

# CSE 332 Winter 2026

## Lecture 13: Sorting 2

Nathan Brunelle

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

# Sorting Definition

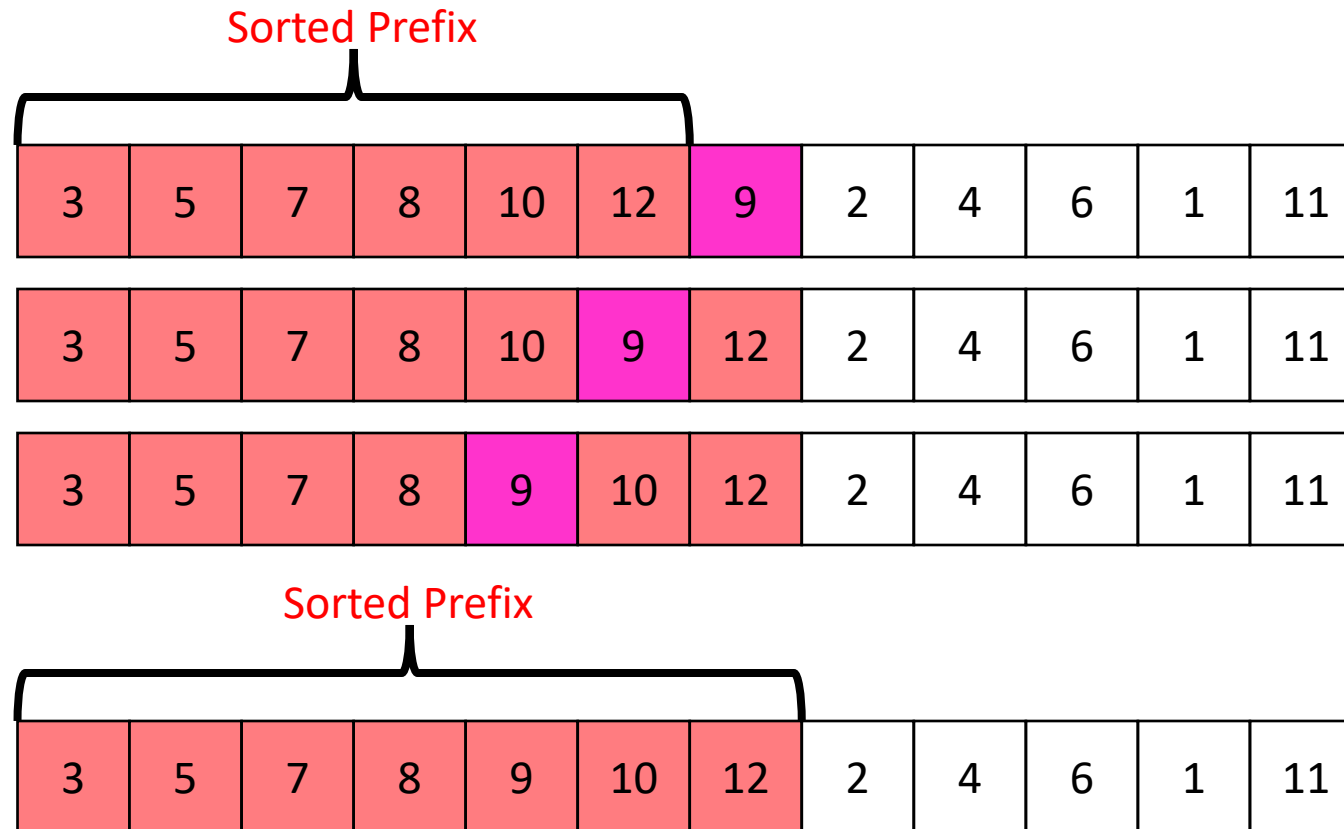
- Input:
  - An array  $A$  of items
  - A comparison function for these items
    - Given two items  $x$  and  $y$ , we can determine whether  $x < y$ ,  $x > y$ , or  $x = y$
    - *Disclaimer: The slides are using integers, but they could be other things, too!*
- Output:
  - A permutation of  $A$  such that if  $i \leq j$  then  $A[i] \leq A[j]$
  - Permutation: a sequence of the same items but perhaps in a different order

# Properties To Consider

- Worst case running time
- In place:
  - We only need to use the pre-existing array to do sorting
  - Constant extra space (only some additional variables needed)
  - *Insertion Sort, Heap Sort*
- Adaptive
  - The running improves as the given list is closer to being sorted
  - It should be linear time for a pre-sorted list, and nearly linear time if the list is nearly sorted
  - *Insertion Sort*
- Online
  - We can start sorting before we have the entire list.
  - *Insertion Sort*
- Stable
  - “Tied” elements keep their original order
  - *Insertion Sort*

# Insertion Sort

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



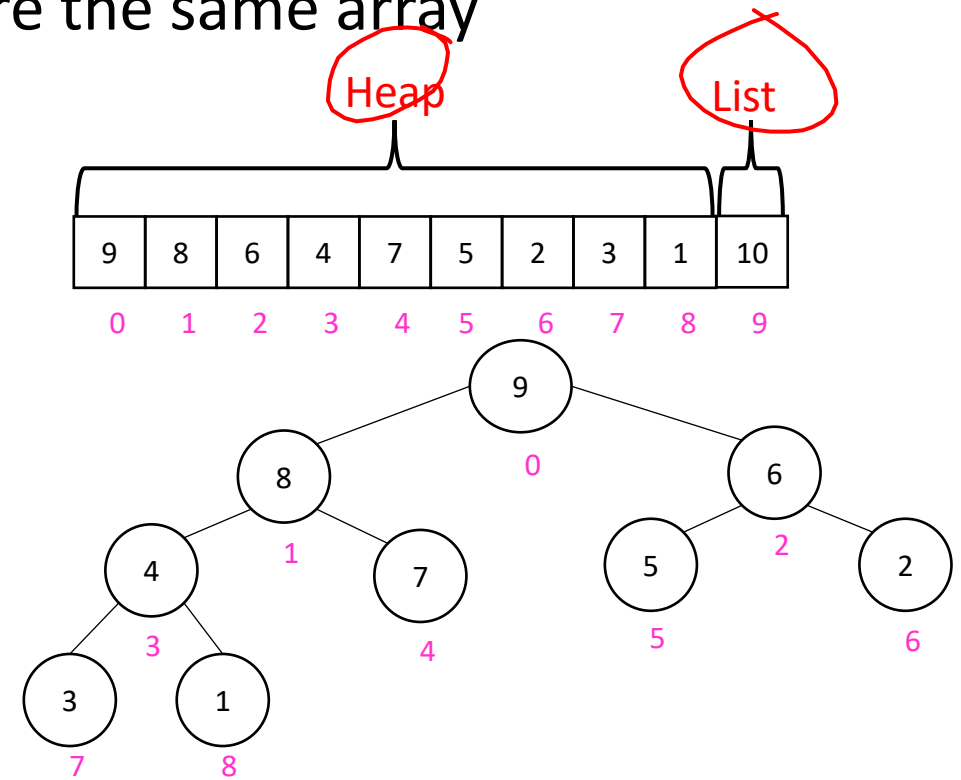
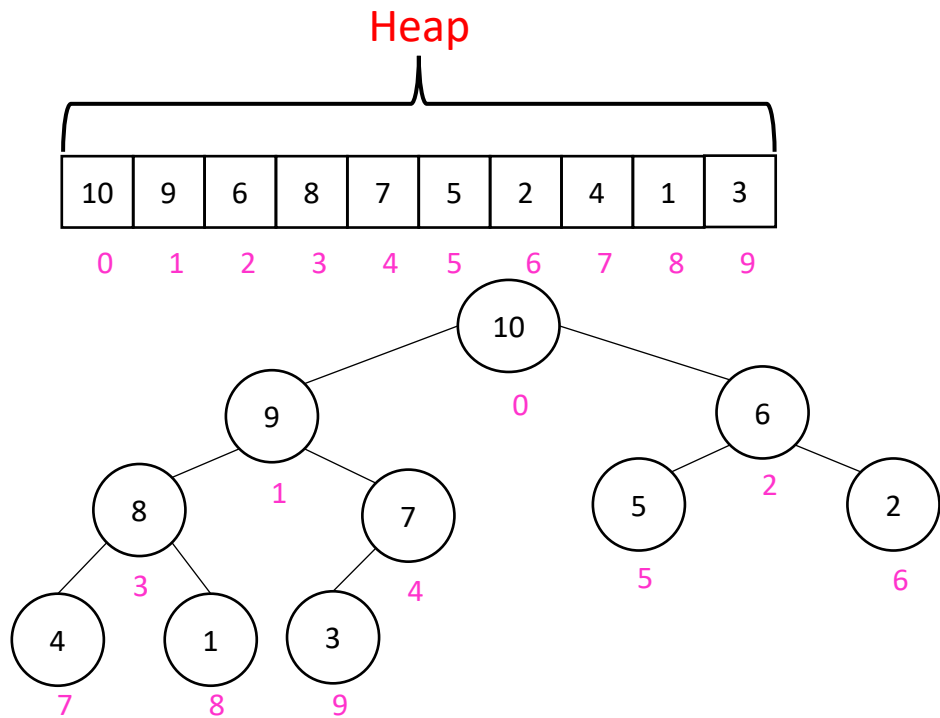
# Insertion Sort Properties

Summary: except for its asymptotic running time, it has everything you might want!

- Worst case running time
  - $\Theta(n^2)$
  - With VERY small constants! (No faster way of sorting a list of  $\leq 50$ ish elements!)
- In place:
  - YES!
  - We only swap items within the given array
- Adaptive
  - YES!
  - The only elements that move are the elements that are out of position
- Online
  - Yes!
  - Each time an item arrives, “insert” it into the sorted prefix
- Stable
  - YES!
  - If the item we’re inserting has a tie with its left neighbor, don’t swap

# Heap Sort

- **Idea:** Insert all items into a max heap, extract the largest one-by-one to fill sorted list from end to beginning, all the while the heap and sorted list share the same array



# Heap Sort Properties

- Worst case running time

→ •  $\Theta(n \log n)$

- In place:

→ • YES!  
• We only swap items within the given array

- Adaptive

→ • NO!  
• Running time is the same regardless of original list's order

- Online

→ • NO!  
• We need all elements in the heap before we can start extracting

- Stable

→ • NO!  
• Question on Exercise 6

←

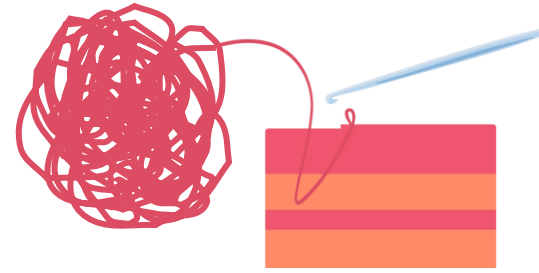
# Divide And Conquer Sorting

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

- Divide and Conquer:
  - Recursive algorithm design technique
  - Solve a large problem by breaking it up into smaller versions of the same problem

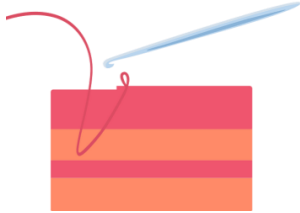


# Divide and Conquer



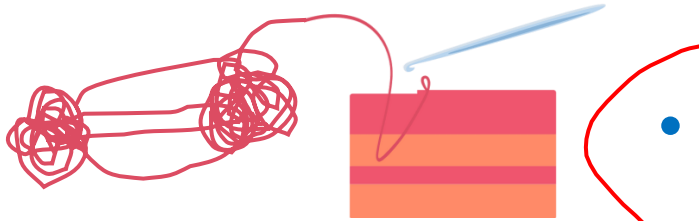
- **Base Case:**

- If the problem is “small” then solve directly and return



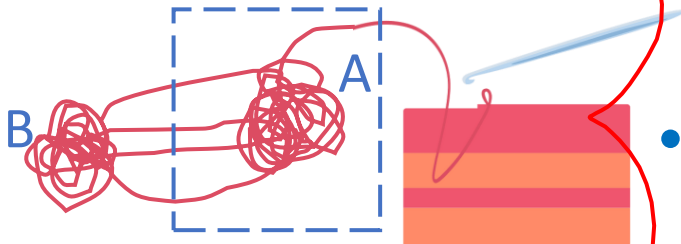
- **Divide:**

- Break the problem into subproblem(s), each smaller instances



- **Conquer:**

- Solve subproblem(s) recursively



- **Combine:**

- Use solutions to subproblems to solve original problem

# Divide and Conquer Template Pseudocode

```
def my_DandC(problem){  
  // Base Case  
  if (problem.size() <= small_value){  
    return solve(problem); // directly solve (e.g., brute force)  
  }  
  
  // Divide  
  List subproblems = divide(problem);  
  
  // Conquer  
  solutions = new List();  
  for (sub : subproblems){  
    subsolution = my_DandC(sub);  
    solutions.add(subsolution);  
  }  
  
  // Combine  
  return combine(solutions);  
}
```

# Merge Sort

5	8	2	9	4	1
---	---	---	---	---	---

5
---

- **Base Case:**

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

5	8	2	9	4	1
---	---	---	---	---	---

- **Divide:**

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

2	5	8	1	4	9
---	---	---	---	---	---

- **Conquer:**

- Sort both lists recursively

2	5	8	1	4	9
---	---	---	---	---	---

- **Combine:**

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

1	2	4	5	8	9
---	---	---	---	---	---

# Merge Sort In Action!

Sort between indices *low* and *high*

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

*low* *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  $\left(\frac{\textit{low} + \textit{high}}{2} + 1, \textit{high}\right)$

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

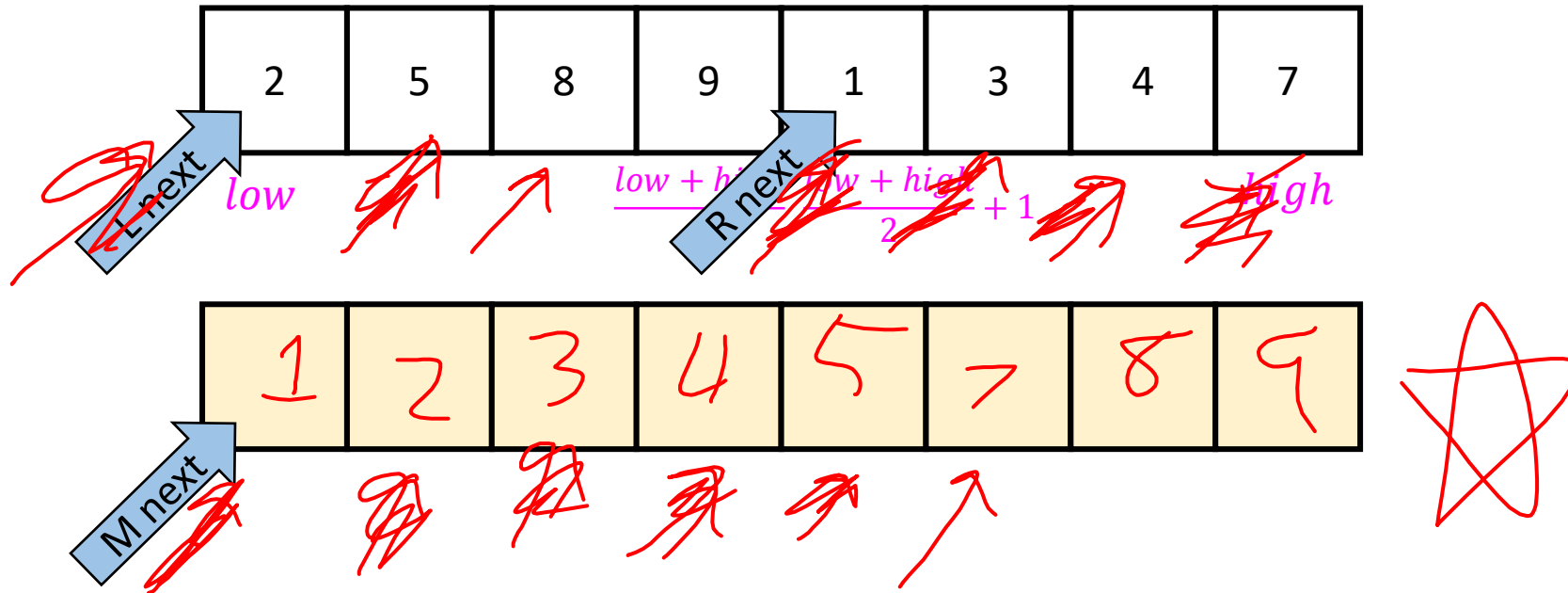
*low*  $\frac{\textit{low} + \textit{high}}{2}$   $\frac{\textit{low} + \textit{high}}{2} + 1$  *high*

After Recursion:

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

*low* *high*

# 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:**

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

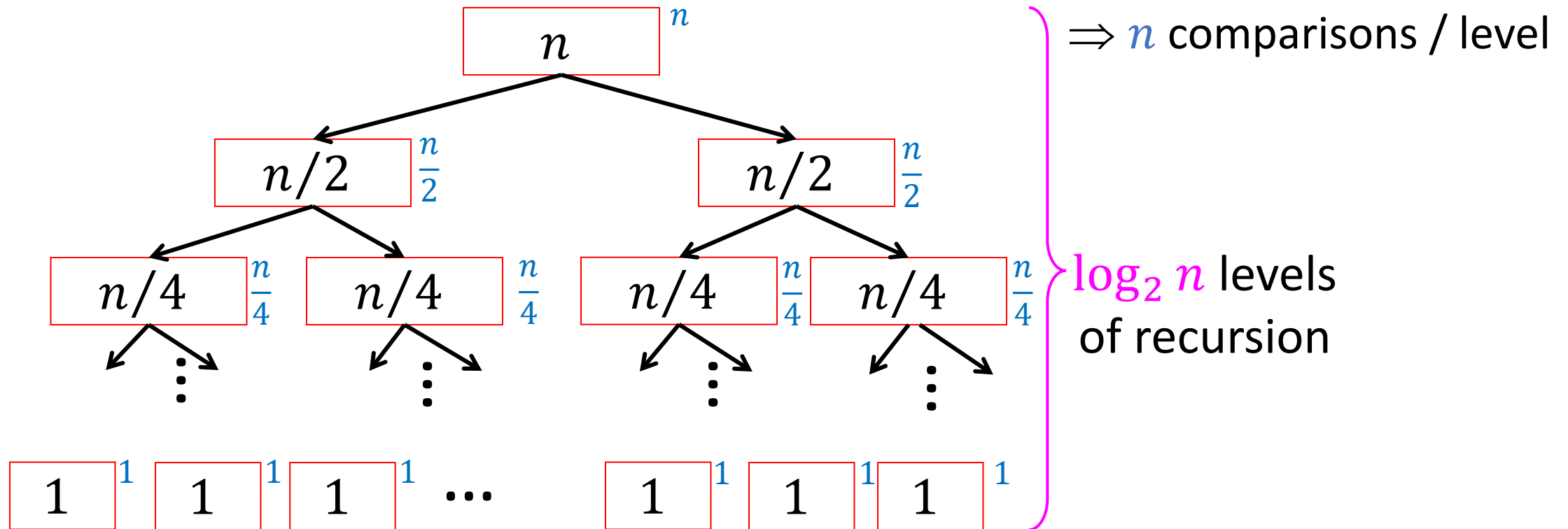
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



Red box represents a problem instance

Blue value represents time spent at that level of recursion

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



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

# Merge Sort Properties

- Worst case running time
  - $\Theta(n \log n)$
- In place:
  - NO!
  - We need a second array to merge into
- Adaptive
  - NO!
- Running time is the same regardless of original list's order
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - YES!
  - When merging, and there's a tie, choose the element from the left

# 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

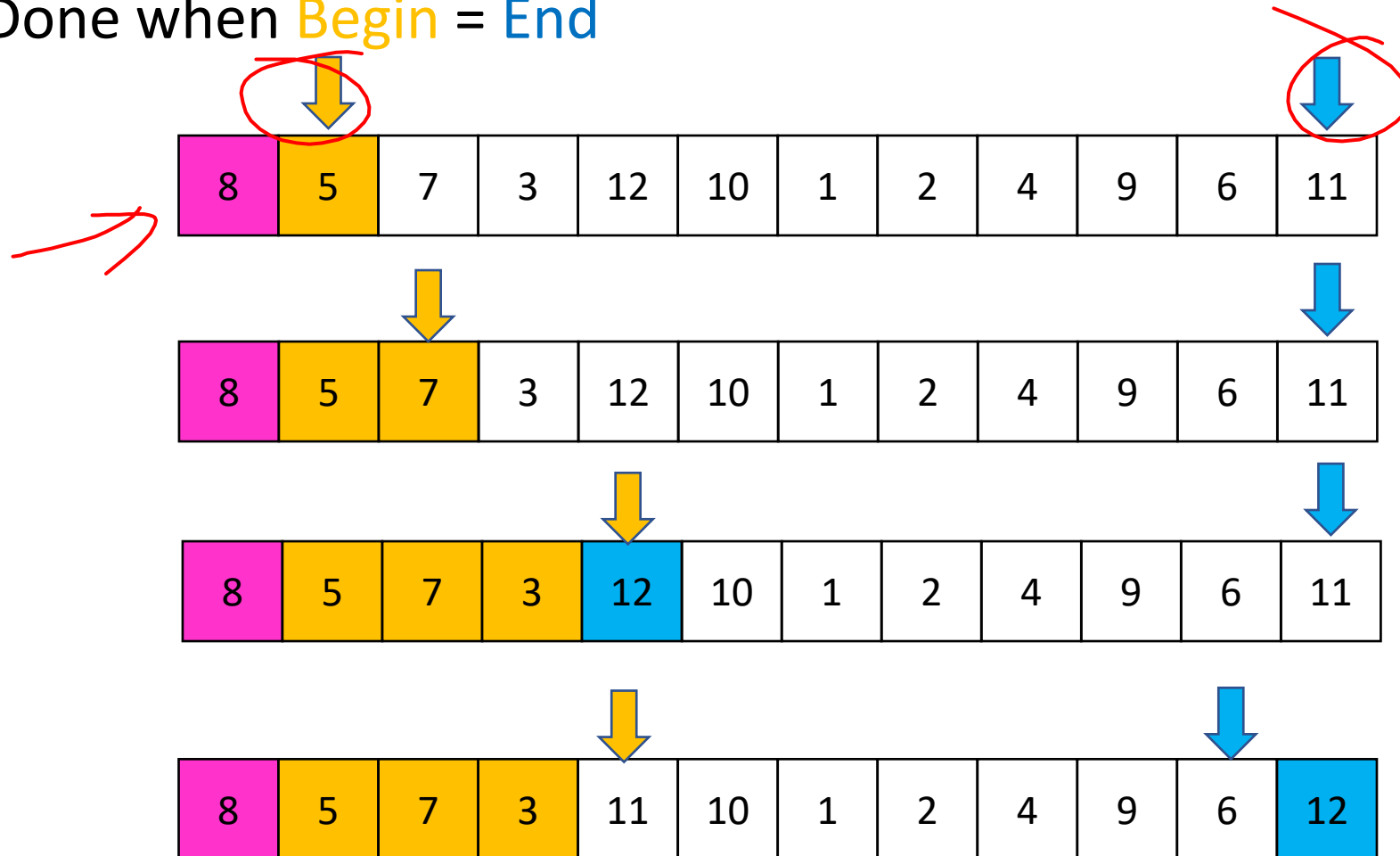
5	7	3	1	2	4	6	8	12	10	9	11
---	---	---	---	---	---	---	---	----	----	---	----

# Partition, Procedure

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

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

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

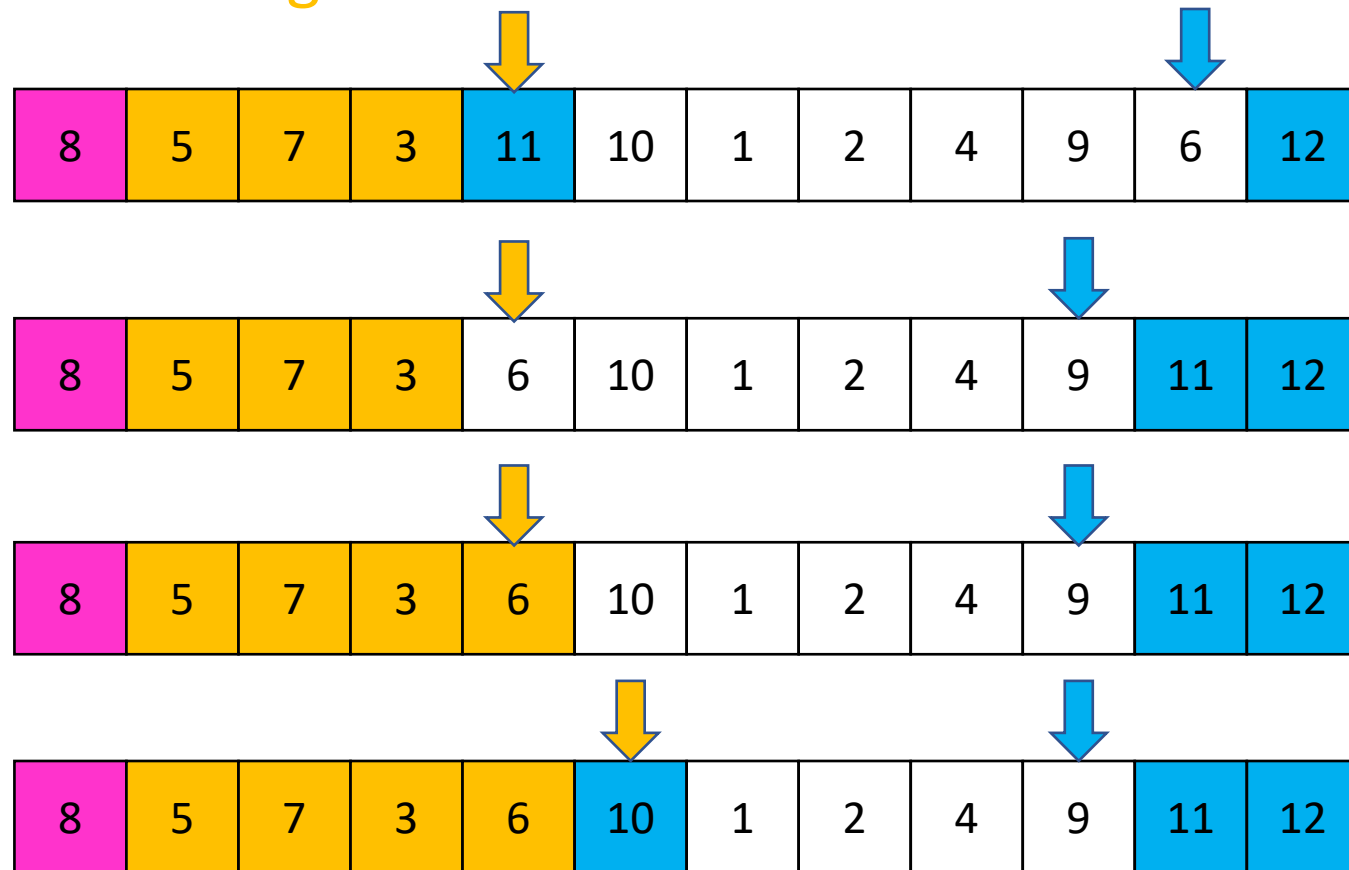


# Partition, Procedure

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

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

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

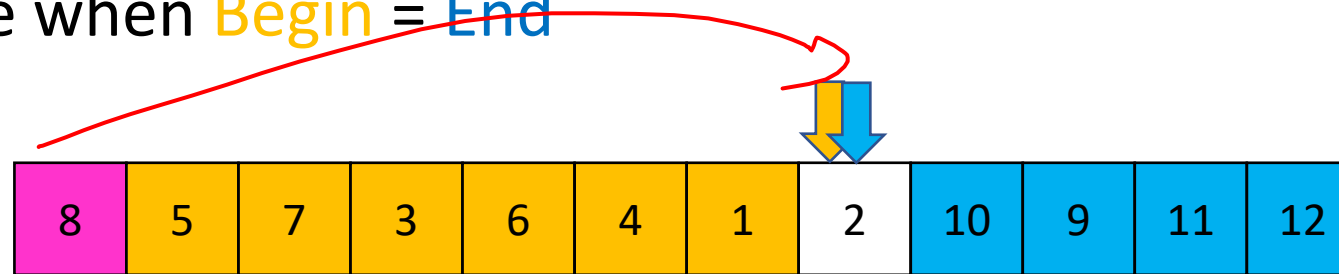


# Partition, Procedure

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

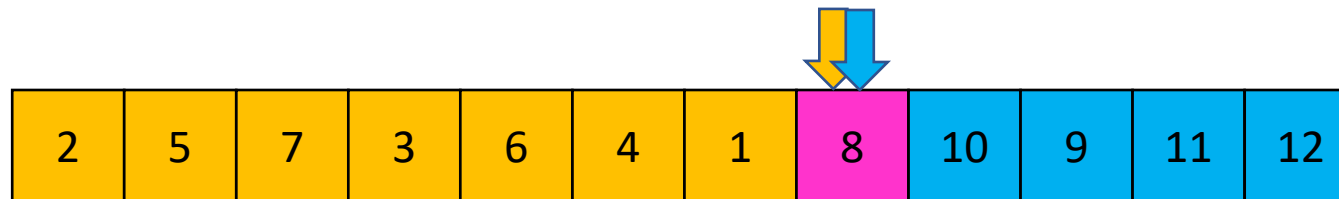
Else swap **Begin** value with **End** value, move **End** Left

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



Case 1: meet at element <  $p$

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



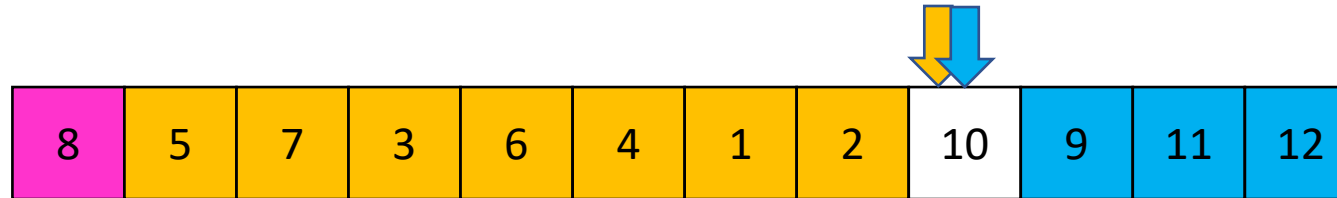


# Partition, Procedure

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

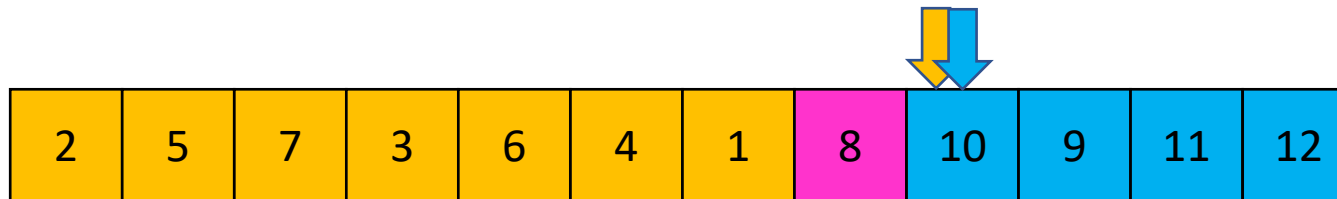
Else swap **Begin** value with **End** value, move **End** Left

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



Case 2: meet at element  $> p$

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

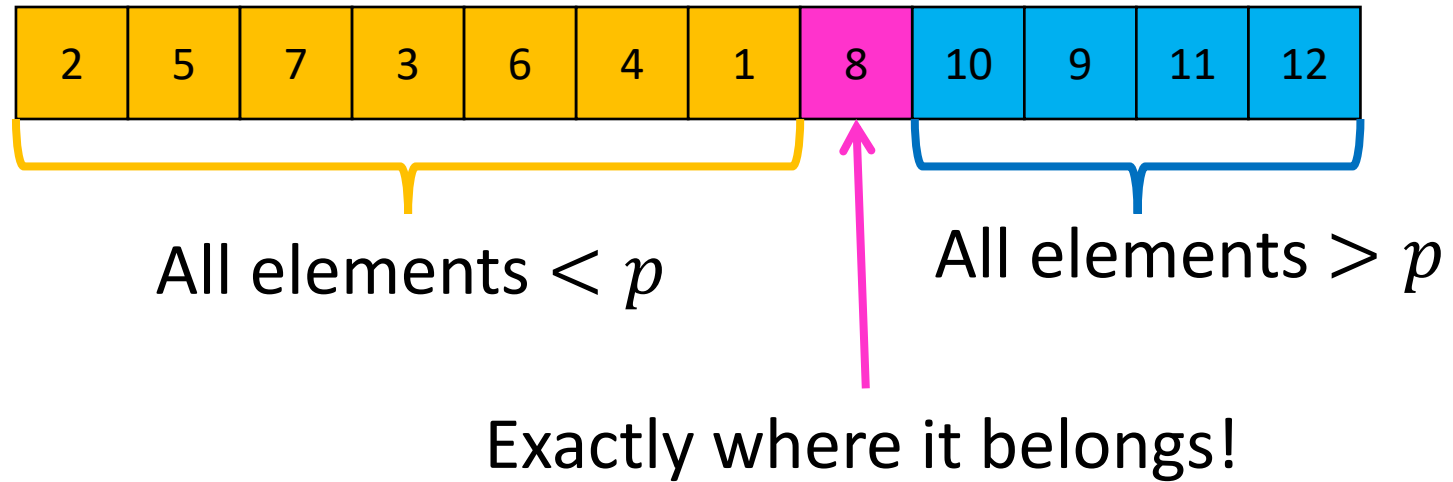


# Partition Summary

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:

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

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

Then we divide in half each time

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

# Quicksort Run Time (Worst)

If the pivot is always at the extreme:

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

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

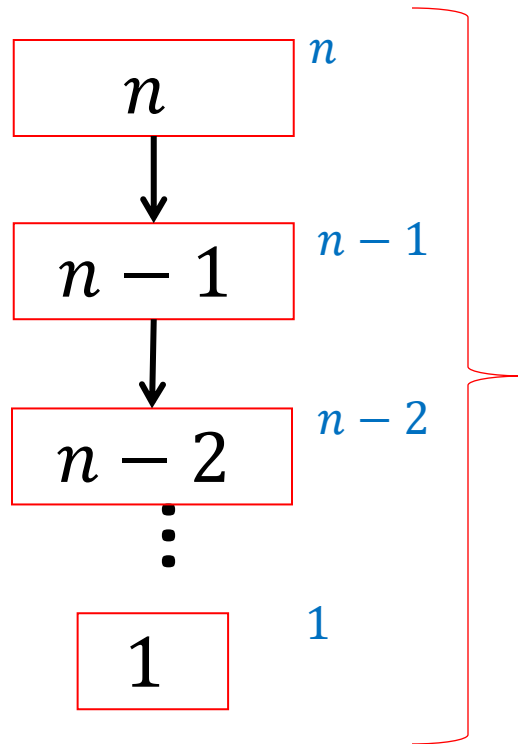
Then we shorten by 1 each time

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

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

# 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

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

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

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



# Quick Sort Properties

- Worst case running time
  - $\Theta(n^2)$ , but “almost always”  $\Theta(n \log n)$  in practice
  - Better constants than merge sort
- In place:
  - Kinda...
  - We swap within the given array, but do so using recursion, so we need space for the stack frames
  - Different textbooks disagree on whether call-stack space “counts”
- Adaptive
  - NO!
- Online
  - NO!
  - We need all elements before we can divide
- Stable
  - NO!
  - Partition procedure may rearrange tied elements

# Improving Running time

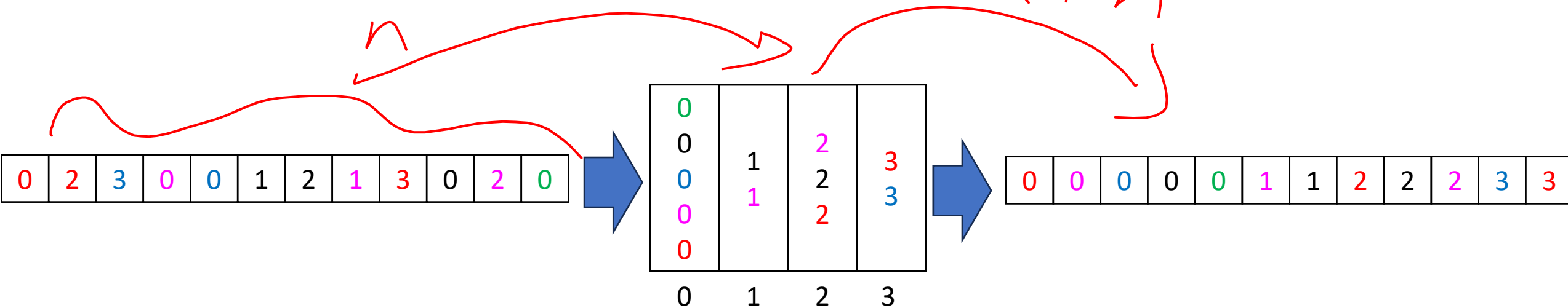
- Recall our definition of the sorting problem:
  - Input:
    - An array  $A$  of items
    - A comparison function for these items
      - Given two items  $x$  and  $y$ , we can determine whether  $x < y$ ,  $x > y$ , or  $x = y$
  - Output:
    - A permutation of  $A$  such that if  $i \leq j$  then  $A[i] \leq A[j]$
- 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

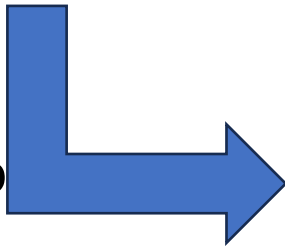
- In-Place?
  - No
- Adaptive?
  - No
- Stable?
  - Yes!

# 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



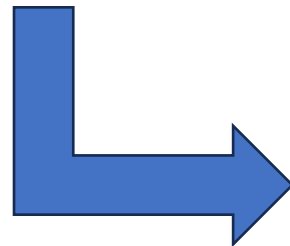
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999
0	1	2	3	4	5	6	7	8	9

# 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

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

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



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

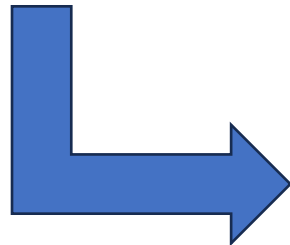


# RadixSort

- 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

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

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

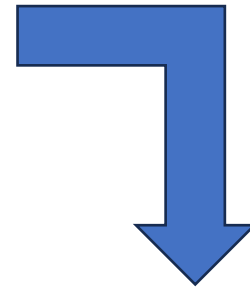


	101							800	
018	103	245	323	401	512			801	901
	113	255			555			823	999
	121								
0	1	2	3	4	5	6	7	8	9

# 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?