CSE 332 Autumn 2024
Lecture 16: Sorting 3

Nathan Brunelle

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

Warm up

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

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

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

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

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

$$\begin{array}{ccccccccccc} n! &=& n &\cdot& (n-1) &\cdot& (n-2) &\cdot \ldots \cdot& 2 &\cdot& 1 \\ && \parallel && \wedge && \wedge && \wedge && \wedge \\ n^n &=& n &\cdot& n &\cdot& n &\cdot \ldots \cdot& n &\cdot& n \end{array}$$

$n! \leq n^n$

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

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

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

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

$$
\begin{array}{ccccccccccccccccc}
n! & = & n & \cdot & (n-1) & \cdot & (n-2) & \cdot \ldots \cdot & \frac{n}{2} & \cdot & \left(\frac{n}{2}-1\right) & \cdot \ldots \cdot & 2 & \cdot & 1 \\
 & & \vee & & \vee & & \vee & & \parallel & & \vee & & \vee & & \parallel \\
\left(\frac{n}{2}\right)^{\frac{n}{2}} & = & \frac{n}{2} & \cdot & \frac{n}{2} & \cdot & \frac{n}{2} & \cdot \ldots \cdot & \frac{n}{2} & \cdot & 1 & \cdot \ldots \cdot & 1 & \cdot & 1
\end{array}
$$

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

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

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

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

Sorting Algorithm Summary

Algorithm	Running Time	Adaptive?	In-Place?	Stable?	Online?
Selection	n^2	No	Yes	No	No
Insertion	n^2	Yes	Yes	Yes	Yes
Heap	$n \log n$	No	Yes	No	No
Merge	$n \log n$	No	No	Yes	No
Quick					

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

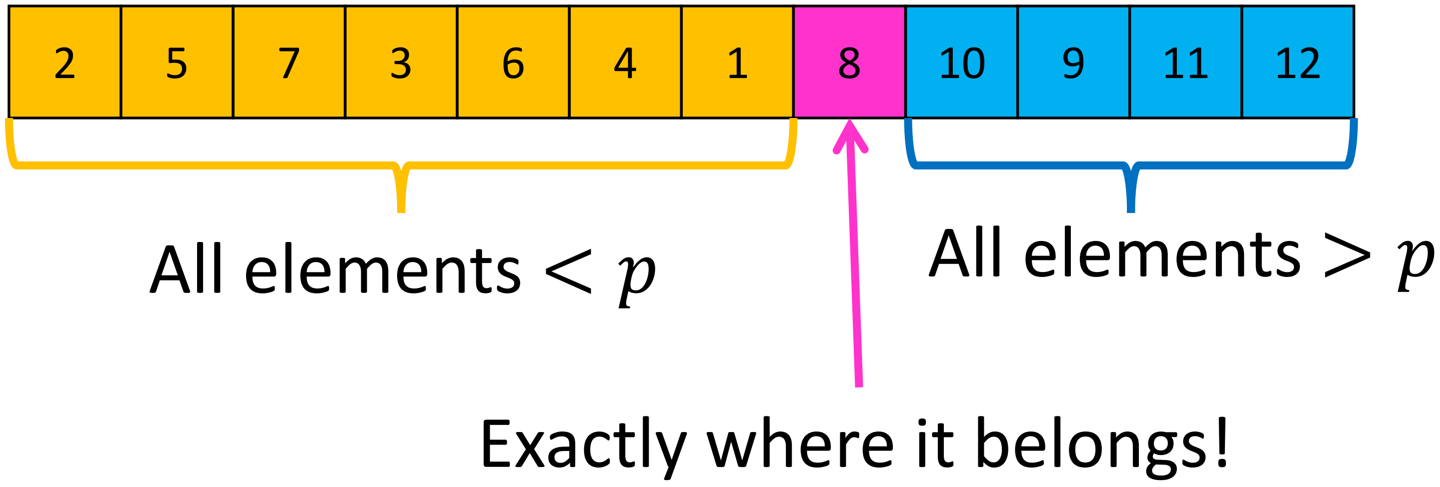
5	7	3	1	2	4	6	8	12	10	9	11

Partition 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

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

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

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

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

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

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:
- In-Place?
- Adaptive?
- Stable?
    -

Sorting Algorithm Summary

Algorithm	Running Time	Adaptive?	In-Place?	Stable?	Online?
Selection	n^2	No	Yes	No	No
Insertion	n^2	Yes	Yes	Yes	Yes
Heap	$n \log n$	No	Yes	No	No
Merge	$n \log n$	No	No	Yes	No
Quick	$n \log n$ (expected)	No	No*	No	No

*Quick Sort can be done in-place within each stack frame. Some textbooks do not include the memory occupied by the stack frame in space analysis, which would mean concluding Quick Sort is in-place. Others will include stack frame space, and therefore conclude Quick Sort is not in-place. If you try to implement it iteratively, you'll need another array somewhere (e.g. to store locations of sub-lists)

Worst Case Lower Bounds

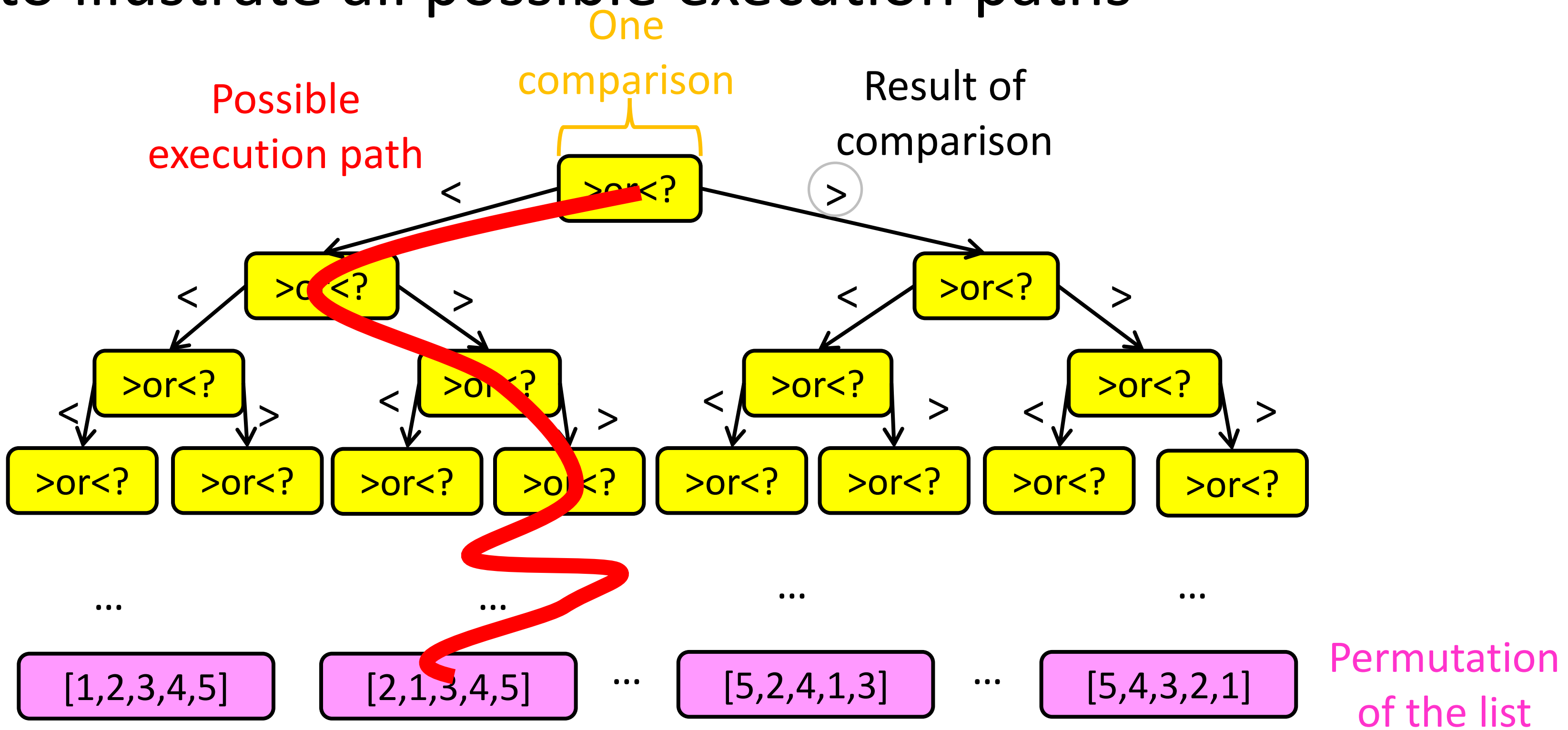
- Prove that there is no algorithm which can sort faster than $O(n \log n)$
  - Every algorithm, in the worst case, must have a certain lower bound
- Non-existence proof!
  - Very hard to do

Sorting Algorithm "Template"

- Compare two things ($a < b$)
- Based on response (true or false), compare two other things ($c_t < d_t$ or $c_f < d_f$)
- Based on that response, compare two more things ($e_{tt} < d_{tt}$, or $e_{tf} < d_{tf}$, or $e_{ft} < d_{ft}$, or $e_{ff} < d_{ff}$)
- Repeat until we know the correct order of elements
- Examples:
  - Quick Sort: compare the pivot to arr[1], then either compare the pivot to arr[2] or the item that was previously at arr[n-1].
  - Insertion Sort: compare arr[0] with arr[1]. Then compare arr[1] with arr[2]. Next either compare arr[1] with arr[0] or arr[3] with arr[2].

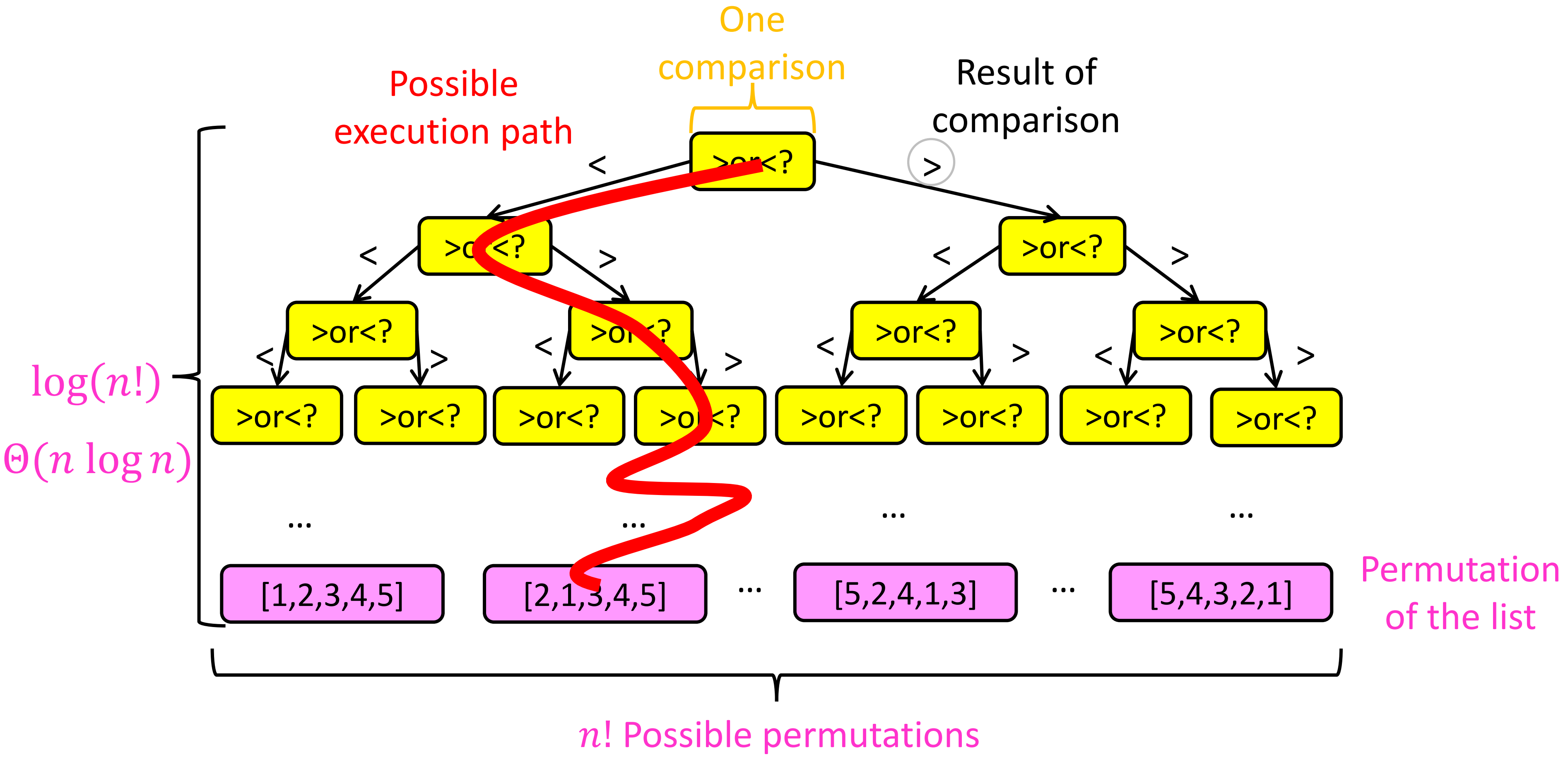
Strategy: Decision Tree

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



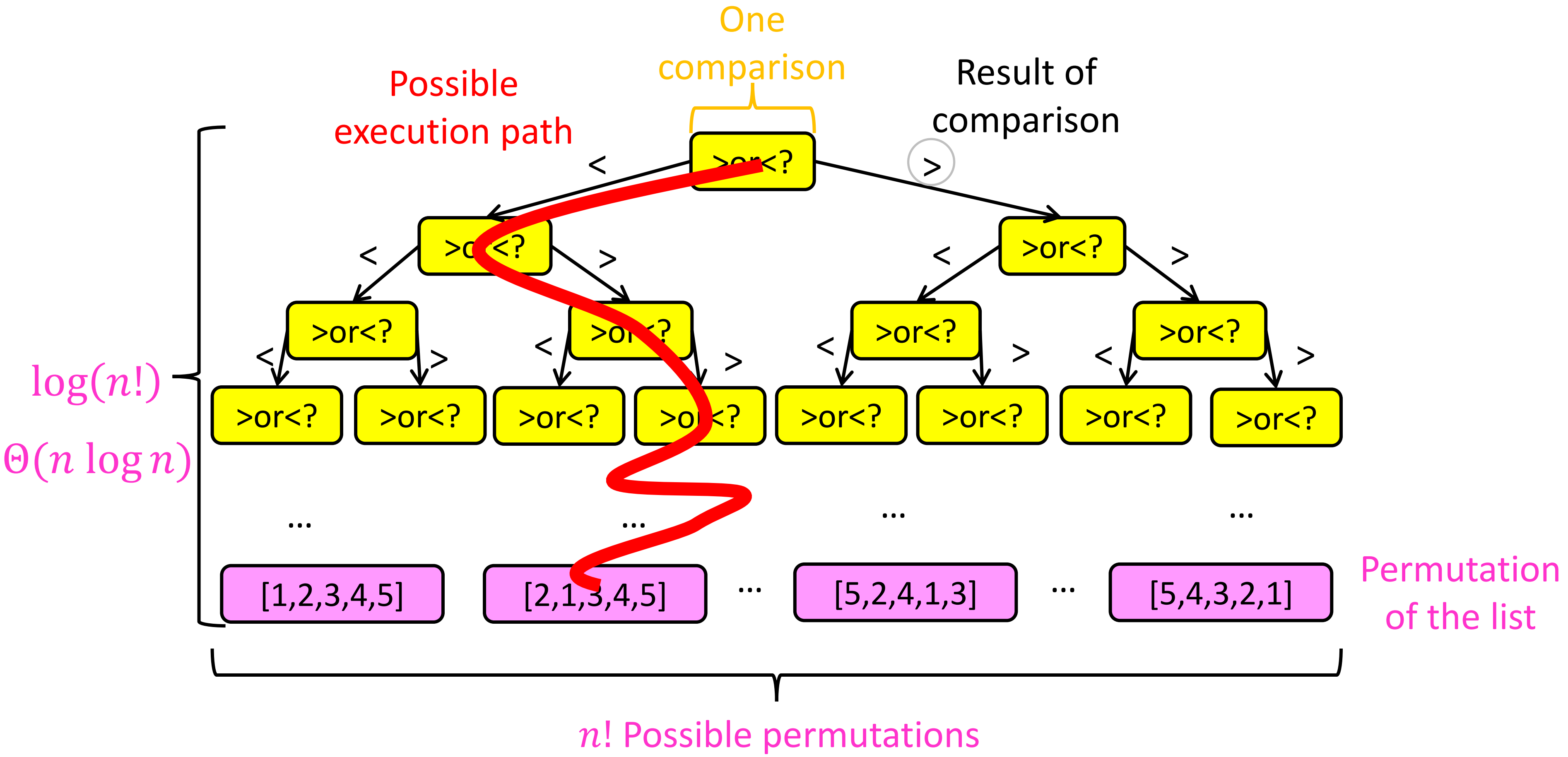
Strategy: Decision Tree

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



Strategy: Decision Tree

- Conclusion: Worst Case Optimal run time of sorting is $\Theta(n \log n)$
  - There is no (comparison-based) sorting algorithm with running time better than $n \log n$



Improving Running time

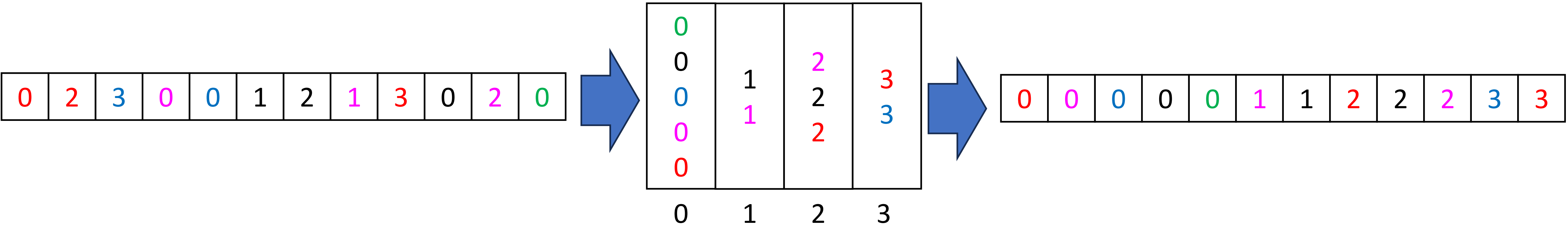
- Recall our definition of the sorting problem:
  - Input:
    - An array A of items
    - A comparison function for these items
      - Given two items x and y, we can determine whether $x < y$, $x > y$, or $x = y$
  - Output:
    - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
-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

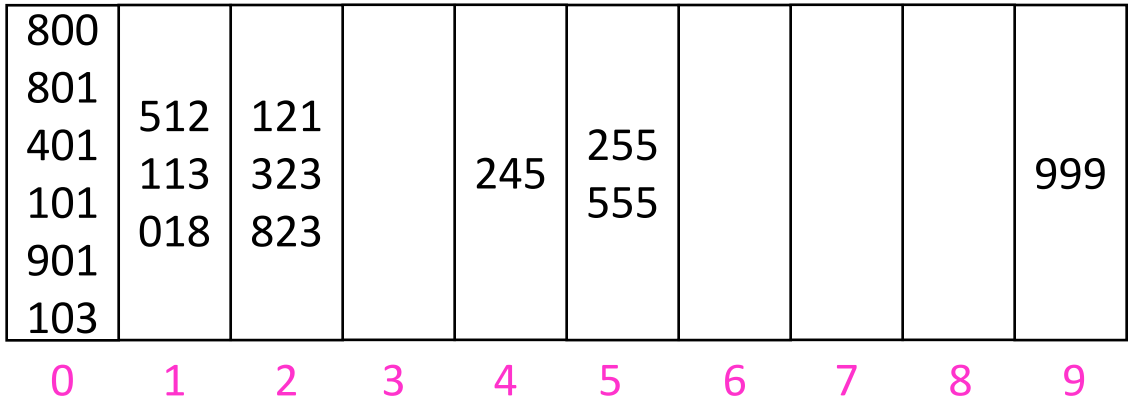
0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

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

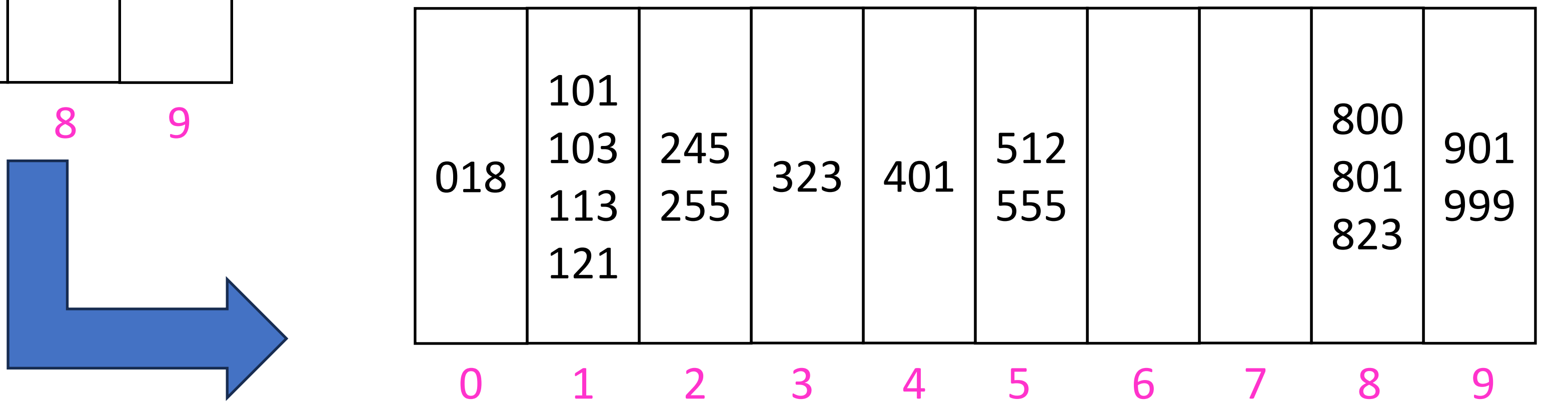
0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

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



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



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

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

Convert back into an array

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

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?