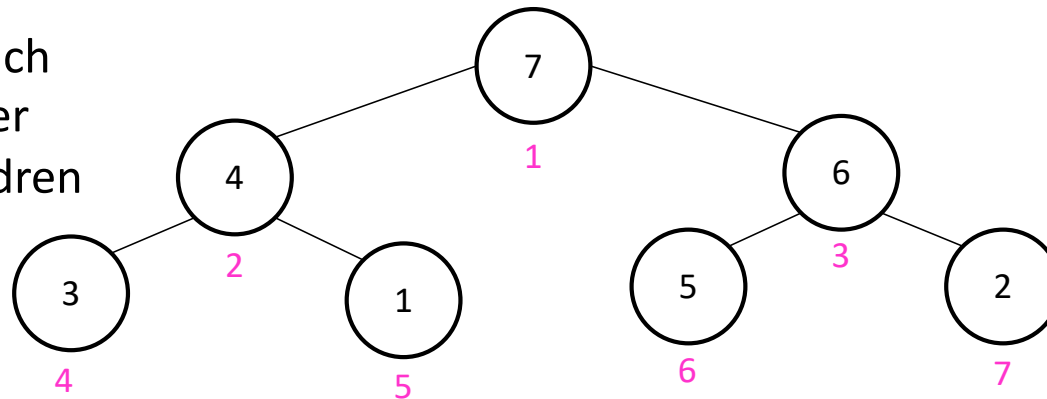
In Place Heap Sort

- **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

Parallelizable?

In Place?

Yes!

Adaptive?

No

Stable?

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

- Sometimes we can sort in linear time!
 - Wait, what? We used decision trees to prove sorting is $\Omega(n \log n)$
 - Remember: proof assumed key-comparison was our basic operation
- Thus, if we can do something more than just compare two keys, then...
 - Similar situation: binary search is optimal, but hashing can be faster
- Possible approach: make some sort of assumption about the contents of the list
 - Small number of unique values
 - Small range of values
 - Etc.
- Cannot be comparison-based! We see examples that use a key's numeric value.

Counting Sort

- Starting point: Determine how often each element occurs

$L =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 6 | 1 | 3 | 4 | 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 :
 $++C[L[i]]$

2. We could easily use C to produce a list of sorted key values in a list B .
Do you see how? (Discuss.)
That's sorting, isn't it, so that's all we need, right? (Answer: no.)

Counting Sort

- **First Idea:** Use **index** and **Count** to create output list of key values

$L =$

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

$C =$

| | | | | | |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 1 | 0 | 3 |
| 1 | 2 | 3 | 4 | 5 | 6 |

```
b_idx = 1 // next position in output list B
for i = 1 to len(C): // look at count for next value in range
  for j = 1 to C[i]: // for each time i occurs...
    B[b_idx] = i // ...put value i into output list
    b_idx = b_idx + 1
```

Complexity: $\Theta(\max(n, k)) = \Theta(n + k)$

What's wrong with this approach?

- **Data associated with keys? Stable?**

| Priority | Associated Data |
|----------|----------------------------------|
| 2 | Data, 1st with pri=2 |
| 1 | Data, 1 st with pri=1 |
| 2 | Data, 2 nd with pri=2 |
| 1 | Data, 2 nd with pri=1 |
| ... | ... |

Counting Sort

- Better Idea: Count how many things are \leq each element

$L =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 6 | 1 | 3 | 4 | 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

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

$C =$

| | | | | | |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 1 | 0 | 3 |
| 1 | 2 | 3 | 4 | 5 | 6 |

running sum



$C =$

| | | | | | |
|---|---|---|---|---|---|
| 2 | 2 | 4 | 5 | 5 | 8 |
| 1 | 2 | 3 | 4 | 5 | 6 |

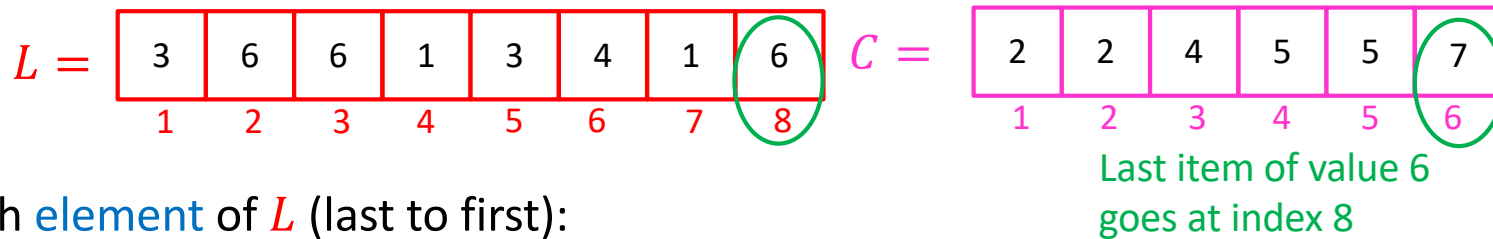
To sort: last item of
value 3 goes at index 4

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

For $i = 2$ to $\text{len}(C)$:
 $C[i] = C[i - 1] + C[i]$

Counting Sort

- Idea: Count how many things are \leq each element



For each element of L (last to first):

Use C to find its proper place in B .

Put element from L there.

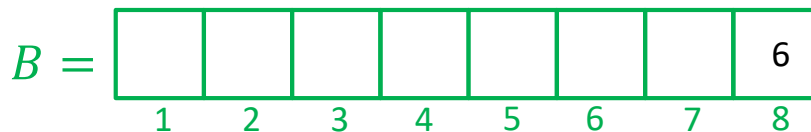
Decrement that position of C .

Why? If earlier element in L with same value is found, placed right before it.

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

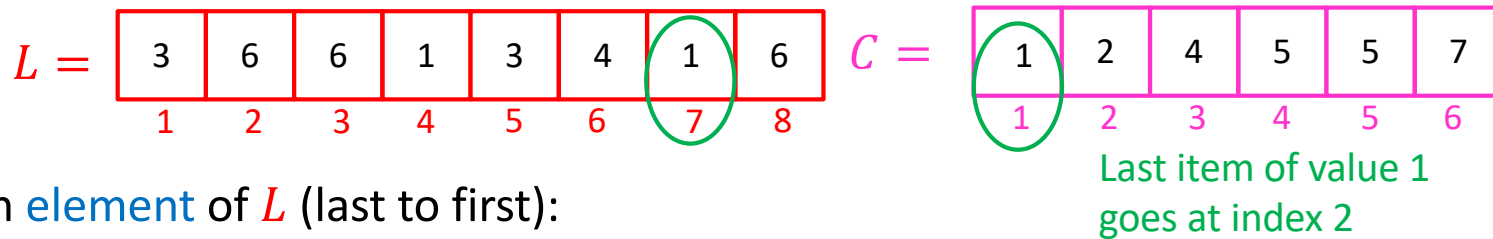
$$B[C[L[i]]] = L[i]$$

$$C[L[i]] = C[L[i]] - 1$$



Counting Sort

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



For each element of L (last to first):

Use C to find its proper place in B .

Put element from L there.

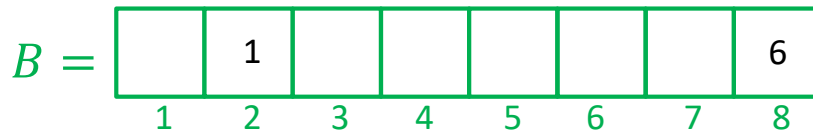
Decrement that position of C .

Why? If earlier element in L with
same value is found, placed right before it.

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

$$B[C[L[i]]] = L[i]$$

$$C[L[i]] = C[L[i]] - 1$$



Run Time: $O(n + k)$

Memory: $O(n + k)$

Is this stable? Why or why not?

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 (?)

One Exabyte = 10^{18} bytes
1 million terabytes (TB)
1 billion gigabytes (GB)

100,000 x Library of Congress (print)

12 Exabytes



https://en.wikipedia.org/wiki/Utah_Data_Center

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

| | | | | | | | | | |
|-----|---------------------------------|-----|--------------------------|---|-------------------|---|---|-----|-----|
| 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 |
| 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 100’s place

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

Run Time: $O(d(n + b))$
 d = digits in largest value
 b = base of representation

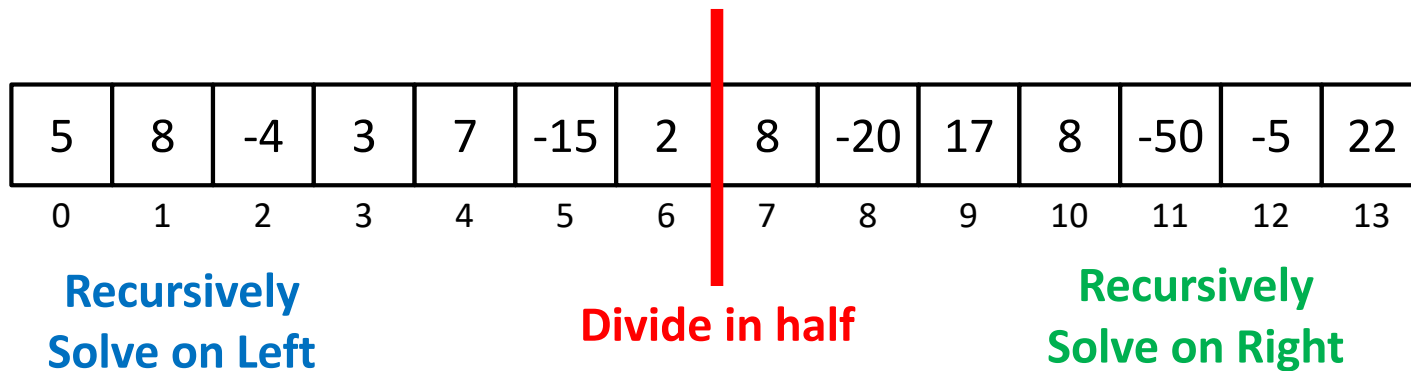
| | | | | | | | | | |
|-----|--------------------------|------------|-----|-----|------------|---|---|-------------------|------------|
| 018 | 101 103 113 121 | 245 255 | 323 | 401 | 512 555 | | | 800 801 823 | 901 999 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Maximum Sum Continuous Subarray Problem

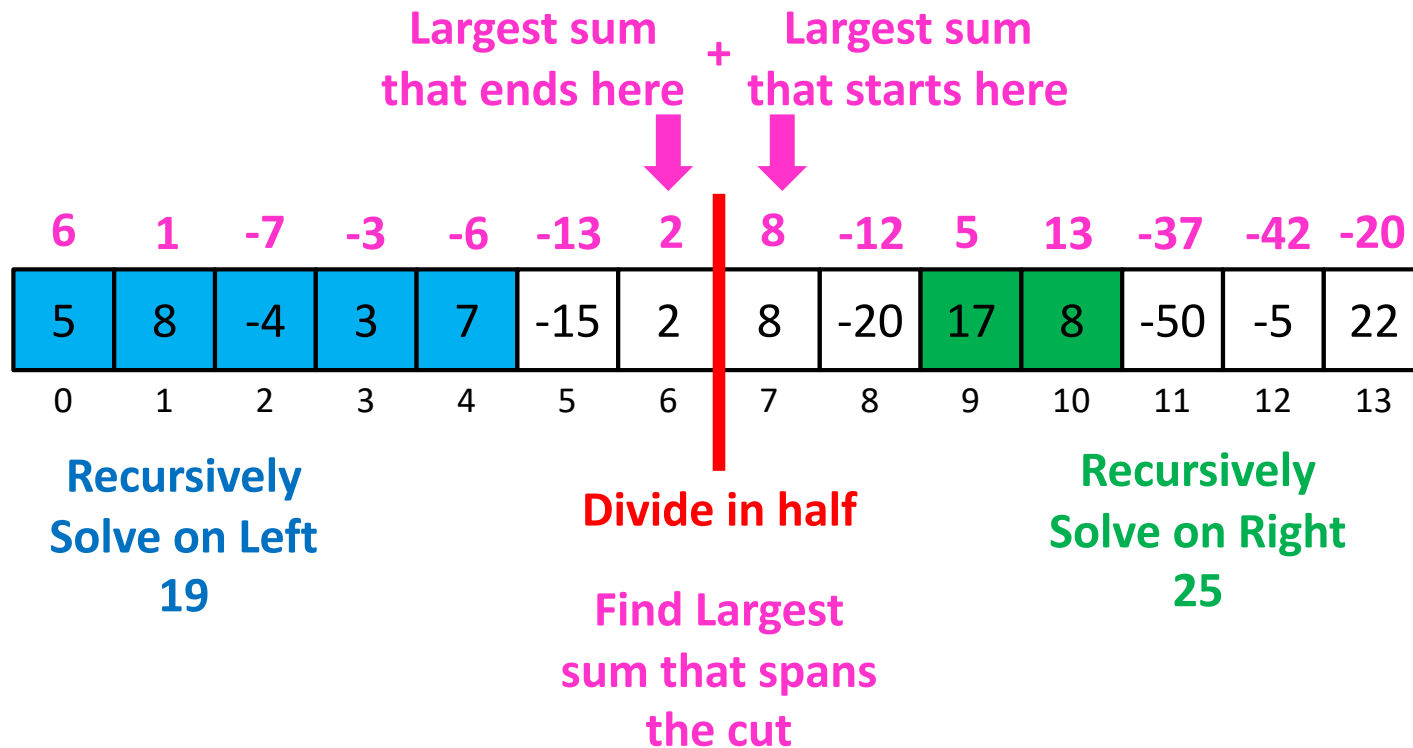
The maximum-sum subarray of a given array of integers A is the interval $[a, b]$ such that the sum of all values in the array between a and b inclusive is maximal.

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 $\Theta(n \log n)$

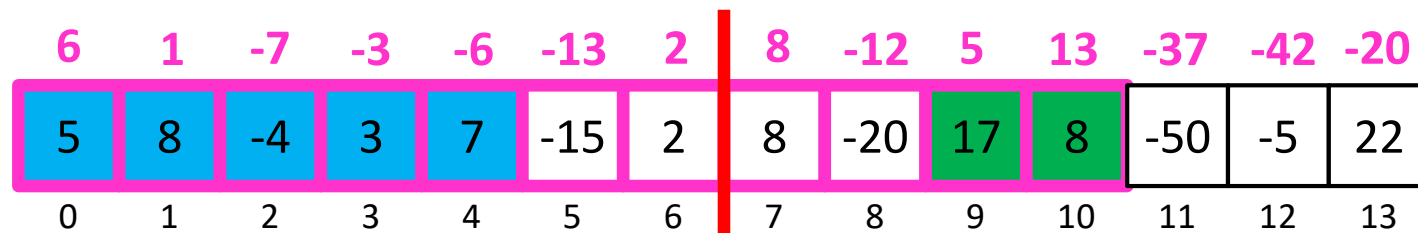


Divide and Conquer $\Theta(n \log n)$



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

Return the Max of
Left, Right, Center



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”

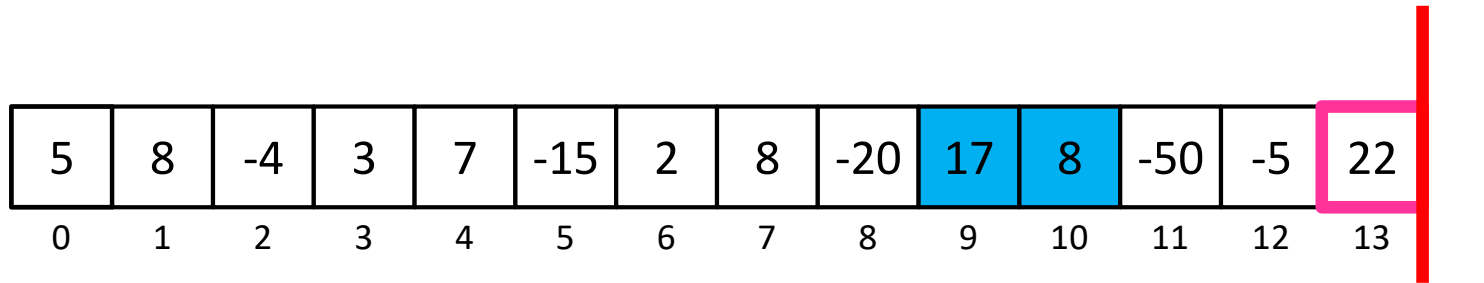
- Divide and Conquer
 - Break the problem up into several subproblems of roughly equal size, recursively solve
 - E.g. Karatsuba, Closest Pair of Points, Mergesort...
- Decrease and Conquer
 - Break the problem into a single smaller subproblem, recursively solve
 - E.g. Impossible Missions Force (Double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

Pattern So Far

- Typically looking to divide the problem by some fraction ($\frac{1}{2}$, $\frac{1}{4}$ the size)
- Not necessarily always the best!
 - Sometimes, we can write faster algorithms by finding **unbalanced** divides.


Chip and Conquer

- Divide
 - Make a subproblem of all but the last element
- Conquer
 - Find best subarray on the left ($BSL(n - 1)$)
 - Find the best subarray ending at the divide ($BED(n - 1)$)
- Combine
 - New Best Ending at the Divide:
 - $BED(n) = \max(BED(n - 1) + arr[n], 0)$
 - New best on the left:
 - $BSL(n) = \max(BSL(n - 1), BED(n))$




Recursively
Solve on Left
25

Find Largest
sum ending at
the cut
22



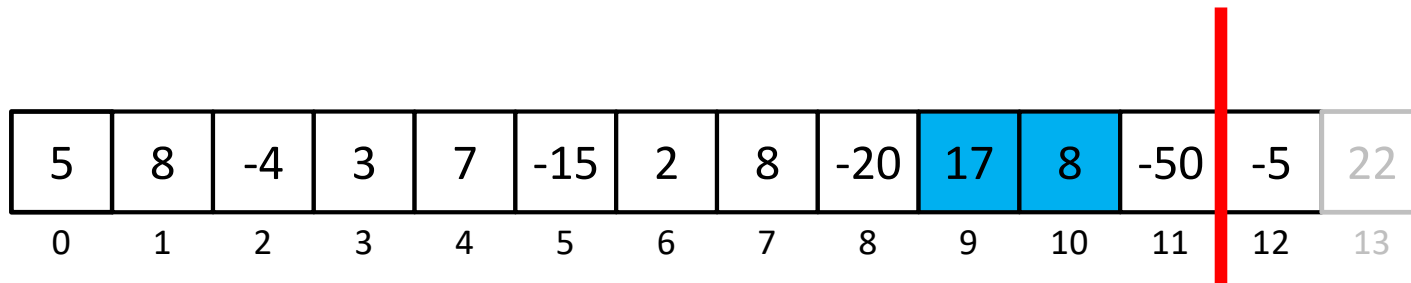
| | | | | | | | | | | | | | |
|---|---|----|---|---|-----|---|---|-----|----|----|-----|----|----|
| 5 | 8 | -4 | 3 | 7 | -15 | 2 | 8 | -20 | 17 | 8 | -50 | -5 | 22 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |



Recursively
Solve on Left
25

Find Largest
sum ending at
the cut
0

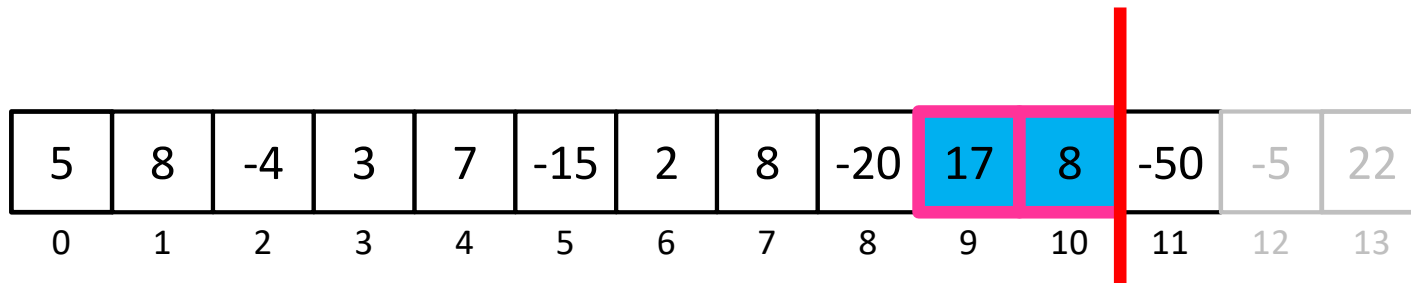
Divide



Recursively
Solve on Left
25

Find Largest
sum ending at
the cut
0

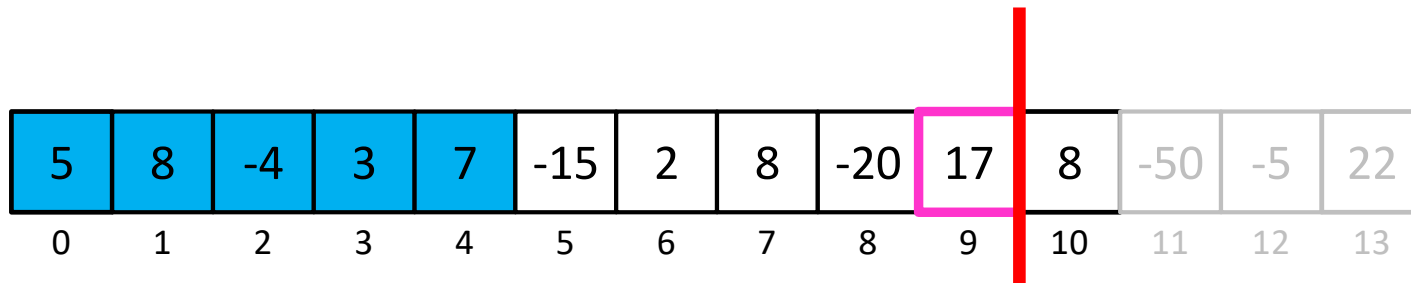
Divide



Recursively
Solve on Left
25

Find Largest
sum ending at
the cut
25

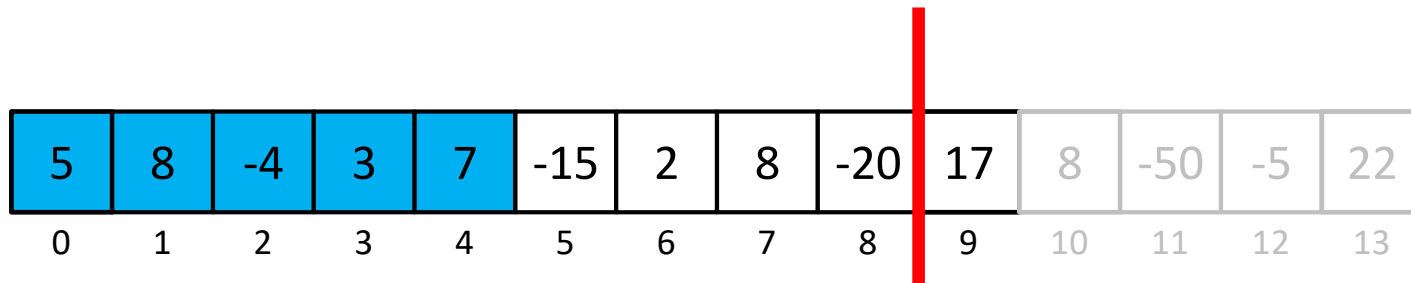
Divide



**Recursively
Solve on Left
19**

Divide

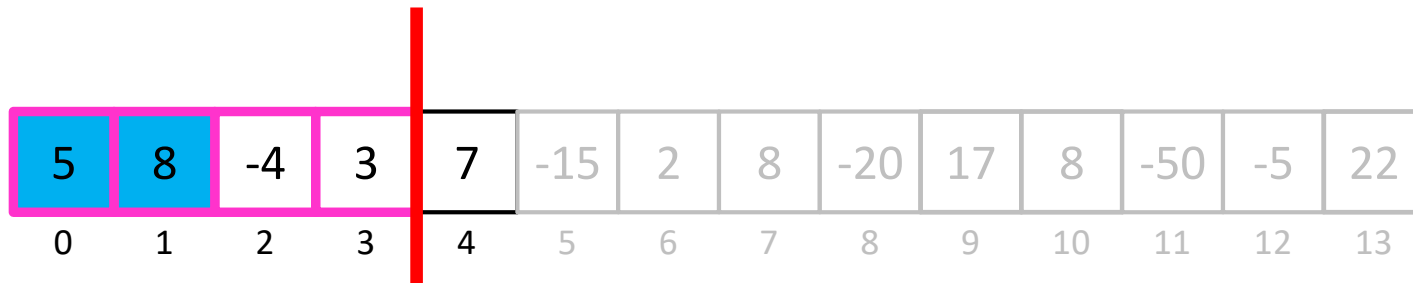
**Find Largest
sum ending at
the cut
17**



**Recursively
Solve on Left
19**

Divide

**Find Largest
sum ending at
the cut
0**



Recursively
Solve on Left
13

Divide

Find Largest
sum ending at
the cut
12

Chip and Conquer

- Divide
 - Make a subproblem of all but the last element
- Conquer
 - Find best subarray on the left ($BSL(n - 1)$)
 - Find the best subarray ending at the divide ($BED(n - 1)$)
- Combine
 - New Best Ending at the Divide:
 - $BED(n) = \max(BED(n - 1) + arr[n], 0)$
 - New best on the left:
 - $BSL(n) = \max(BSL(n - 1), BED(n))$

Was unbalanced better? YES

- Old:

- We divided in **Half**
- We solved 2 different problems:
 - Find the best overall on **BOTH** the **left/right**
 - Find the best which end/start on **BOTH** the **left/right** respectively
- **Linear** time combine

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

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

- New:

- We divide by **1, n-1**
- We solve 2 different problems:
 - Find the best overall on the **left ONLY**
 - Find the best which ends on the **left ONLY**
- **Constant** time combine

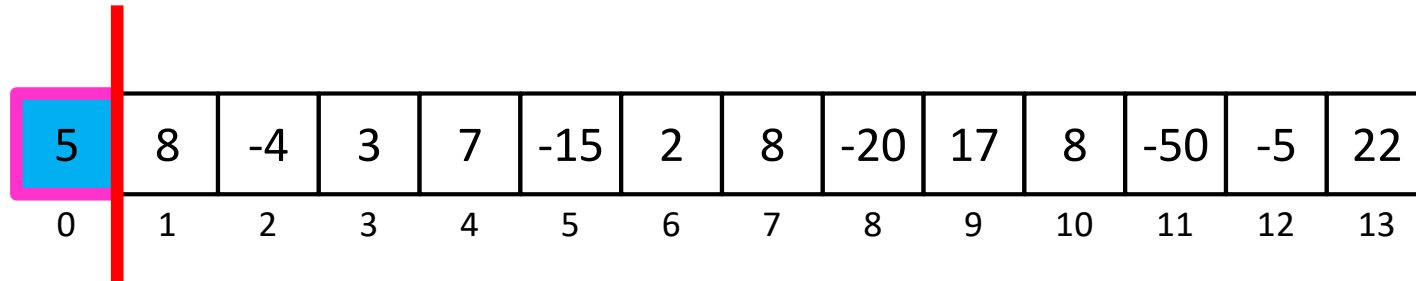
$$T(n) = 1T(n-1) + 1$$

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

MSCS Problem - Redux

- Solve in $O(n)$ by increasing the problem size by 1 each time.
- **Idea:** Only include negative values if the positives on both sides of it are “worth it”

$\Theta(n)$ Solution



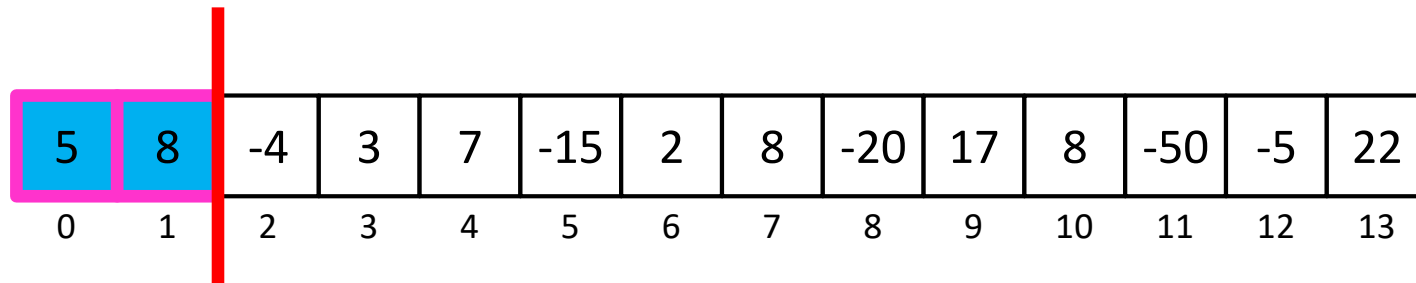
Begin here

Remember two values:

Best So Far
5

Best ending here
5

$\Theta(n)$ Solution

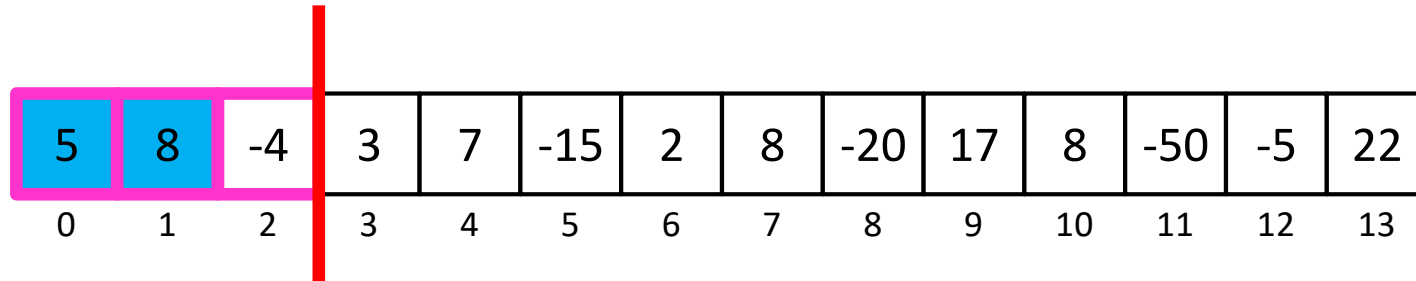


Remember two values:

Best So Far
13

Best ending here
13

$\Theta(n)$ Solution

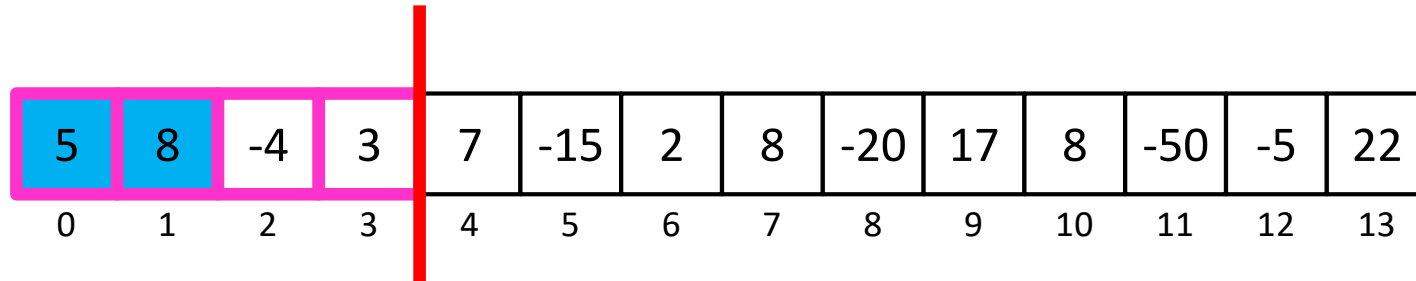


Remember two values:

Best So Far
13

Best ending here
9

$\Theta(n)$ Solution

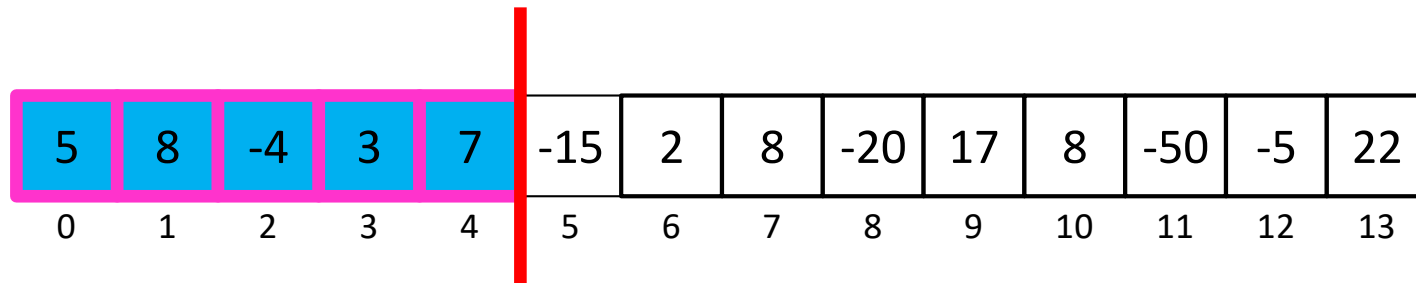


Remember two values:

Best So Far
13

Best ending here
12

$\Theta(n)$ Solution

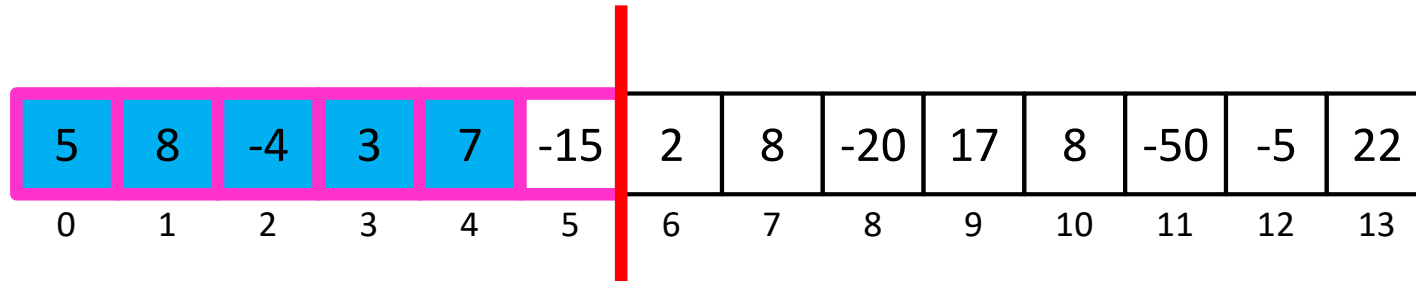


Remember two values:

Best So Far
19

Best ending here
19

$\Theta(n)$ Solution

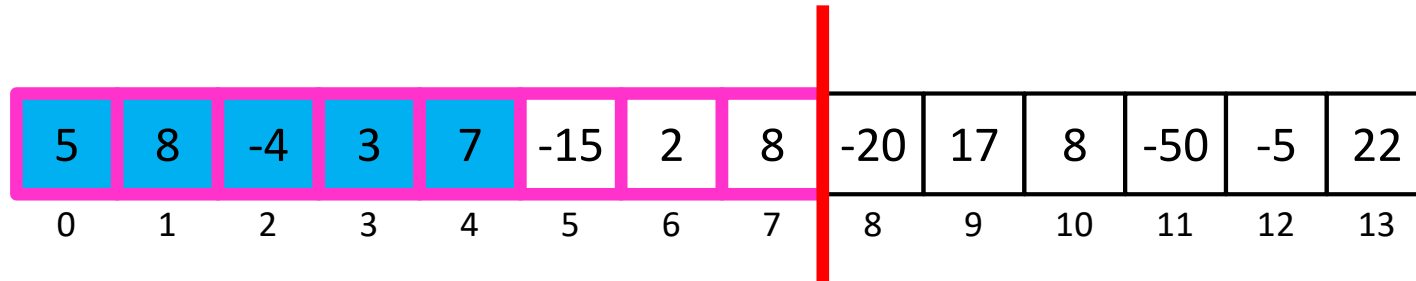


Remember two values:

Best So Far
19

Best ending here
4

$\Theta(n)$ Solution

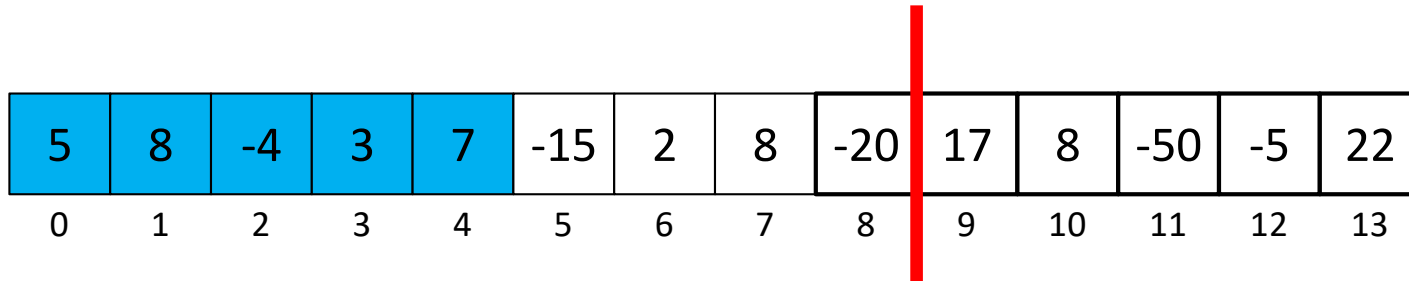


Remember two values:

Best So Far
19

Best ending here
14

$\Theta(n)$ Solution

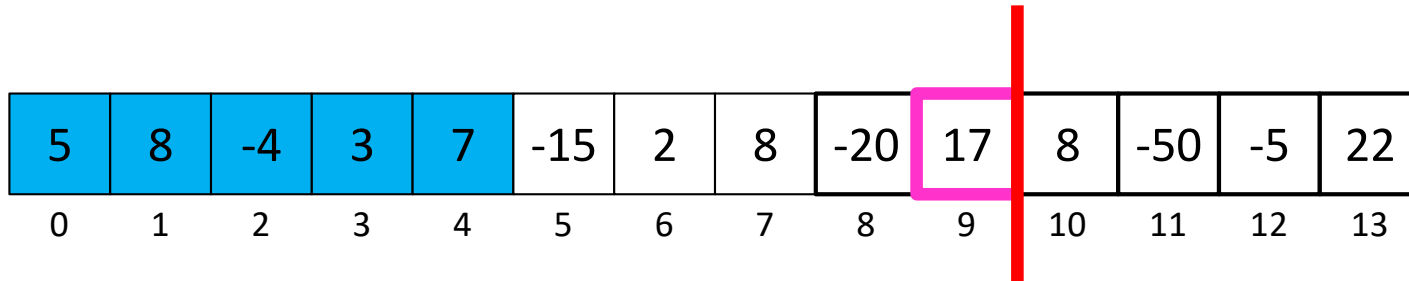


Remember two values:

Best So Far
19

Best ending here
0

$\Theta(n)$ Solution



Remember two values:

Best So Far
19

Best ending here
17

$\Theta(n)$ Solution

| | | | | | | | | | | | | | |
|---|---|----|---|---|-----|---|---|-----|----|----|-----|----|----|
| 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 |

Remember two values:

Best So Far
25

Best ending here
25