

CSE 332 Summer 2024

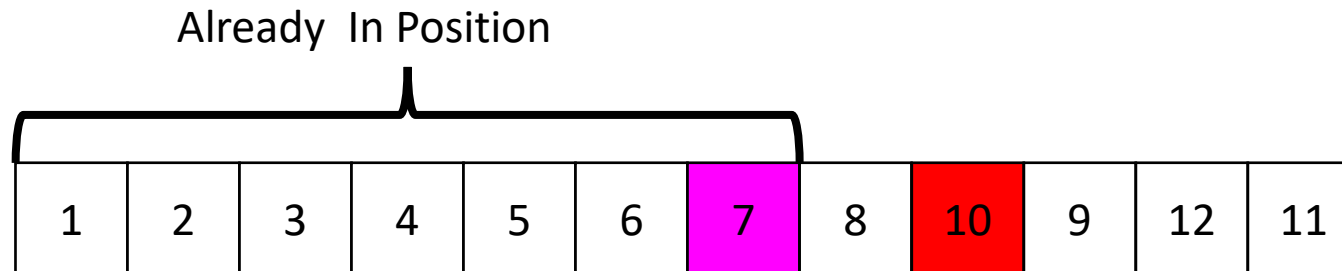
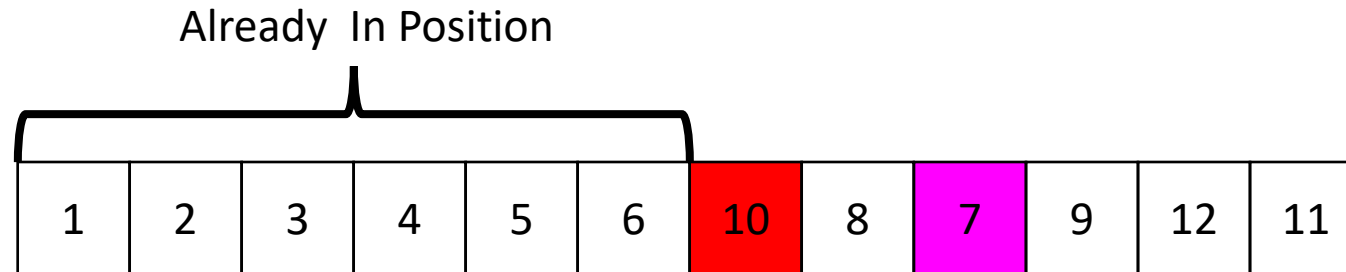
Lecture 12: Sorting

Nathan Brunelle

<http://www.cs.uw.edu/332>

Selection Sort

- Idea: Find the **next smallest** element, swap it into the **next index** in the array



Selection Sort

- Swap the thing at index 0 with the smallest thing in the array
- Swap the thing at index 1 with the smallest thing after index 0
- ...
- Swap the thing at index i with the smallest thing after index $i - 1$

```
for (i=0; i<a.length; i++){  
    smallest = i;  
    for (j=i; j<a.length; j++){  
        if (a[j]<a[smallest]){ smallest=j;}  
    }  
    temp = a[i];  
    a[i] = a[smallest];  
    a[smallest] = temp;  
}
```

Running Time:

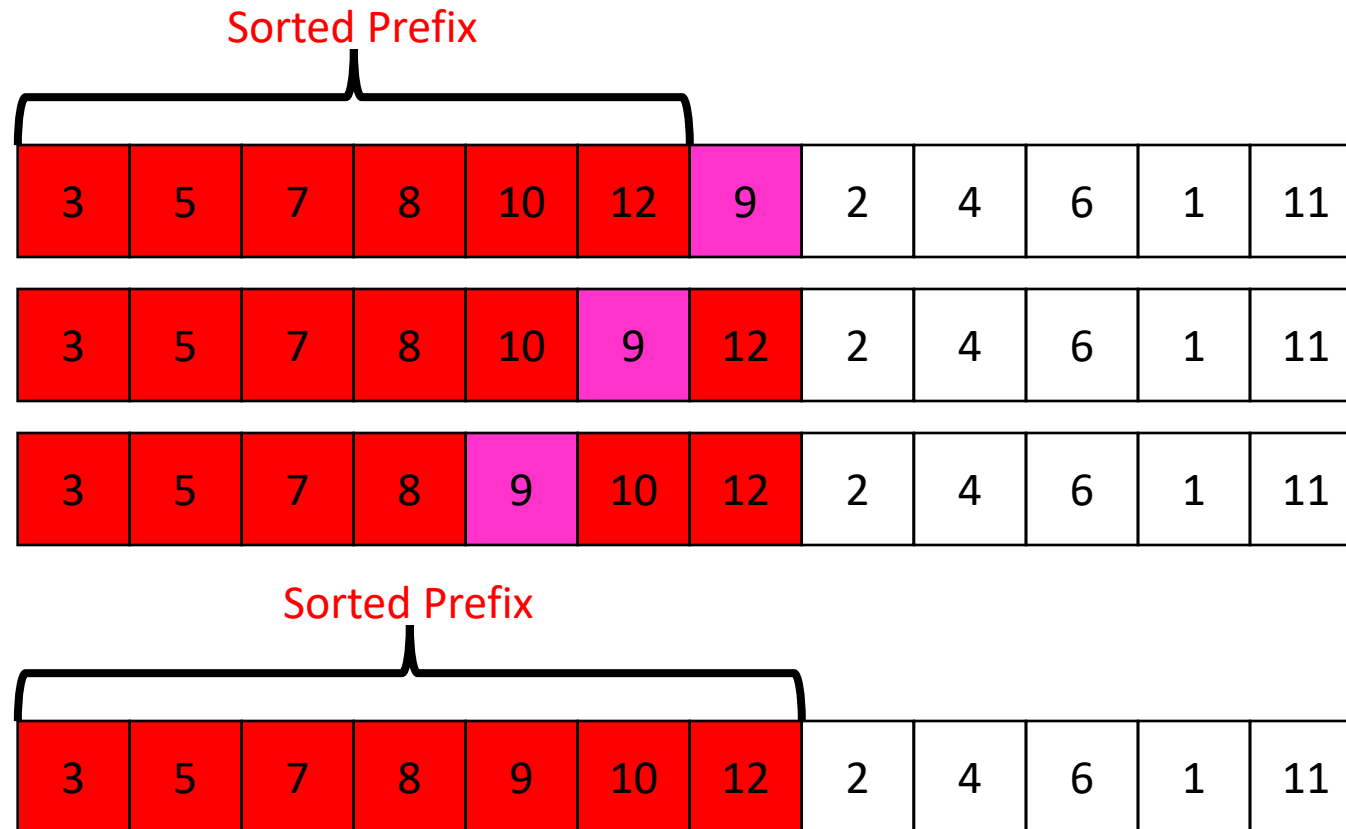
Worst Case: $\Theta(n^2)$

Best Case: $\Theta(n^2)$

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insertion Sort

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



Insertion Sort

- If the items at index 0 and 1 are out of order, swap them
- Keep swapping the item at index 2 with the thing to its left as long as the left thing is larger
- ...
- Keep swapping the item at index i with the thing to its left as long as the left thing is larger

```
for (i=1; i<a.length; i++){  
  prev = i-1;  
  while(a[i] < a[prev] && prev > -1){  
    temp = a[i];  
    a[i] = a[prev];  
    a[prev] = temp;  
    i--;  
    prev--;  
  }  
}
```

Running Time:

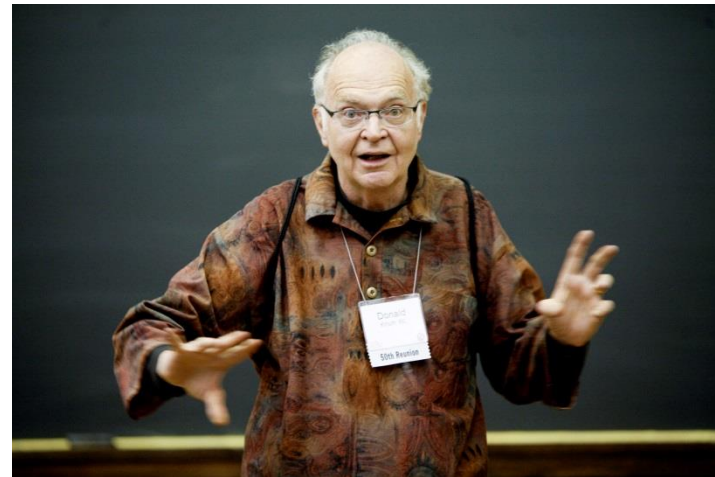
Worst Case: $\Theta(n^2)$

Best Case: $\Theta(n)$

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

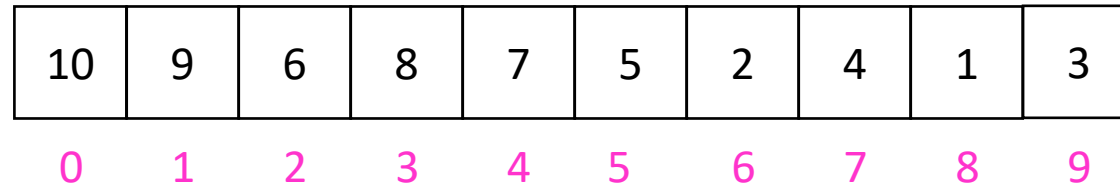
Aside: Bubble Sort – we won't cover it

"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



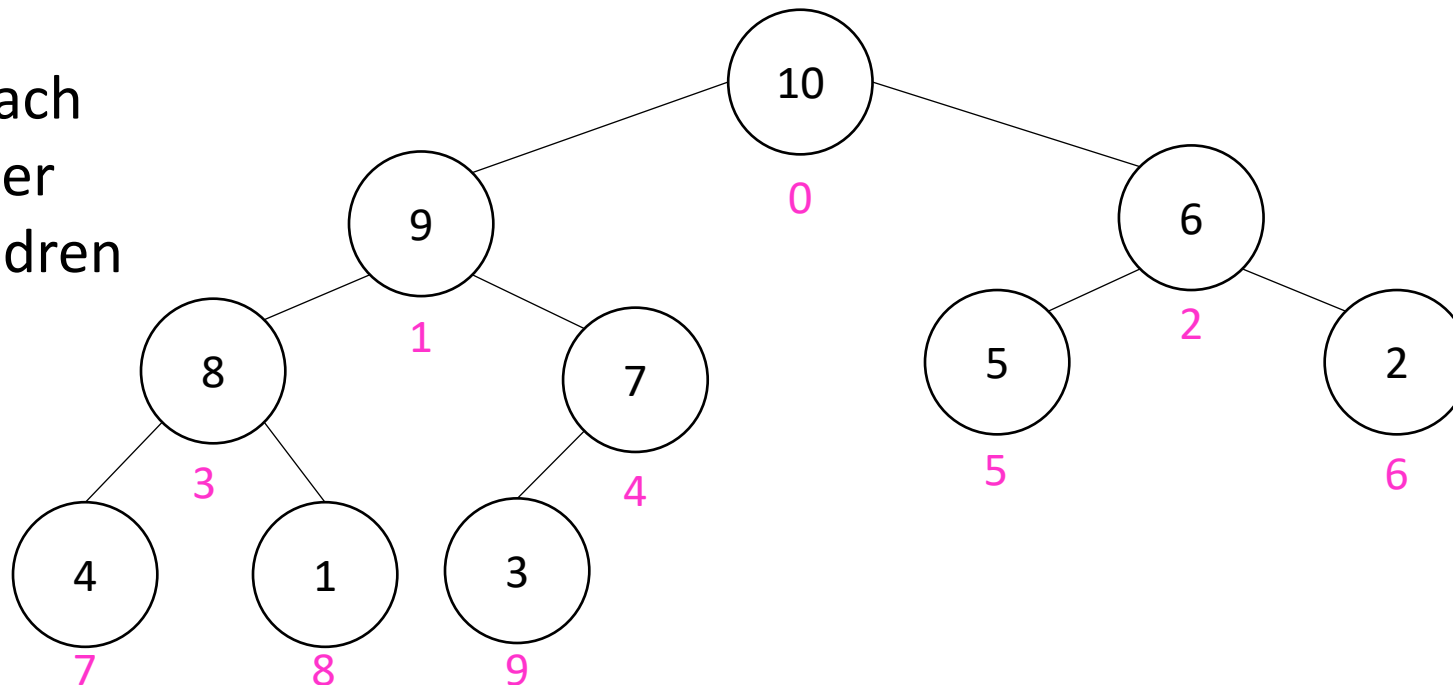
Heap Sort

- **Idea:** Build a maxHeap, repeatedly delete the max element from the heap to build sorted list Right-to-Left



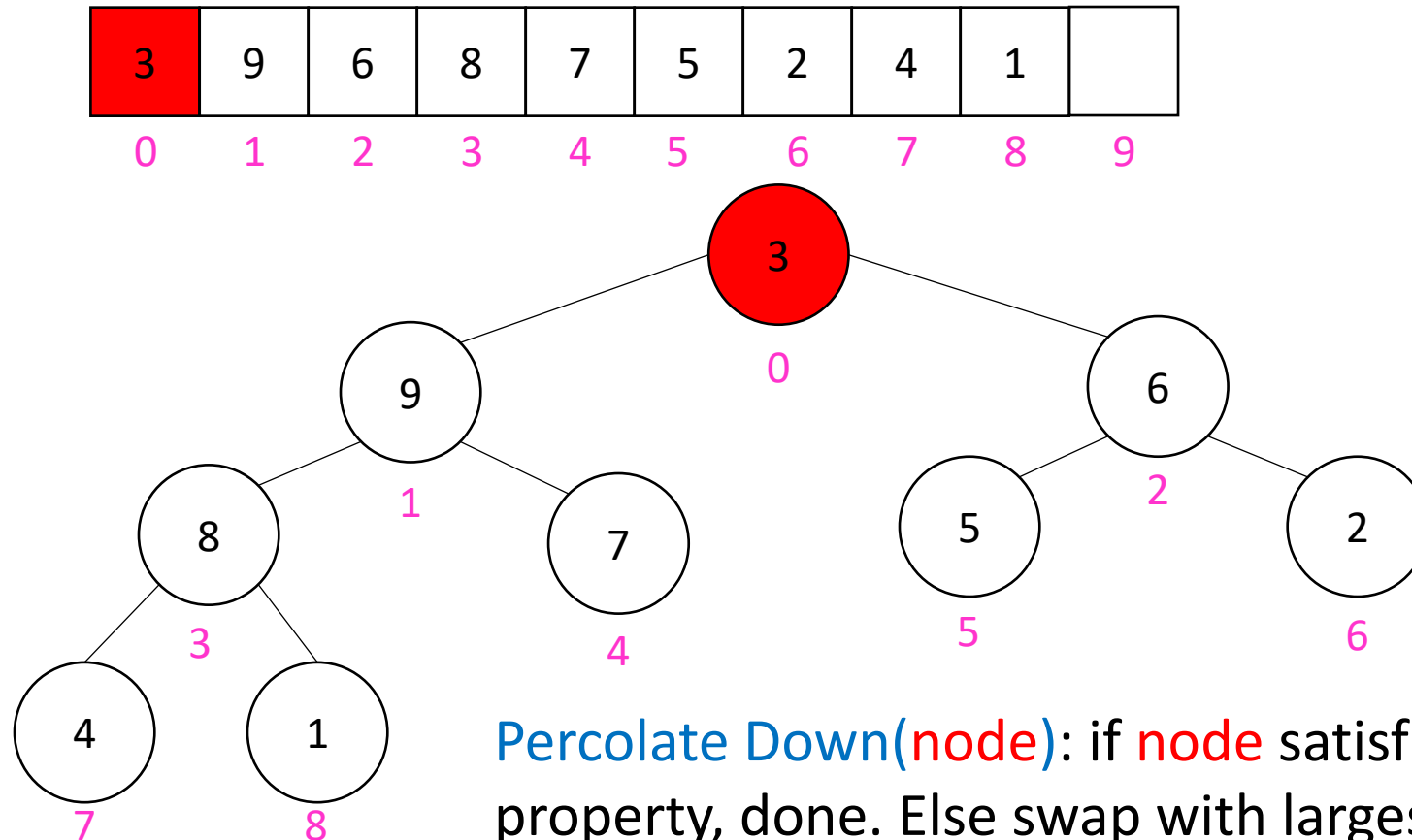
Max Heap

Property: Each node is larger than its children



Heap Sort

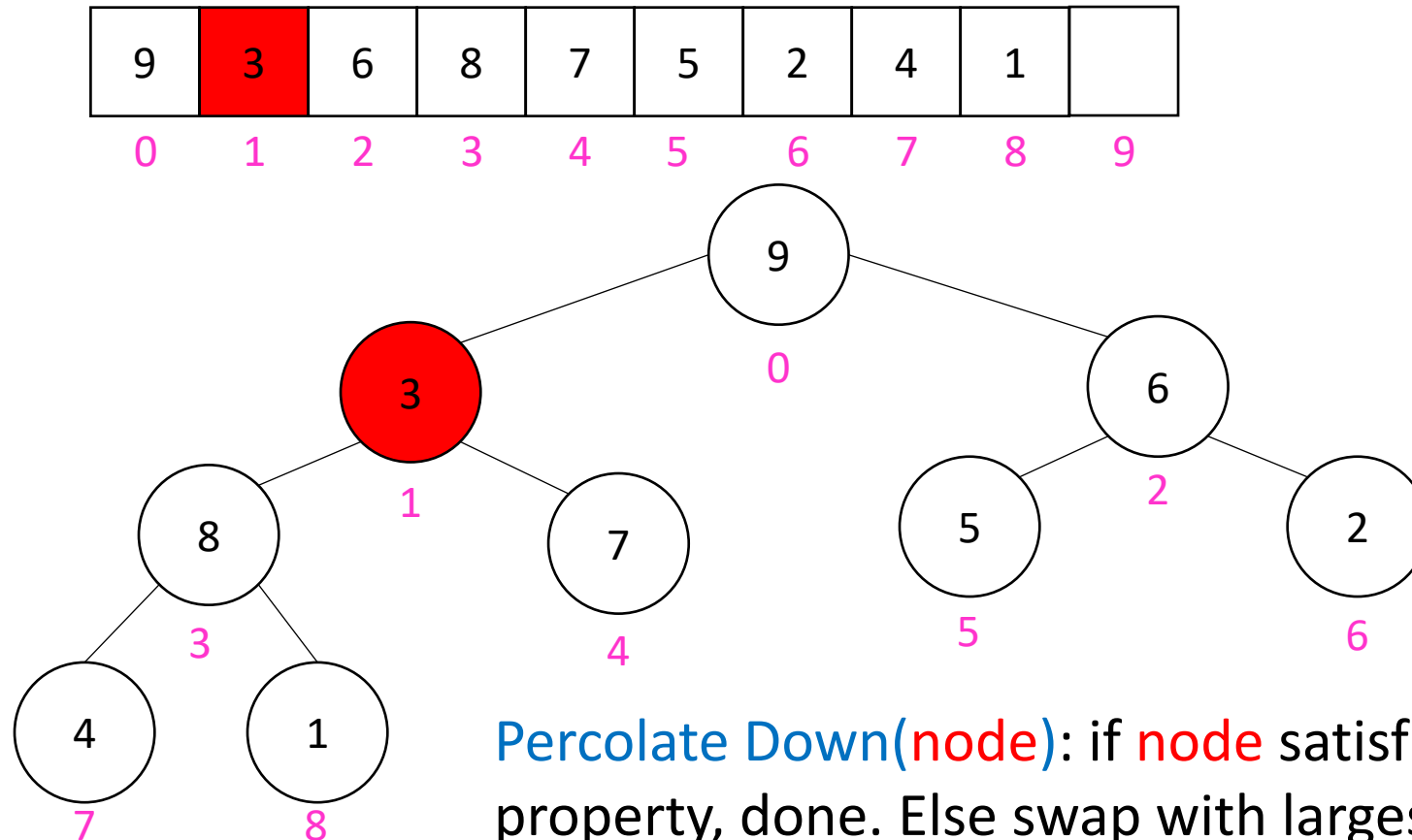
- Remove the Max element (i.e. the root) from the Heap: replace with last element, call `percolateDown(root)`



Percolate Down(node): if **node** satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

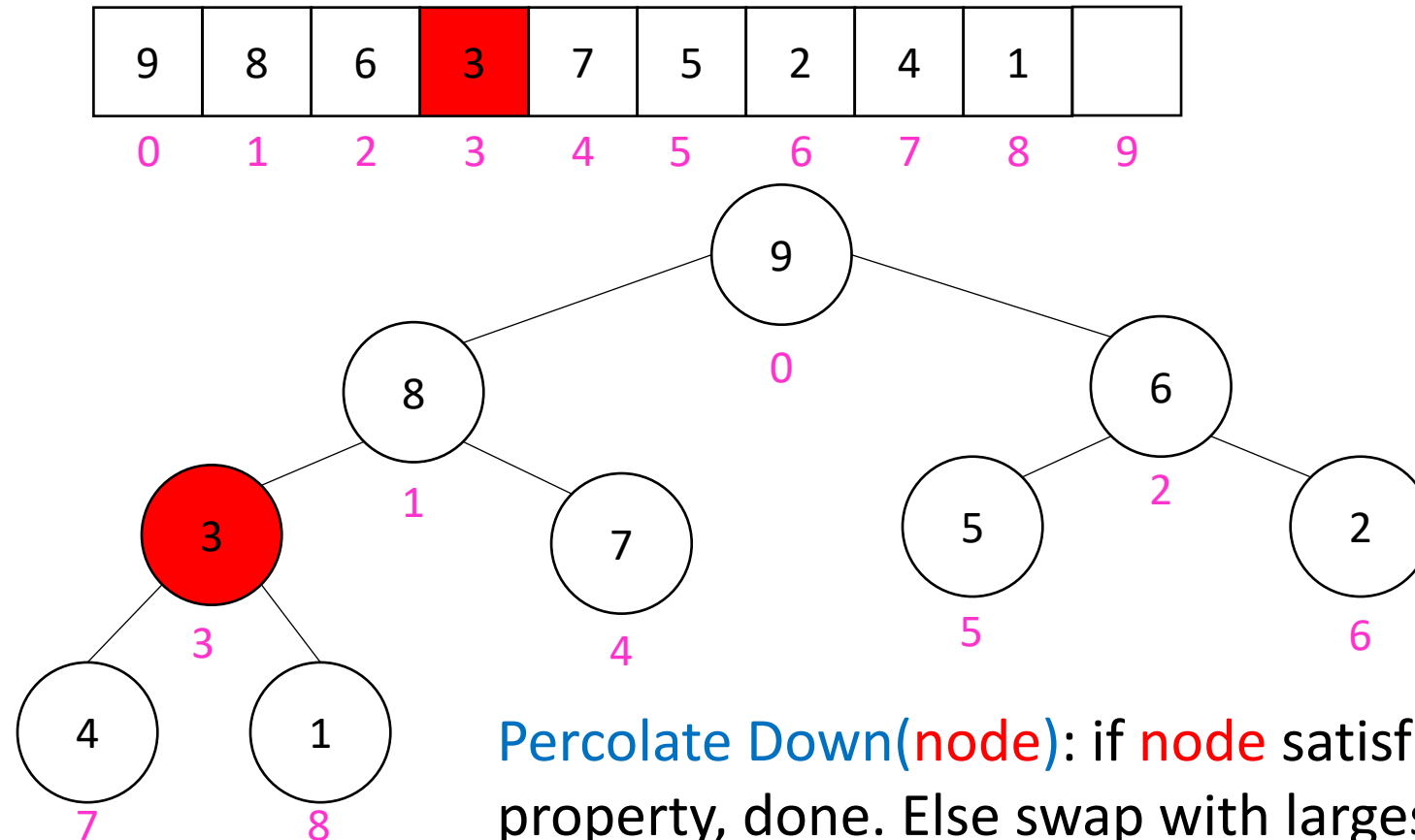
- Remove the Max element (i.e. the root) from the Heap: replace with last element, call `percolateDown(root)`



Percolate Down(node): if **node** satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

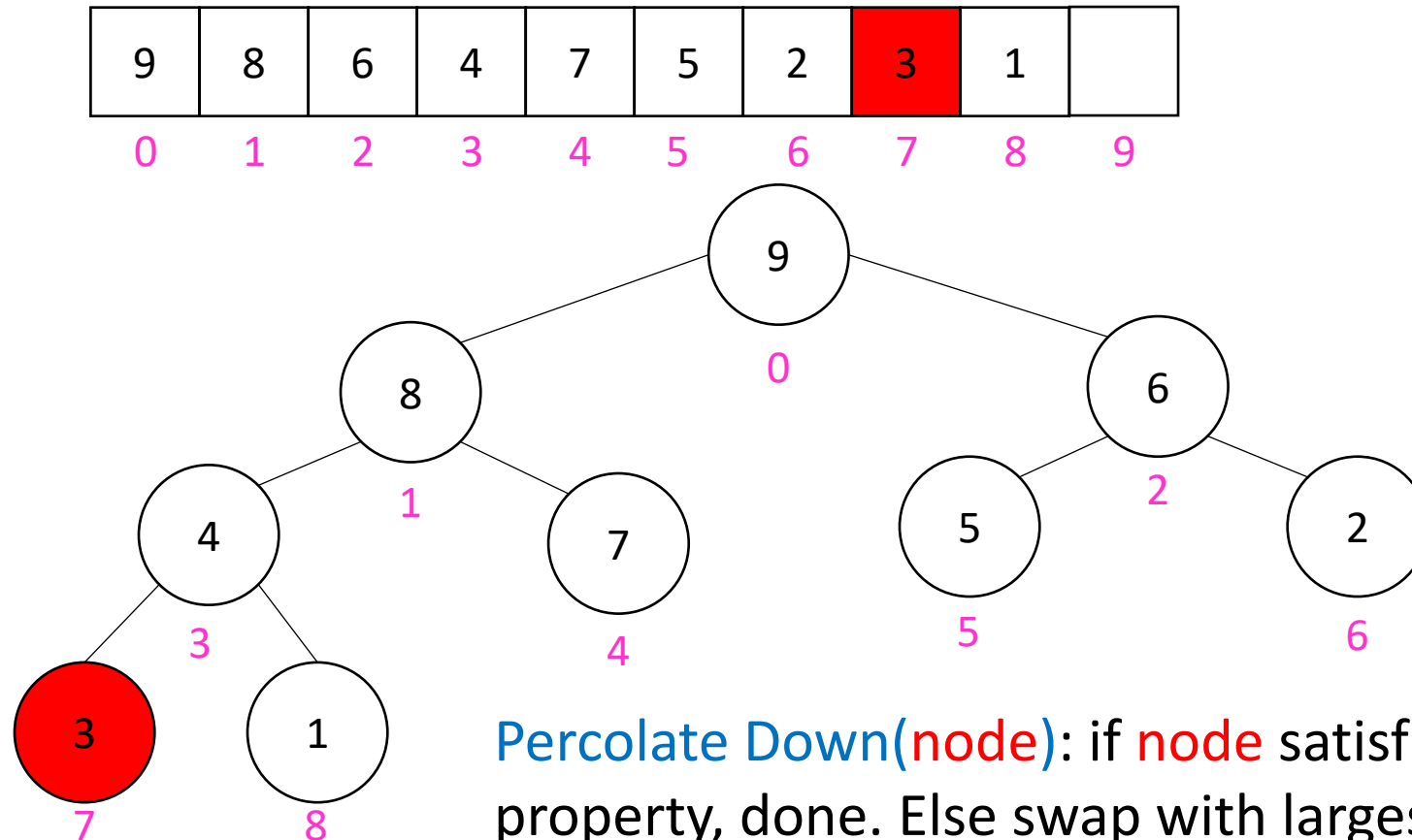
- Remove the Max element (i.e. the root) from the Heap: replace with last element, call `percolateDown(root)`



Percolate Down(node): if **node** satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

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



Percolate Down(node): if **node** satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort

- Build a heap
- Call deleteMax
- Put that at the end of the array

```
myHeap = buildHeap(a);  
for (int i = a.length-1; i>=0; i--){  
    item = myHeap.deleteMax();  
    a[i] = item;  
}
```

Running Time:

Worst Case: $\Theta(\cdot)$

Best Case: $\Theta(\cdot)$

“In Place” Sorting Algorithm

- A sorting algorithm which requires no extra data structures
- Idea: It sorts items just by swapping things in the same array given
- Definition: it only uses $\Theta(1)$ extra space

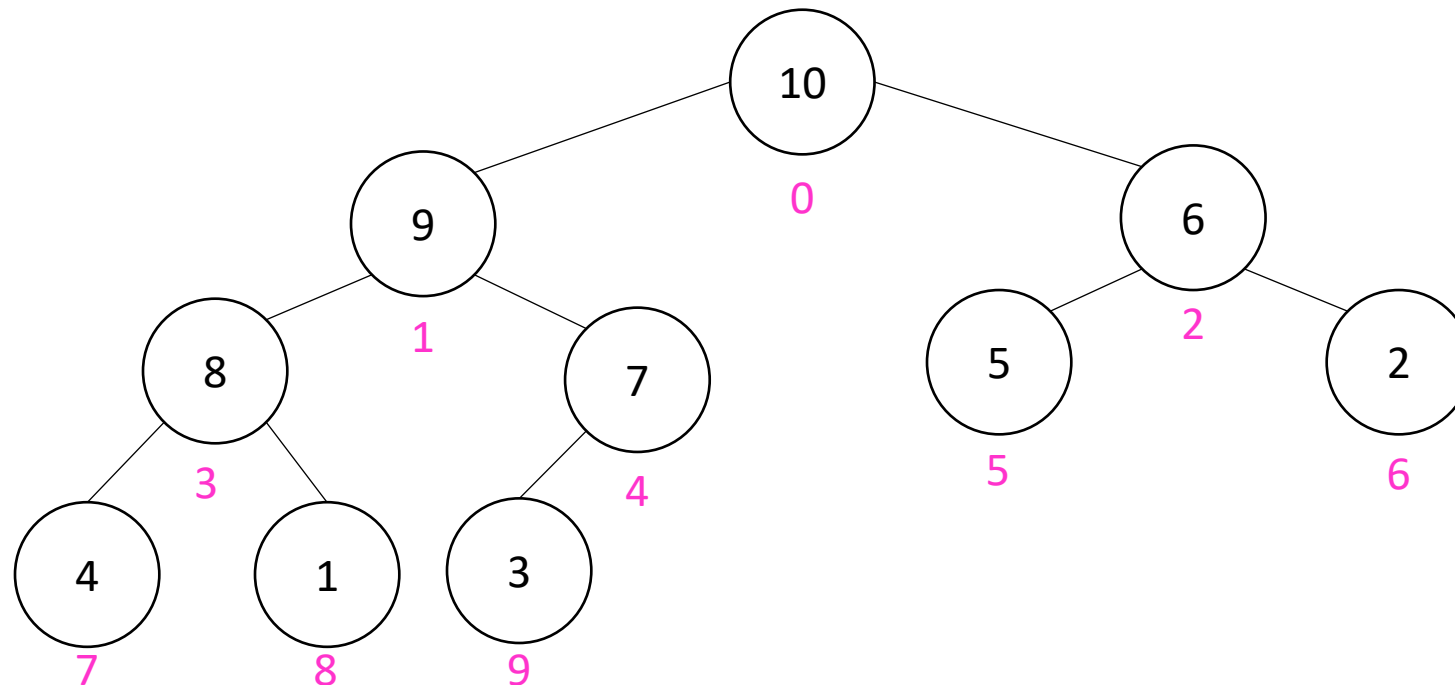
- Selection sort: In Place!
- Insertion sort: In Place!
- Heap sort: Not In Place!
 - But we can fix that!

In Place Heap Sort

- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

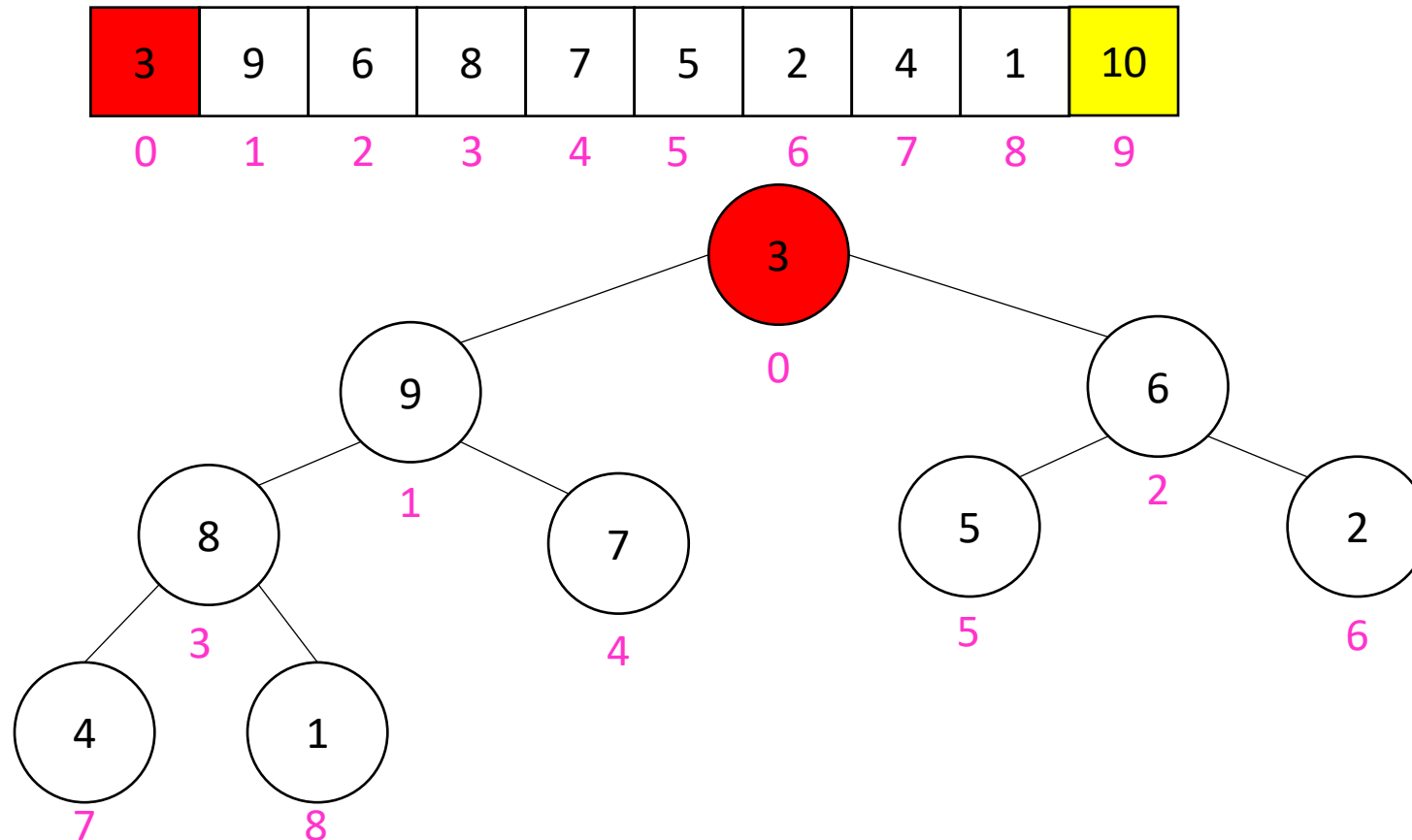
10	9	6	8	7	5	2	4	1	3
----	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9



Heap Sort

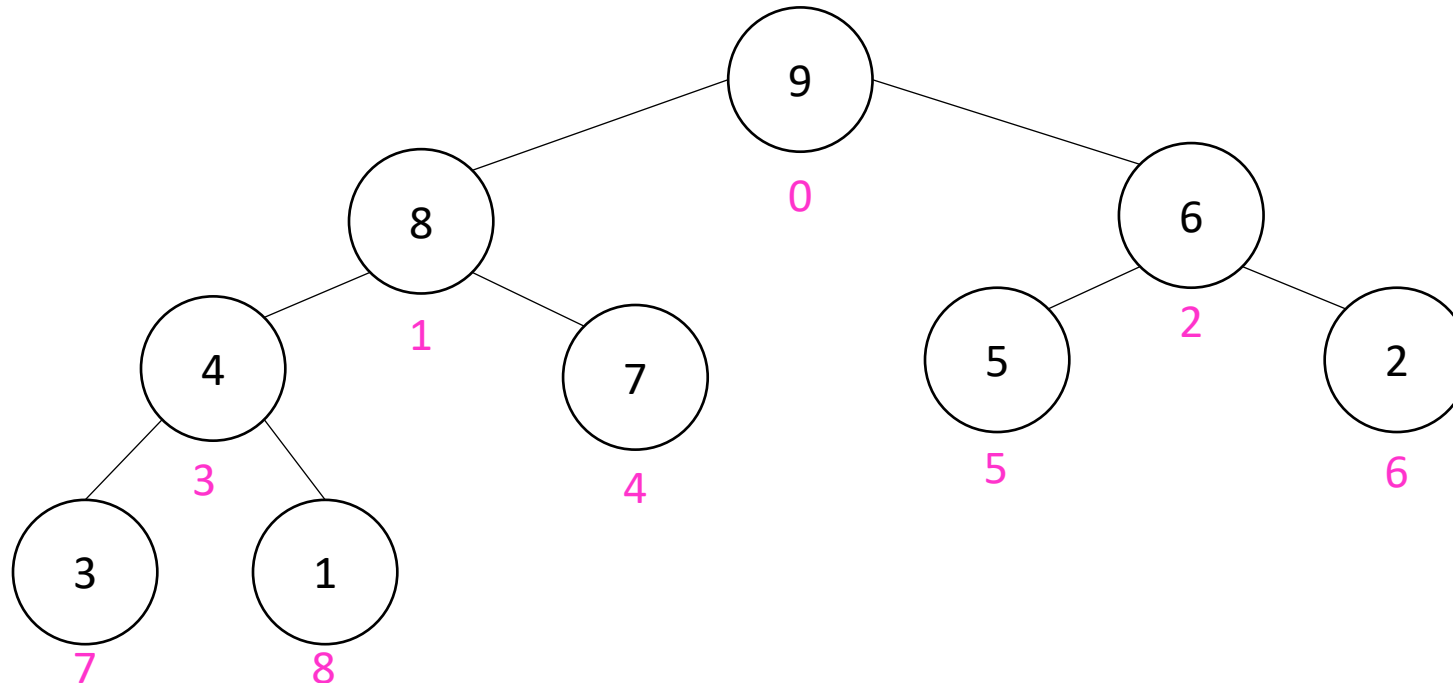
- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



Heap Sort

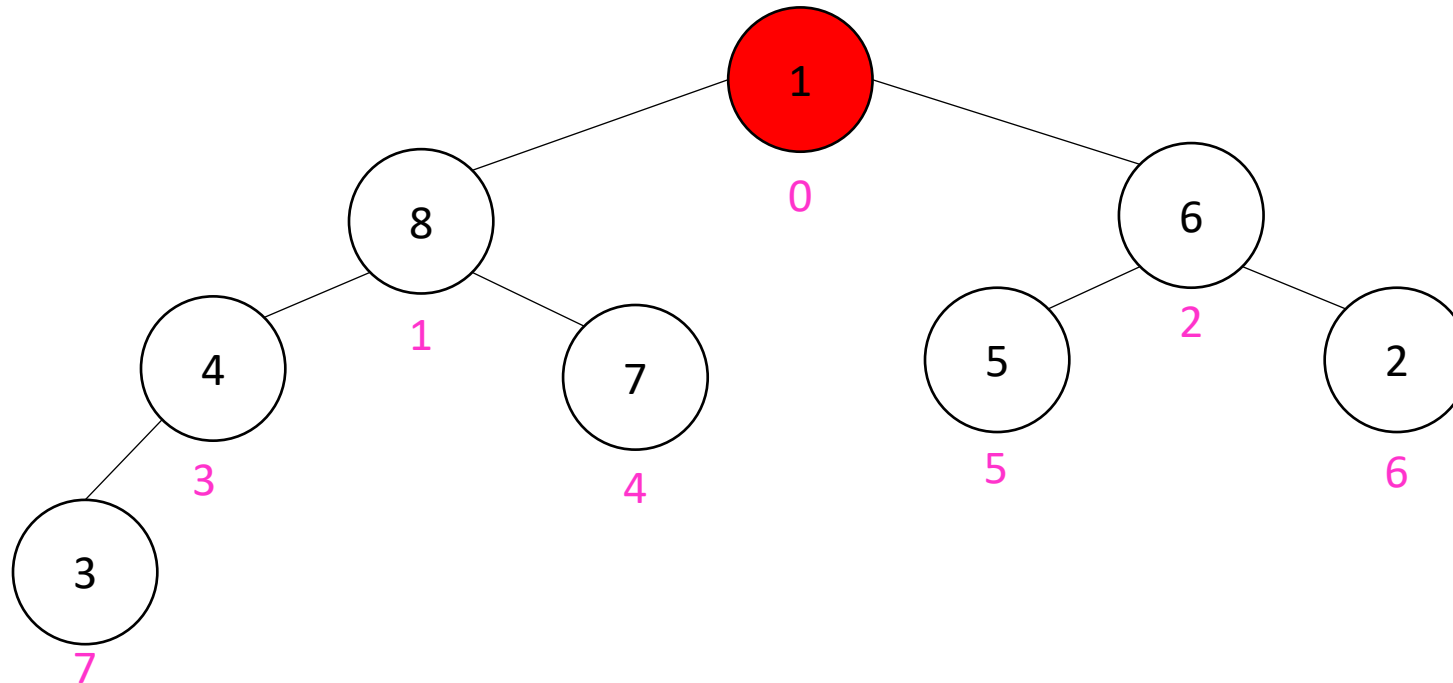
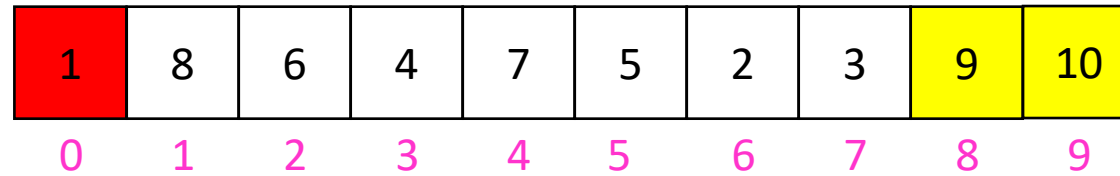
- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

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



Heap Sort

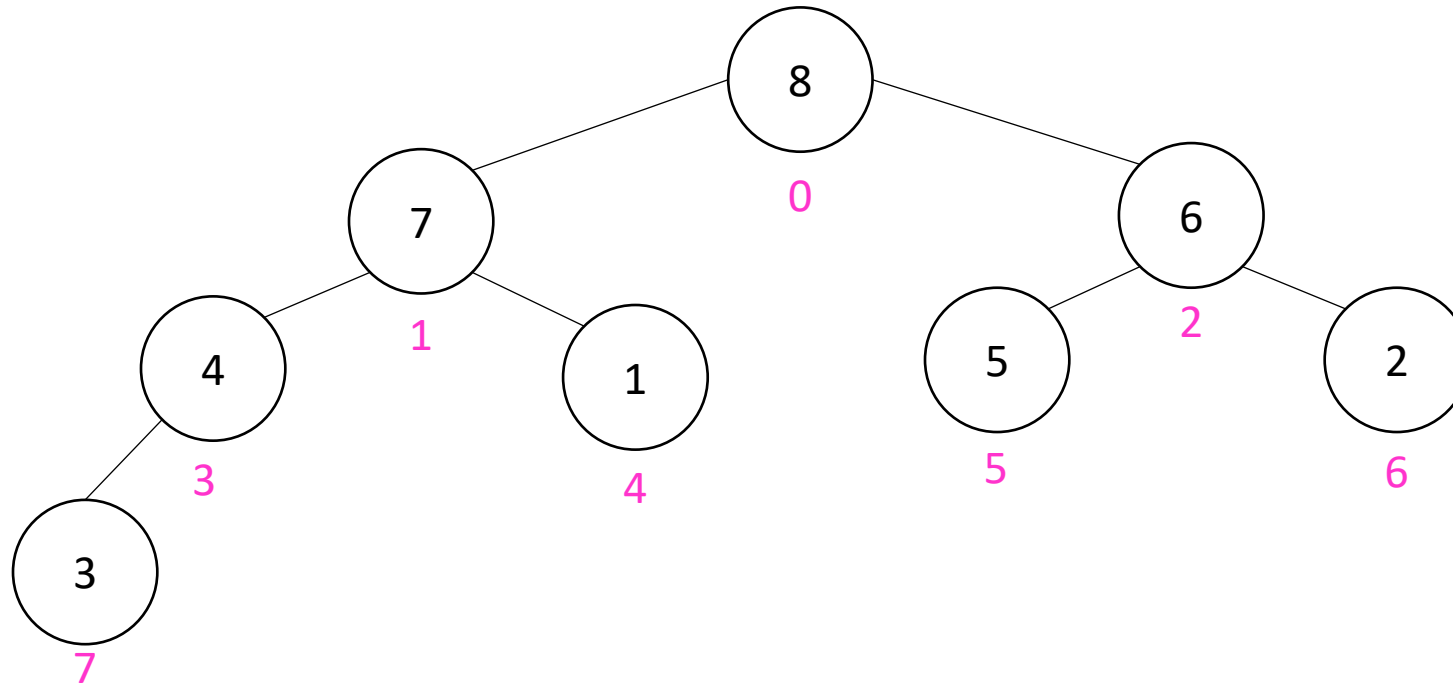
- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



Heap Sort

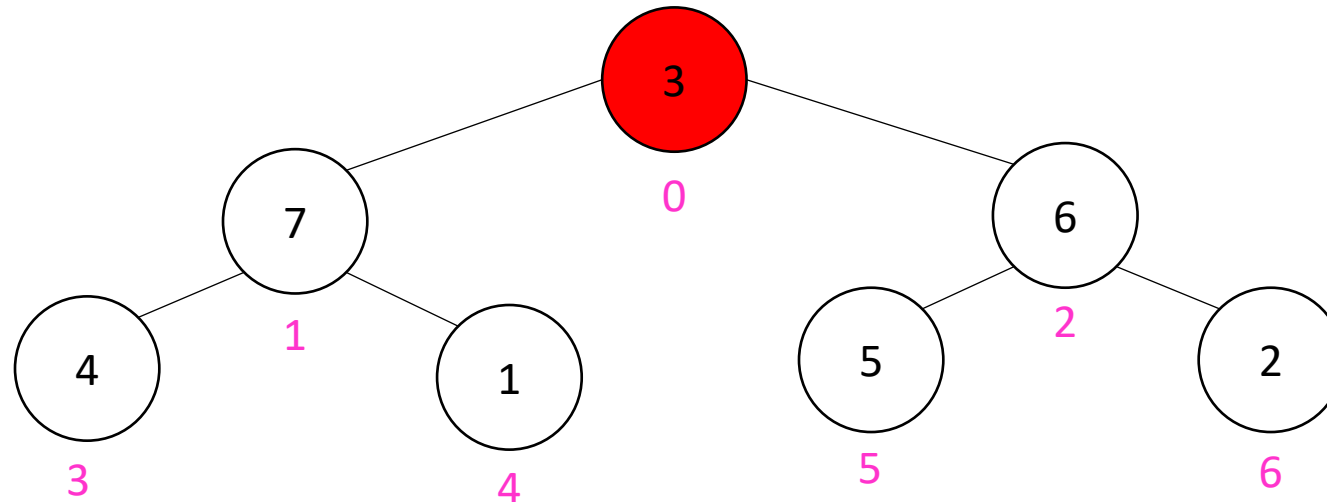
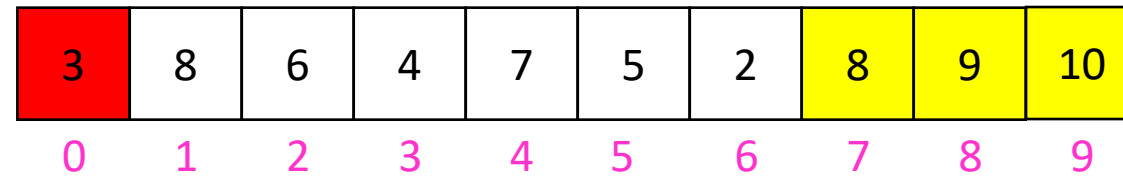
- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

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



Heap Sort

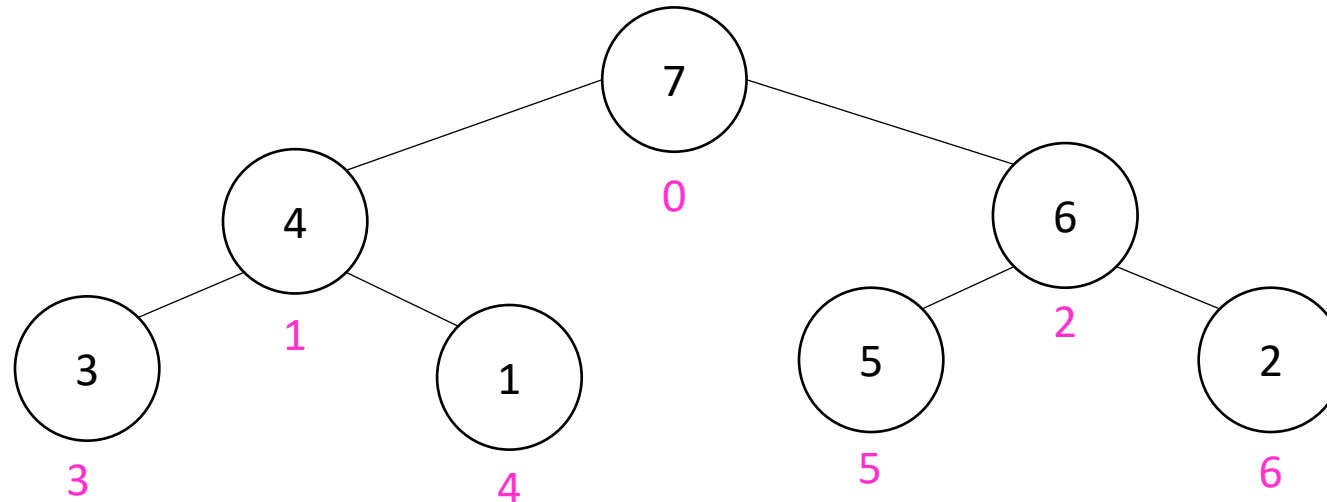
- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



Heap Sort

- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter

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



In Place Heap Sort

- Build a heap using the same array (Floyd's build heap algorithm works)
- Call deleteMax
- Put that at the end of the array

```
buildHeap(a);
for (int i = a.length-1; i>=0; i--){
    temp=a[i]
    a[i] = a[0];
    a[0] = temp;
    percolateDown(0);
}
```

Running Time:

Worst Case: $\Theta(\cdot)$

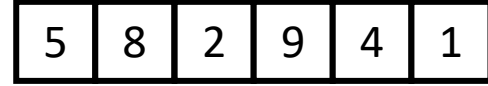
Best Case: $\Theta(\cdot)$

Floyd's buildHeap method

- Working towards the root, one row at a time, percolate down

```
buildHeap(){  
    for(int i = size; i>0; i--){  
        percolateDown(i);  
    }  
}
```

Merge Sort



- **Base Case:**

- If the list is of length 1 or 0, it's already sorted, so just return it



- **Divide:**

- Split the list into two "sublists" of (roughly) equal length



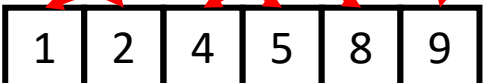
- **Conquer:**

- Sort both lists recursively



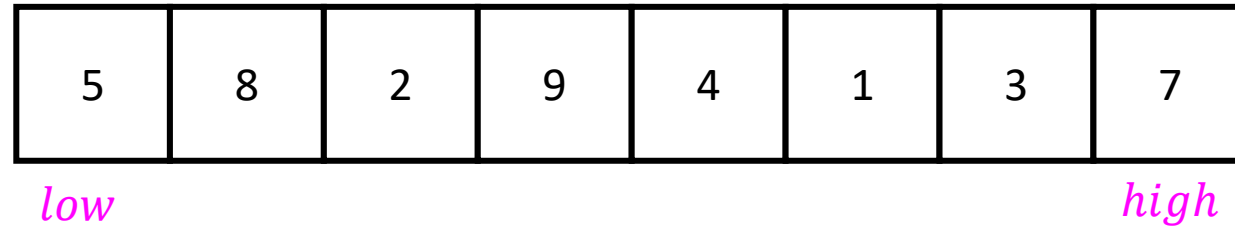
- **Combine:**

- **Merge** sorted sublists into one sorted list



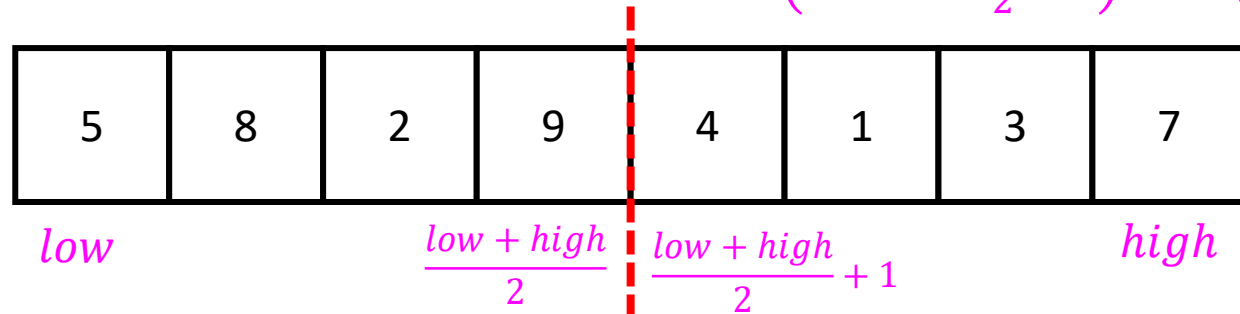
Merge Sort In Action!

Sort between indices *low* and *high*

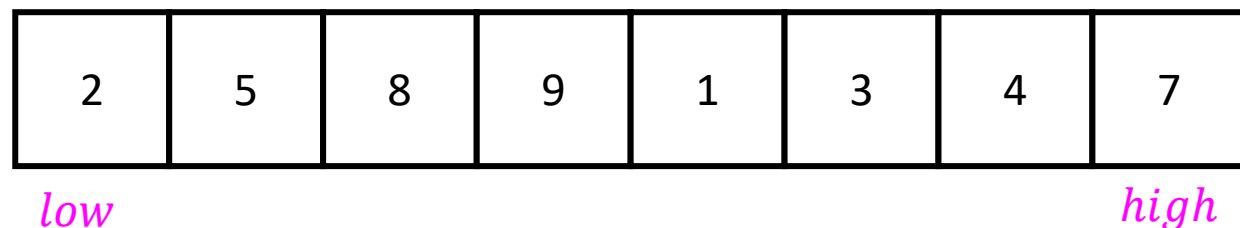


Base Case: if *low* == *high* then that range is already sorted!

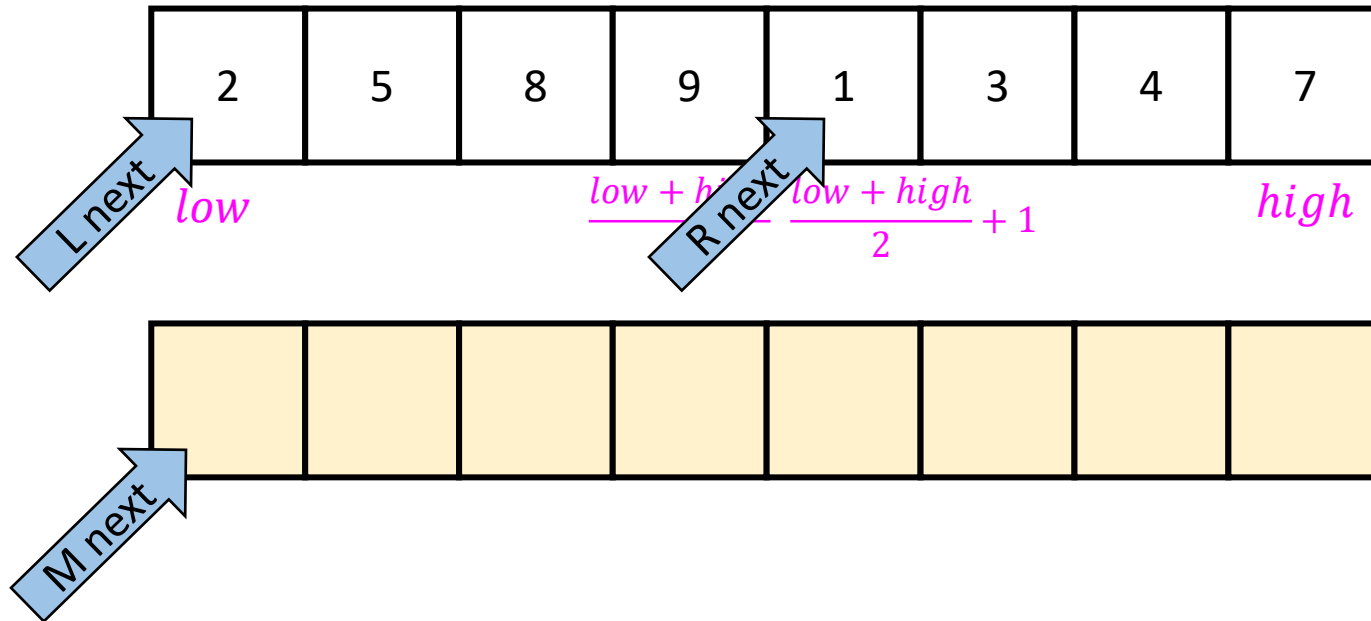
Divide and Conquer: Otherwise call mergesort on ranges $\left(\textit{low}, \frac{\textit{low} + \textit{high}}{2}\right)$ and $\left(\frac{\textit{low} + \textit{high}}{2} + 1, \textit{high}\right)$



After Recursion:



Merge (the combine part)



Create a **new array to merge into**, and 3 pointers/indices:

- **L_next**: the smallest “unmerged” thing on the left
- **R_next**: the smallest “unmerged” thing on the right
- **M_next**: where the next smallest thing goes in the merged array

One-by-one: put the smallest of **L_next** and **R_next** into **M_next**, then advance both **M_next** and whichever of **L/R** was used.

Merge Sort Pseudocode

```
void mergesort(myArray){
    ms_helper(myArray, 0, myArray.length());
}

void mshelper(myArray, low, high){
    if (low == high){return;} // Base Case
    mid = (low+high)/2;
    ms_helper(low, mid);
    ms_helper(mid+1, high);
    merge(myArray, low, mid, high);
}
```

Merge Pseudocode

```
void merge(myArray, low, mid, high){
    merged = new int[high-low+1]; // or whatever type is in myArray
    l_next = low;
    r_next = high;
    m_next = 0;
    while (l_next <= mid && r_next <= high){
        if (myArray[l_next] <= myArray[r_next]){
            merged[m_next++] = myArray[l_next++];
        }
        else{
            merged[m_next++] = myArray[r_next++];
        }
    }
    while (l_next <= mid){ merged[m_next++] = myArray[l_next++]; }
    while (r_next <= high){ merged[m_next++] = myArray[r_next++]; }
    for(i=0; i<=merged.length; i++){ myArray[i+low] = merged[i];}
}
```

Analyzing Merge Sort

1. Identify time required to Divide and Combine
2. Identify all subproblems and their sizes
3. Use recurrence relation to express recursive running time
4. Solve and express running time asymptotically

- **Divide:** 0 comparisons
- **Conquer:** recursively sort two lists of size $\frac{n}{2}$
- **Combine:** n comparisons
- **Recurrence:**

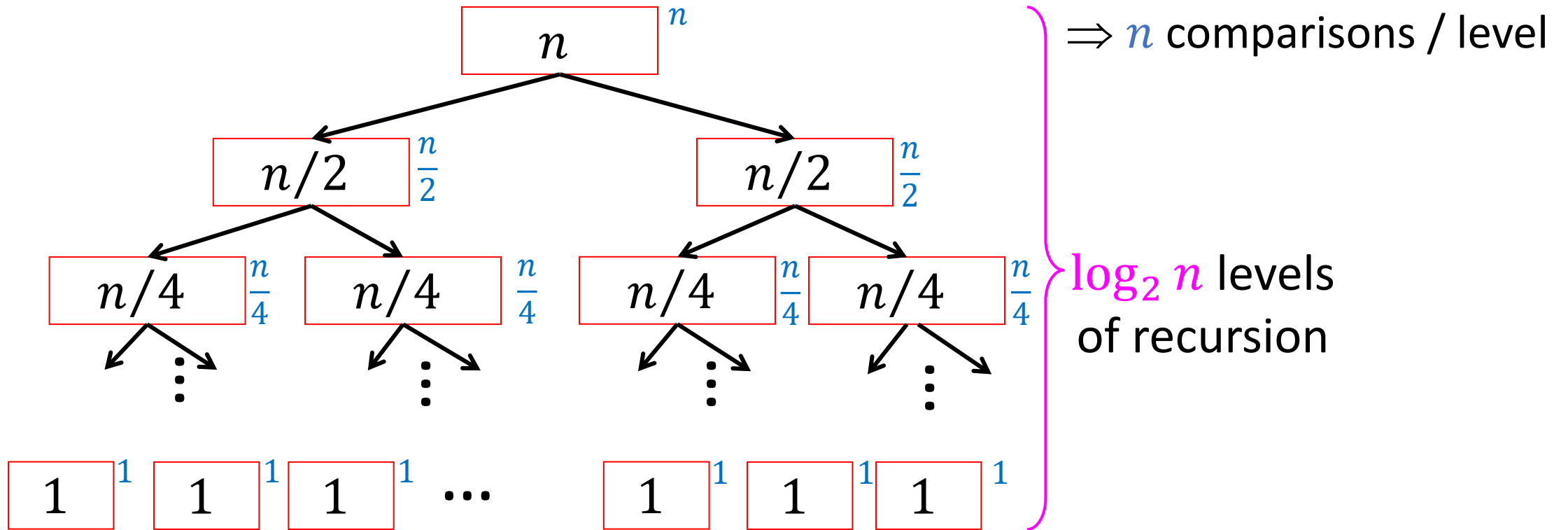
$$T(n) = 0 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

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

Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



$$T(n) = \sum_{i=1}^{\log_2 n} n = n \log_2 n$$

Properties of Merge Sort

- Worst Case Running time:
 - $\Theta(n \log n)$
- In-Place?
 - No!
- Adaptive?
 - No!
- Stable?
 - Yes!
 - As long as in a tie you always pick `l_next`

Quicksort

- Like Mergesort:
 - Divide and conquer
 - $O(n \log n)$ run time (kind of...)
- Unlike Mergesort:
 - Divide step is the “hard” part
 - *Typically* faster than Mergesort

Quicksort

Idea: pick a **pivot** element, recursively sort two sublists around that element

- **Divide:** select **pivot** element p , **Partition(p)**
- **Conquer:** recursively sort left and right sublists
- **Combine:** Nothing!

Partition (Divide step)

Given: a list, a pivot p

Start: unordered list

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

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

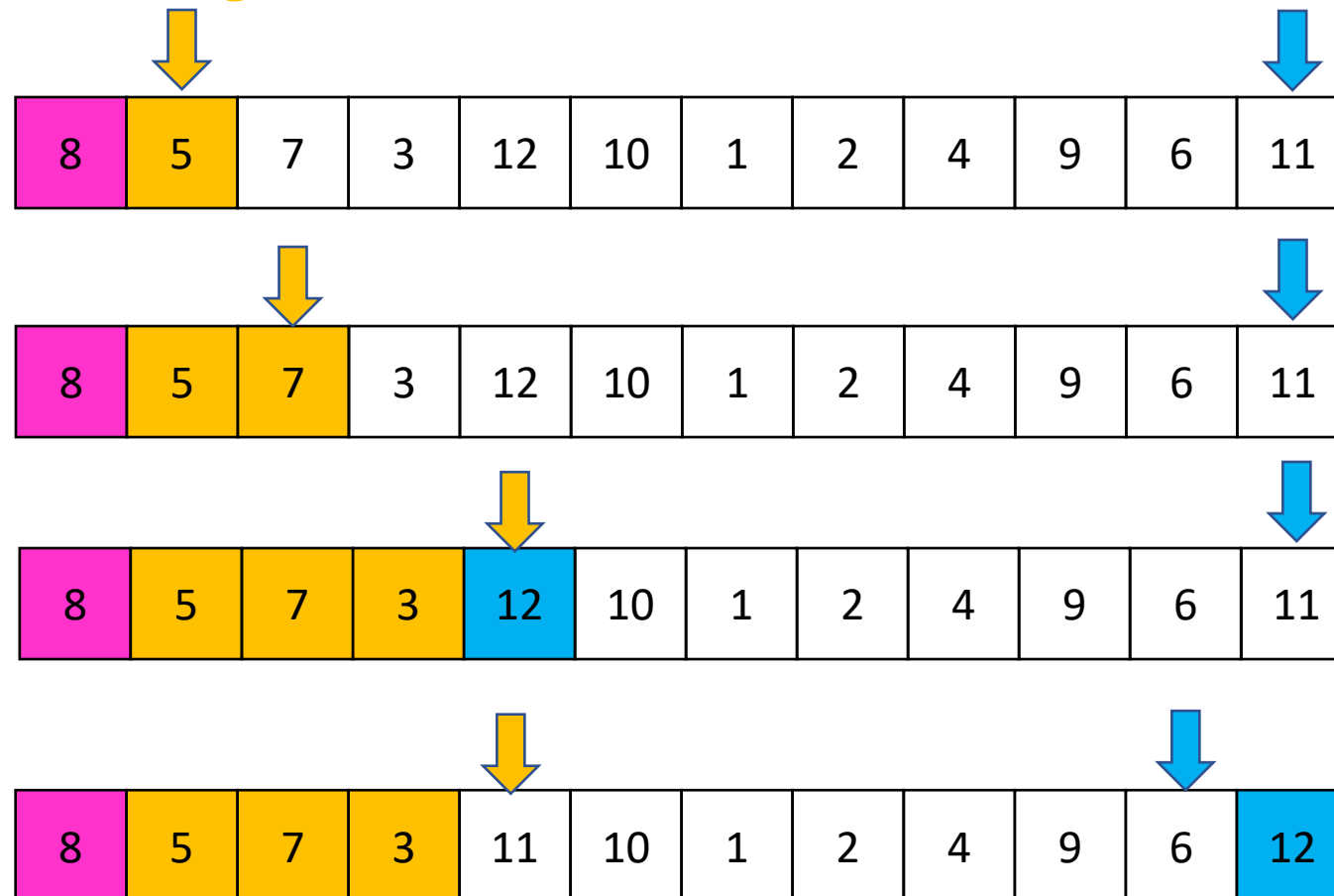
5	7	3	1	2	4	6	8	12	10	9	11
---	---	---	---	---	---	---	---	----	----	---	----

Partition, Procedure

If **Begin** value $< p$, move **Begin** right

Else swap **Begin** value with **End** value, move **End** Left

Done when **Begin** = **End**

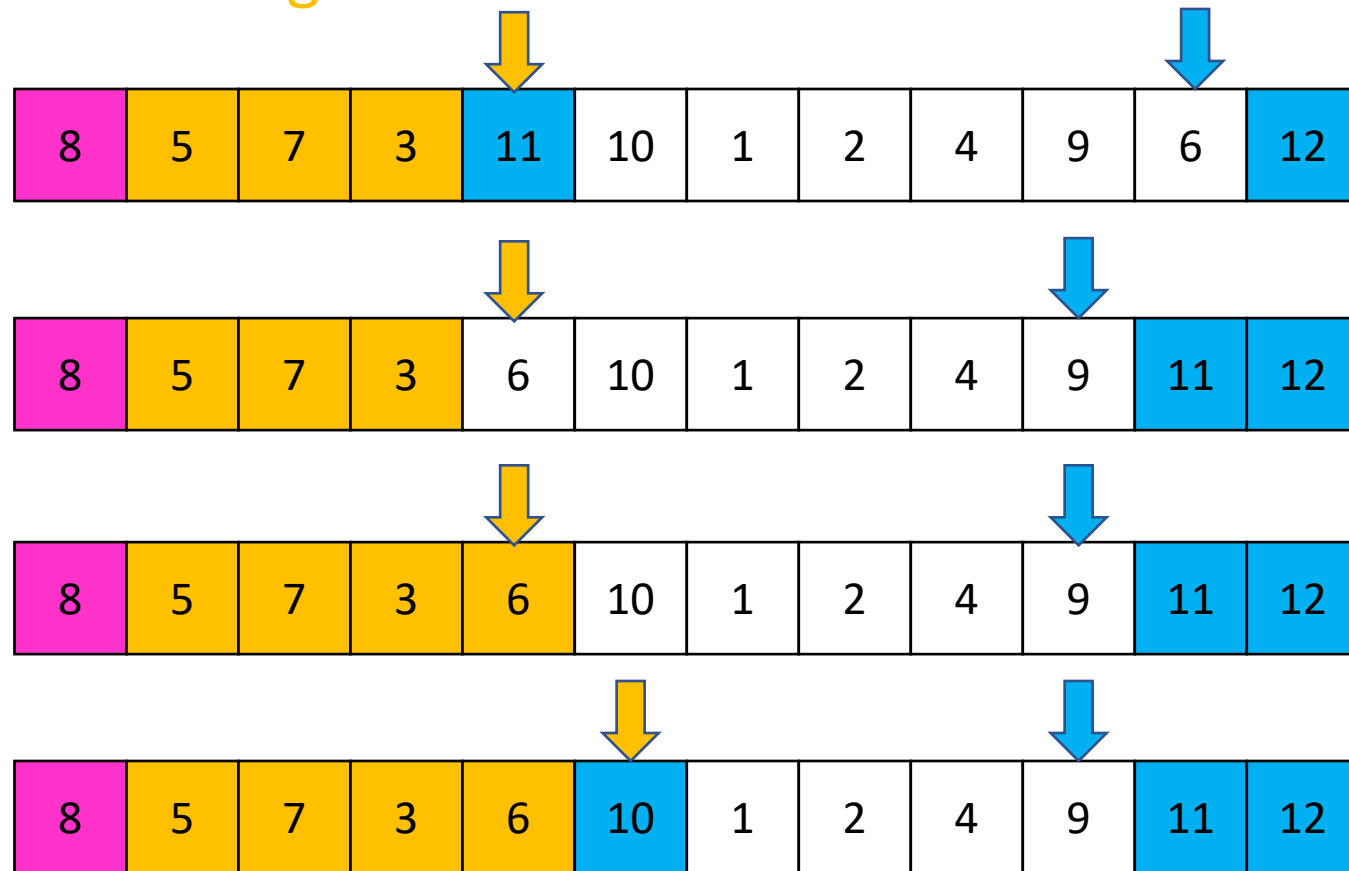


Partition, Procedure

If **Begin** value $< p$, move **Begin** right

Else swap **Begin** value with **End** value, move **End** Left

Done when **Begin** = **End**

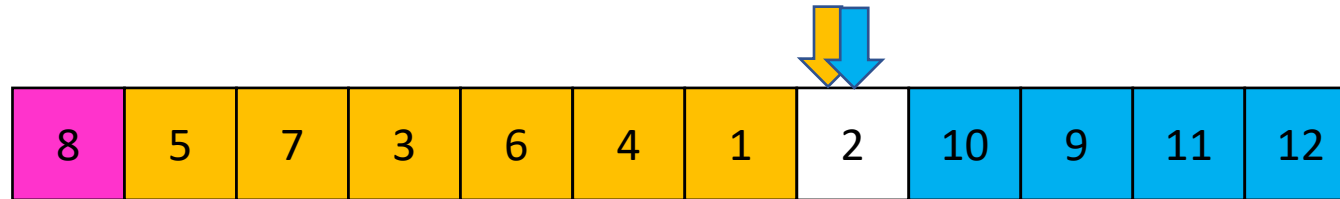


Partition, Procedure

If **Begin** value $< p$, move **Begin** right

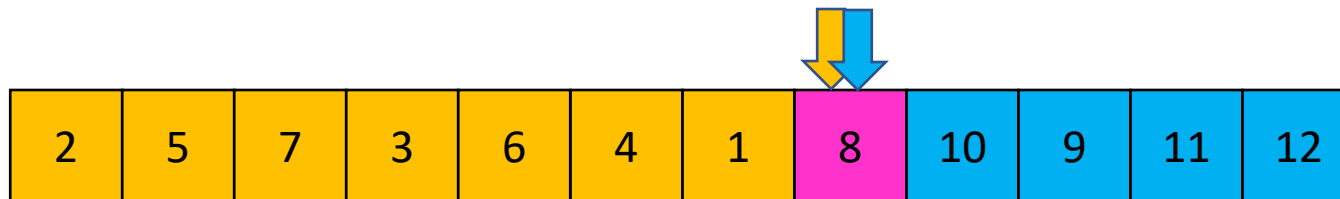
Else swap **Begin** value with **End** value, move **End** Left

Done when **Begin** = **End**



Case 1: meet at element $< p$

Swap p with **pointer position** (2 in this case)

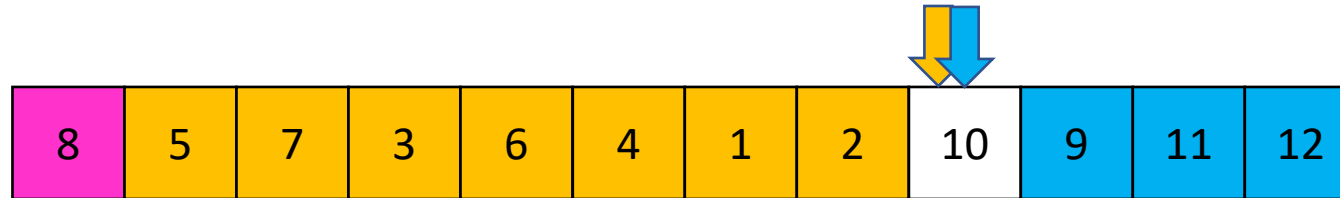


Partition, Procedure

If **Begin** value $< p$, move **Begin** right

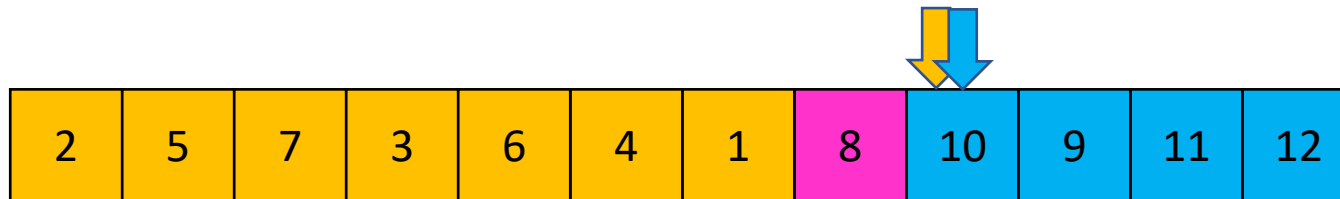
Else swap **Begin** value with **End** value, move **End** Left

Done when **Begin** = **End**



Case 2: meet at element $> p$

Swap p with **value to the left** (2 in this case)

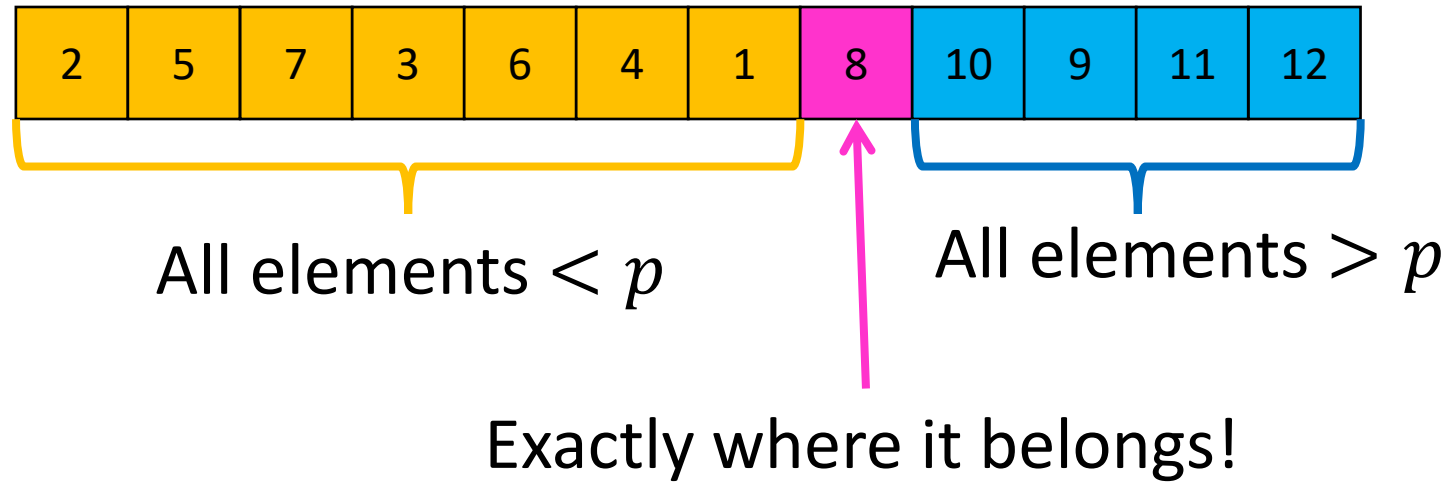


Partition Summary

1. Put p at beginning of list
2. Put a pointer (**Begin**) just after p , and a pointer (**End**) at the end of the list
3. While **Begin** < **End**:
 1. If **Begin** value < p , move **Begin** right
 2. Else swap **Begin** value with **End** value, move **End** Left
4. If pointers meet at element < p : Swap p with **pointer position**
5. Else If pointers meet at element > p : Swap p with **value to the left**

Run time? $O(n)$

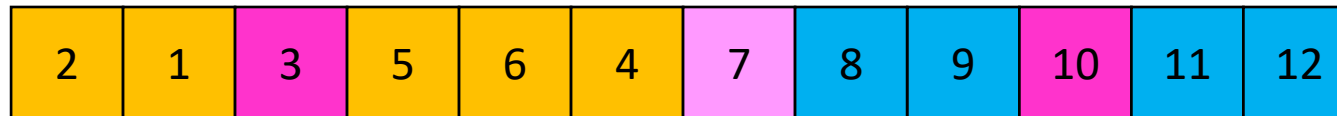
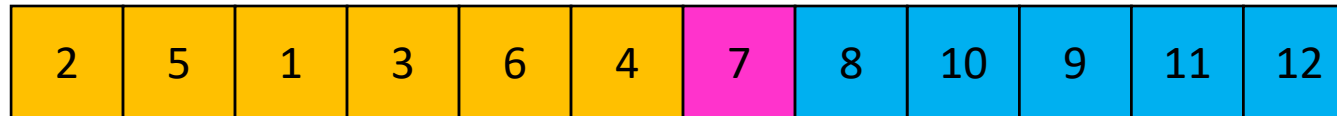
Conquer



Recursively sort **Left** and **Right** sublists

Quicksort Run Time (Best)

If the **pivot** is always the median:



Then we divide in half each time

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

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

Quicksort Run Time (Worst)

If the pivot is always at the extreme:



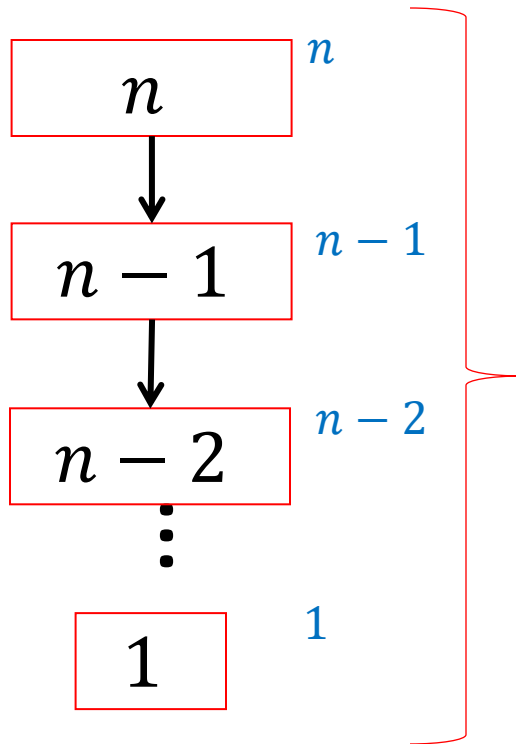
Then we shorten by 1 each time

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

$$T(n) = O(n^2)$$

Quicksort Run Time (Worst)

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



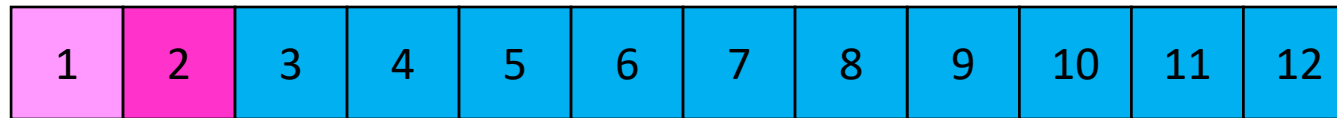
$$T(n) = 1 + 2 + 3 + \dots + n$$

$$T(n) = \frac{n(n + 1)}{2}$$

$$T(n) = O(n^2)$$

Quicksort on a (nearly) Sorted List

First element always yields unbalanced pivot



So we shorten by 1 each time

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

$$T(n) = O(n^2)$$

Good Pivot

- What makes a good Pivot?
 - Roughly even split between left and right
 - Ideally: median
- There are ways to find the median in linear time, but it's complicated and slow and you're better off using mergesort
- In Practice:
 - Pick a random value as a pivot
 - Pick the middle of 3 random values as the pivot

Properties of Quick Sort

- Worst Case Running time:
 - $\Theta(n^2)$
 - But $\Theta(n \log n)$ average! And typically faster than mergesort!
- In-Place?
 -Debatable
- Adaptive?
 - No!
- Stable?
 - No!

Improving Running time

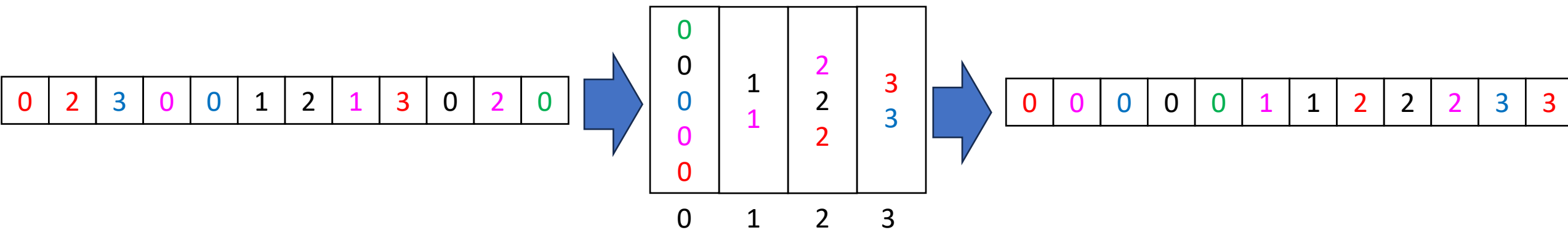
- Recall our definition of the sorting problem:
 - Input:
 - An array A of items
 - A comparison function for these items
 - Given two items x and y , we can determine whether $x < y$, $x > y$, or $x = y$
 - Output:
 - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
- Under this definition, it is impossible to write an algorithm faster than $n \log n$ asymptotically.
- Observation:
 - Sometimes there might be ways to determine the position of values without comparisons!

“Linear Time” Sorting Algorithms

- Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
 - Examples:
 - The list contains only positive integers less than k
 - The number of distinct values in the list is much smaller than the length of the list
- The running time expression will always have a term other than the list's length to account for this assumption
 - Examples:
 - Running time might be $\Theta(k \cdot n)$ where k is the range/count of values

BucketSort

- Assumes the array contains integers between 0 and $k - 1$ (or some other small range)
- Idea:
 - Use each value as an index into an array of size k
 - Add the item into the “bucket” at that index (e.g. linked list)
 - Get sorted array by “appending” all the buckets



BucketSort Running Time

- Create array of k buckets
 - Either $\Theta(k)$ or $\Theta(1)$ depending on some things...
- Insert all n things into buckets
 - $\Theta(n)$
- Empty buckets into an array
 - $\Theta(n + k)$
- Overall:
 - $\Theta(n + k)$
- When is this better than mergesort?

Properties of BucketSort

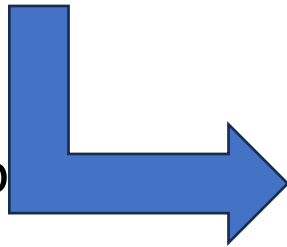
- In-Place?
 - No
- Adaptive?
 - No
- Stable?
 - Yes!

RadixSort

- Radix: The base of a number system
 - We'll use base 10, most implementations will use larger bases
- Idea:
 - BucketSort by each digit, one at a time, 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



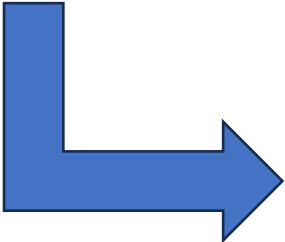
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999
0	1	2	3	4	5	6	7	8	9

RadixSort

- Radix: The base of a number system
 - We'll use base 10, most implementations will use larger bases
- Idea:
 - BucketSort by each digit, one at a time, from least significant to most significant

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

Place each element into a "bucket" according to its 10's place



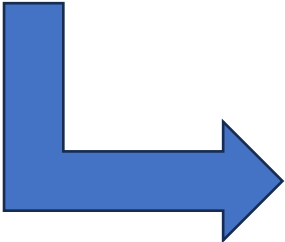
800									
801	512	121							
401	113	323		245	255				999
101	018	823			555				
901									
103									
0	1	2	3	4	5	6	7	8	9

RadixSort

- Radix: The base of a number system
 - We'll use base 10, most implementations will use larger bases
- Idea:
 - BucketSort by each digit, one at a time, from least significant to most significant

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

Place each element into a "bucket" according to its 100's place

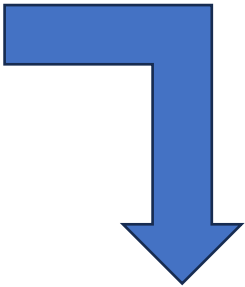


	101							800	901
018	103	245	323	401	512			801	999
	113	255			555			823	
	121								
0	1	2	3	4	5	6	7	8	9

RadixSort

- Radix: The base of a number system
 - We'll use base 10, most implementations will use larger bases
- Idea:
 - BucketSort by each digit, one at a time, from least significant to most significant

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



Convert back into an array

018	811	103	113	121	245	255	323	401	512	555	800	801	823	901	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

RadixSort Running Time

- Suppose largest value is m
- Choose a radix (base of representation) b
- BucketSort all n things using b buckets
 - $\Theta(n + k)$
- Repeat once per each digit
 - $\log_b m$ iterations
- Overall:
 - $\Theta(n \log_b m + b \log_b m)$
- In practice, you can select the value of b to optimize running time
- When is this better than mergesort?