Sorting and Asymptotic Complexity

Lecture 13

CS2110 – Fall 2008

Prelim Tonight

InsertionSort

```
//sort a[], an array of int
for (int i = 1; i < a.length; i++) {
    int temp = a[i];
    int k;
    for (k = i; 0 < k && temp < a[k-1]; k--)
        a[k] = a[k-1];
    a[k] = temp;
}
```

- Many people sort cards this way
- Invariant: everything to left of `i` is already sorted
- Works especially well when input is *nearly sorted*

- Worst-case is $O(n^2)$
  - Consider reverse-sorted input
- Best-case is $O(n)$
  - Consider sorted input
- Expected case is $O(n^2)$
  - Expected number of inversions is $n(n–1)/4$

SelectionSort

- To sort an array of size n:
  - Examine `a[0]` to `a[n-1]`; find the smallest one and swap it with `a[0]`
  - Examine `a[1]` to `a[n-1]`; find the smallest one and swap it with `a[1]`
  - In general, in step i, examine `a[i]` to `a[n-1]`; find the smallest one and swap it with `a[i]`
- This is the other common way for people to sort cards
- Runtime
  - Worst-case $O(n^2)$
  - Best-case $O(n^2)$
  - Expected-case $O(n^2)$

Divide & Conquer?

- It often pays to
  - Break the problem into smaller subproblems,
  - Solve the subproblems separately, and then
  - Assemble a final solution

- This technique is called *divide-and-conquer*
  - Caveat: It won’t help unless the *partitioning* and *assembly* processes are inexpensive

- Can we apply this approach to sorting?

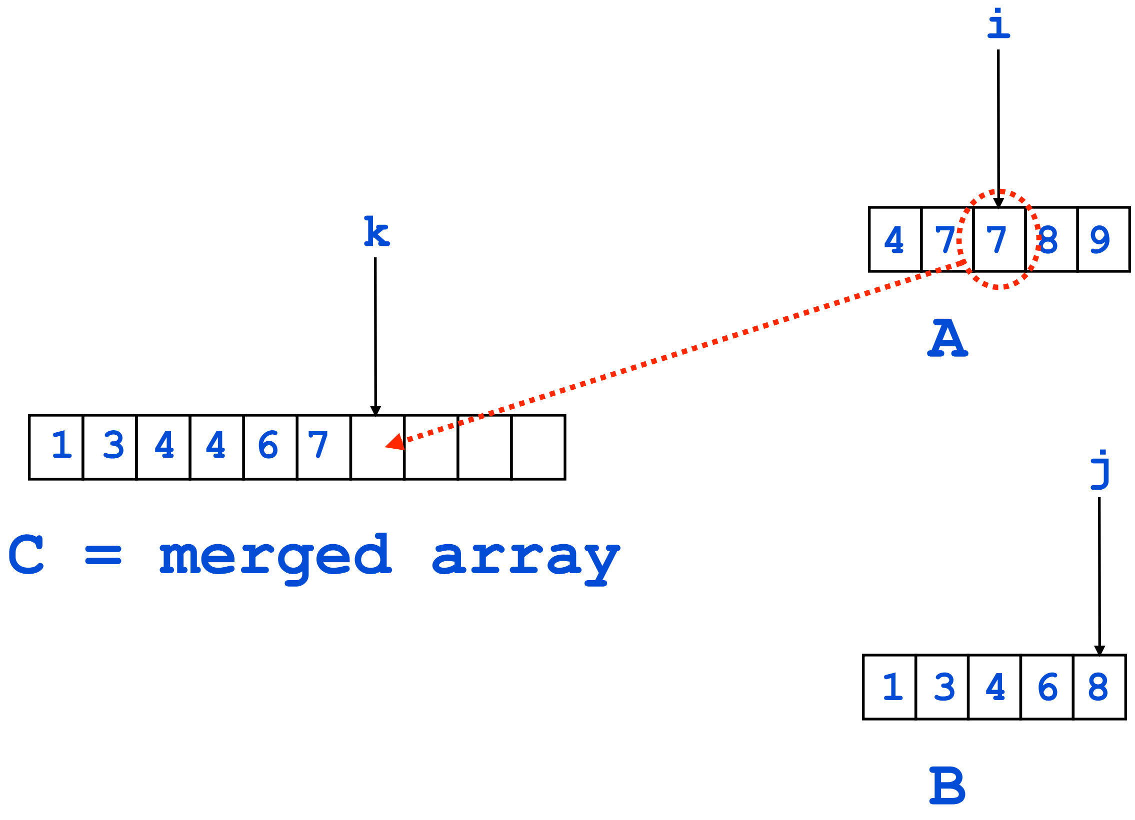
MergeSort

- Quintessential divide-and-conquer algorithm
- Divide array into equal parts, sort each part, then merge
- Questions:
  - Q1: How do we divide array into two equal parts?
  - A1: Find middle index: **`a.length/2`**
  - Q2: How do we sort the parts?
  - A2: call **`MergeSort`** recursively!
  - Q3: How do we merge the sorted subarrays?
  - A3: We have to write some (easy) code

Merging Sorted Arrays **`A`** and **`B`**

- Create an array **`C`** of size = size of **`A`** + size of **`B`**
- Keep three indices:
  - **`i`** into **`A`**
  - **`j`** into **`B`**
  - **`k`** into **`C`**
- Initialize all three indices to **`0`** (start of each array)
- Compare element **`A[i]`** with **`B[j]`**, and move the smaller element into **`C[k]`**
- Increment **`i`** or **`j`**, whichever one we took, and **`k`**
- When either **`A`** or **`B`** becomes empty, copy remaining elements from the other array (**`B`** or **`A`**, respectively) into **`C`**

Merging Sorted Arrays



`MergeSort` Analysis

- Outline (detailed code on the website)
  - Split array into two halves
  - Recursively sort each half
  - Merge the two halves
- Merge = combine two sorted arrays to make a single sorted array
  - Rule: always choose the smallest item
  - Time: $O(n)$ where n is the combined size of the two arrays
- Runtime recurrence
  - Let $T(n)$ be the time to sort an array of size n

  $T(n) = 2T(n/2) + O(n)$

  $T(1) = 1$
- Can show by induction that $T(n)$ is $O(n \log n)$
- Alternately, can see that $T(n)$ is $O(n \log n)$ by looking at tree of recursive calls

`MergeSort` Notes

- Asymptotic complexity: O(n log n)
  - Much faster than $O(n^2)$
- Disadvantage
  - Need extra storage for temporary arrays
  - In practice, this can be a disadvantage, even though **`MergeSort`** is *asymptotically optimal for sorting*
  - Can do **`MergeSort`** in place, but this is *very* tricky (and it slows down the algorithm significantly)
- Are there good sorting algorithms that do not use so much extra storage?
  - Yes: **`QuickSort`**

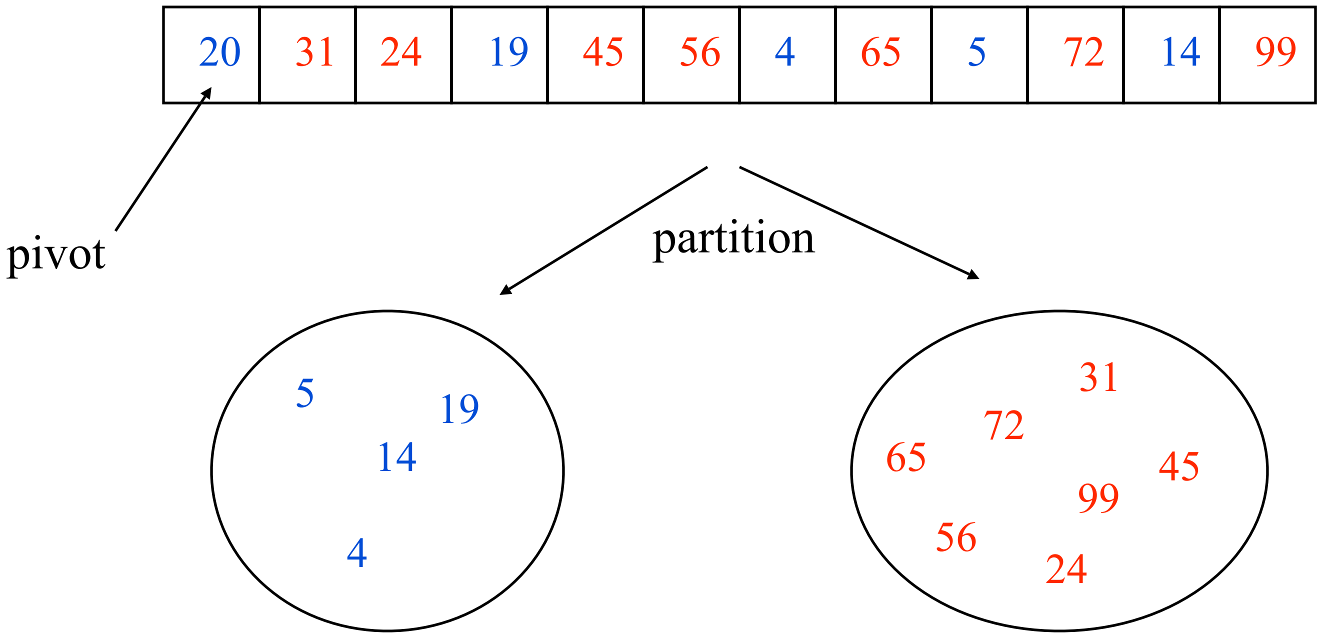
`QuickSort`

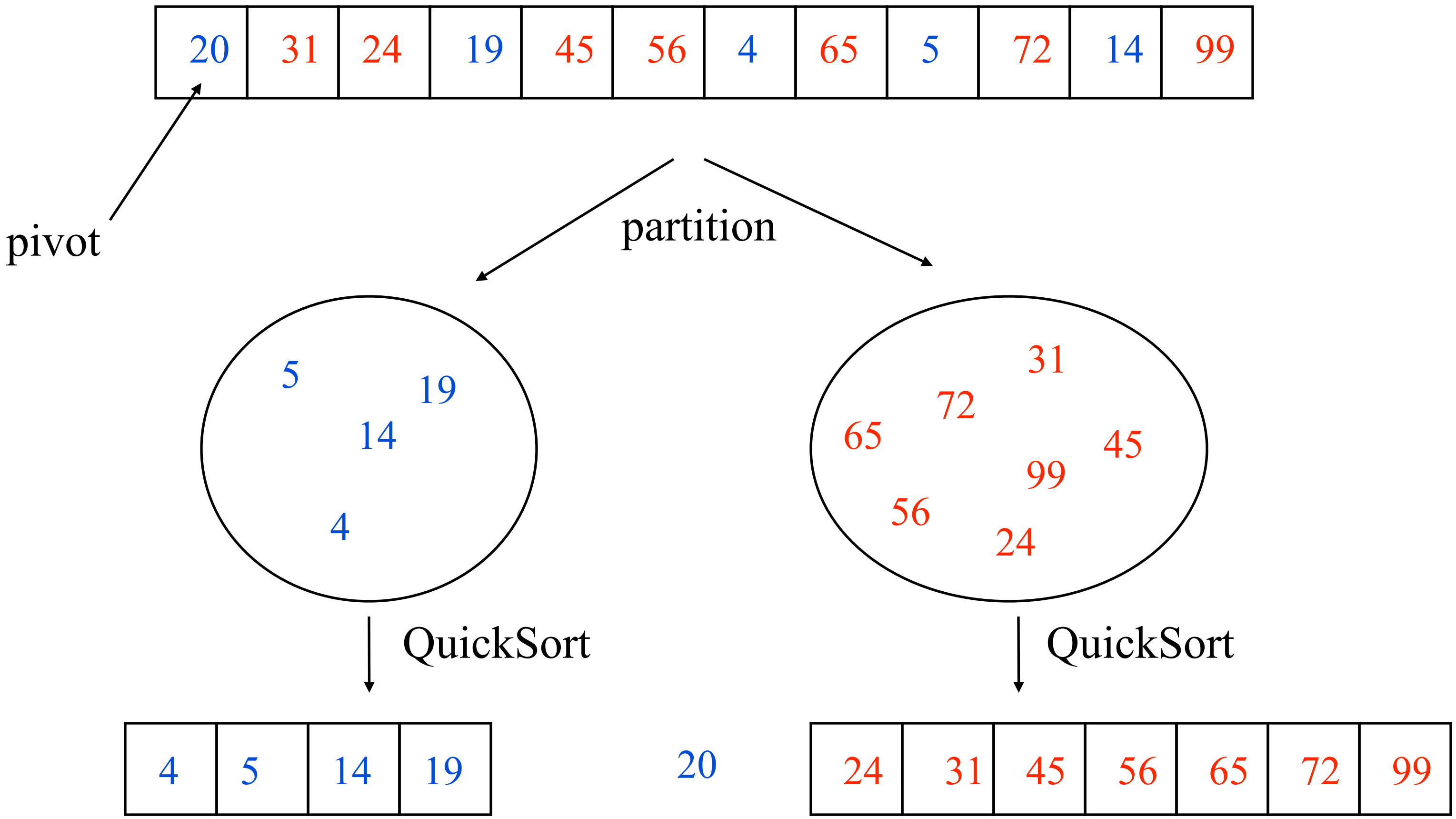
- Intuitive idea
  - Given an array **`A`** to sort, choose a pivot value **`p`**
  - Partition **`A`** into two subarrays, **`AX`** and **`AY`**
    - **`AX`** contains only elements ≤ **`p`**
    - **`AY`** contains only elements ≥ **`p`**
  - Sort subarrays **`AX`** and **`AY`** separately
  - *Concatenate* (not merge!) sorted **`AX`** and **`AY`** to get sorted **`A`**
    - Concatenation is easier than merging – O(1)

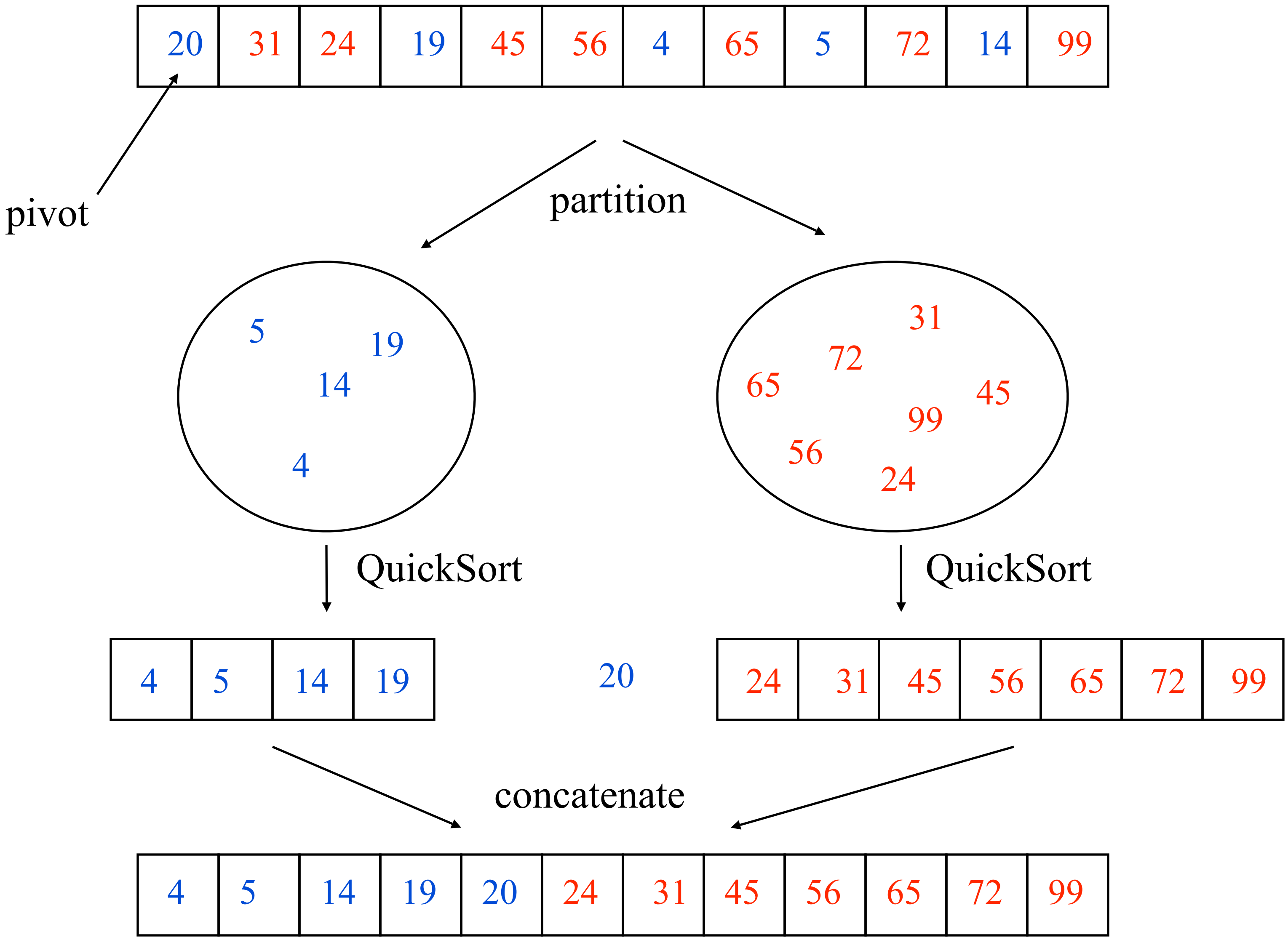
20	31	24	19	45	56	4	65	5	72	14	99

20	31	24	19	45	56	4	65	5	72	14	99

pivot


20 31 24 19 45 56 4 65 5 72 14 99
pivot
partition
5
19
14
4
31
72
65
45
99
56
24


20 31 24 19 45 56 4 65 5 72 14 99
pivot
partition
5 19 14 4
31 72 65 45 99 56 24
QuickSort
QuickSort
4 5 14 19
20
24 31 45 56 65 72 99

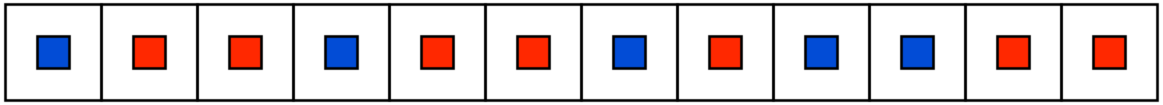


20 31 24 19 45 56 4 65 5 72 14 99
pivot
partition
5 19 14 4
31 72 65 45 99 56 24
QuickSort
QuickSort
4 5 14 19
20
24 31 45 56 65 72 99
concatenate
4 5 14 19 20 24 31 45 56 65 72 99

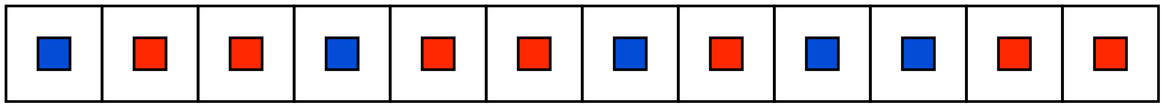
`QuickSort` Questions

- Key problems
  - How should we choose a *pivot*?
  - How do we *partition* an array in place?
- Partitioning in place
  - Can be done in O(n) time (next slide)
- Choosing a pivot
  - Ideal pivot is the median, since this splits array in half
  - Computing the median of an unsorted array is O(n), but algorithm is quite complicated
  - Popular heuristics:
    - Use first value in array (usually not a good choice)
    - Use middle value in array
    - Use median of first, last, and middle values in array
    - Choose a random element

In-Place Partitioning



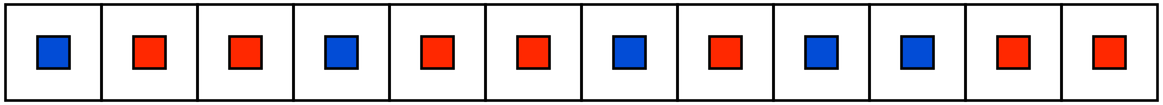
In-Place Partitioning



How can we move all the blues to the left of all the reds?

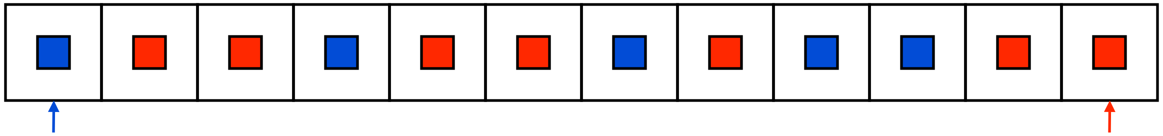
1. Keep two indices, LEFT and RIGHT
2. Initialize LEFT at start of array and RIGHT at end of array
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant

In-Place Partitioning

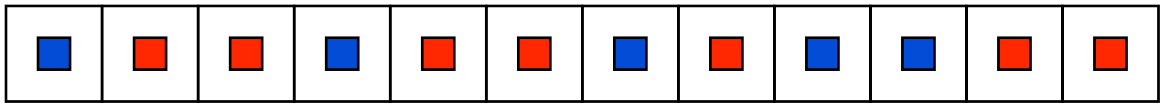


How can we move all the blues to the left of all the reds?

1. Keep two indices, LEFT and RIGHT
2. Initialize LEFT at start of array and RIGHT at end of array
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant

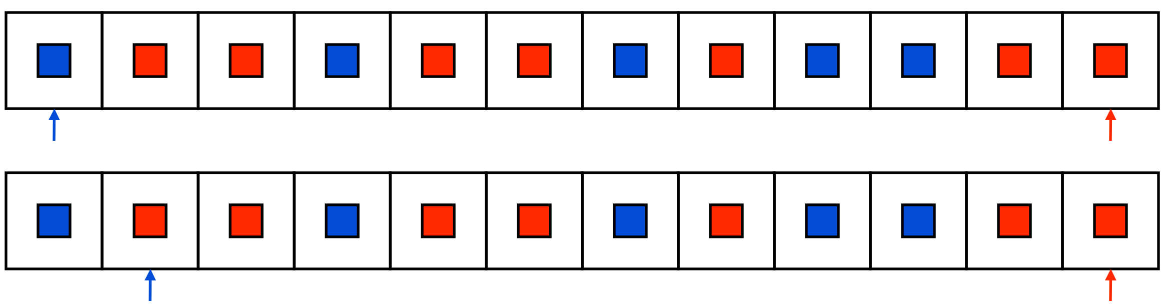


In-Place Partitioning

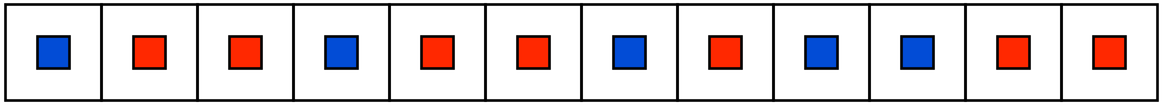


How can we move all the blues to the left of all the reds?

1. Keep two indices, LEFT and RIGHT
2. Initialize LEFT at start of array and RIGHT at end of array
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant

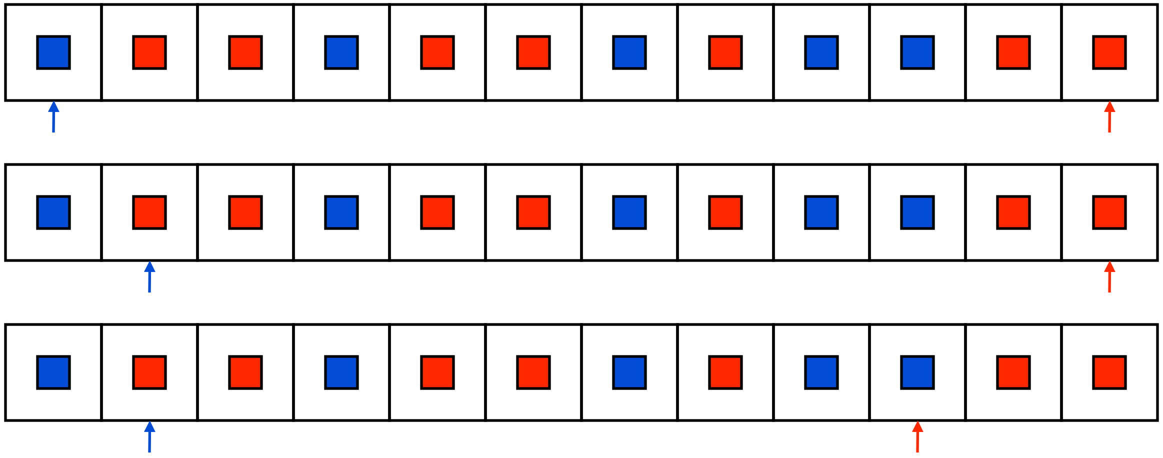


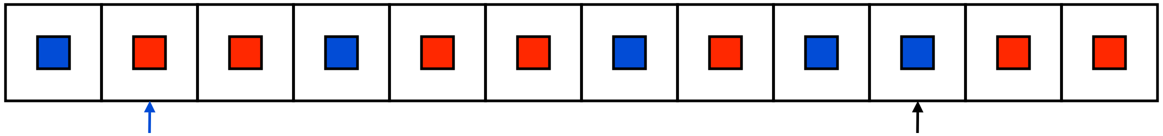
In-Place Partitioning



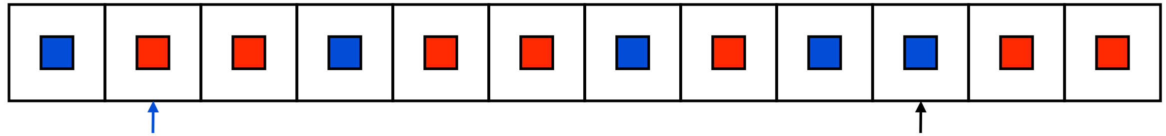
How can we move all the blues to the left of all the reds?

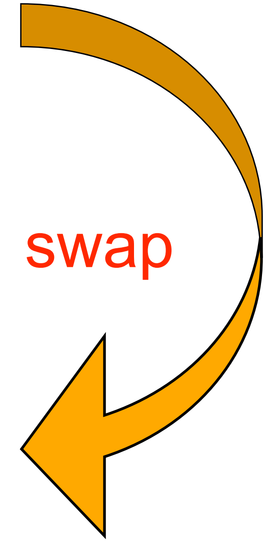
1. Keep two indices, LEFT and RIGHT
2. Initialize LEFT at start of array and RIGHT at end of array
3. Invariant: all elements to left of LEFT are blue
   all elements to right of RIGHT are red
4. Keep advancing indices until they pass, maintaining invariant



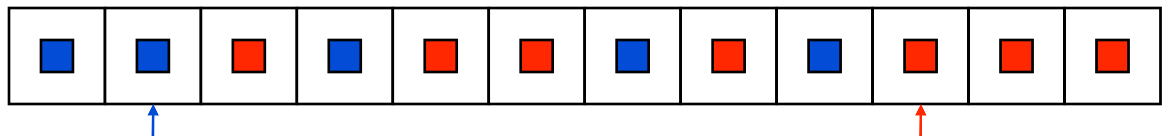


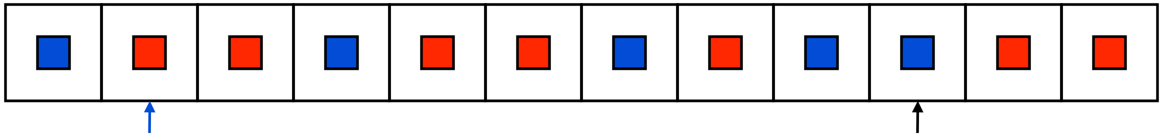
Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

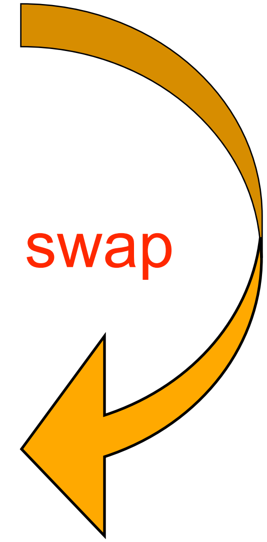




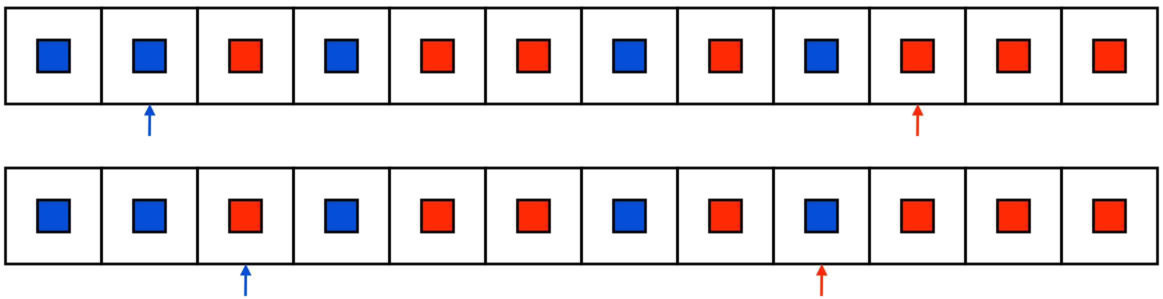
Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

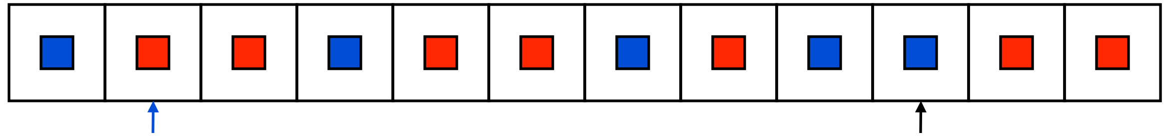


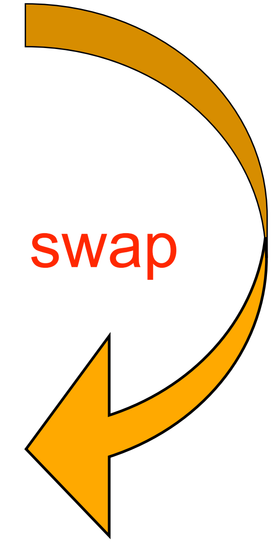




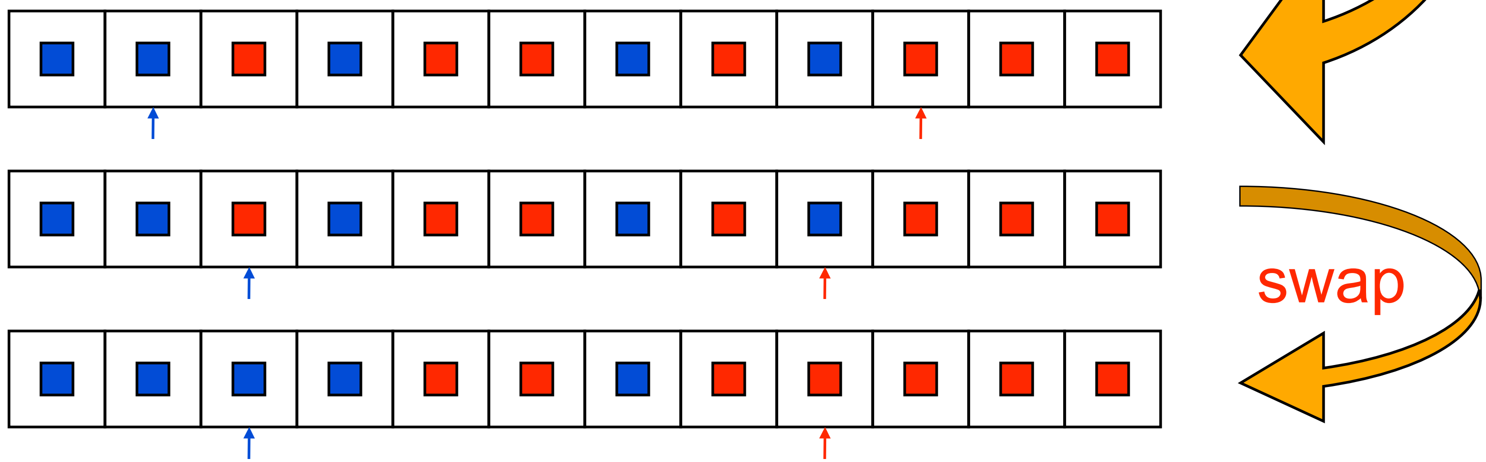
Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

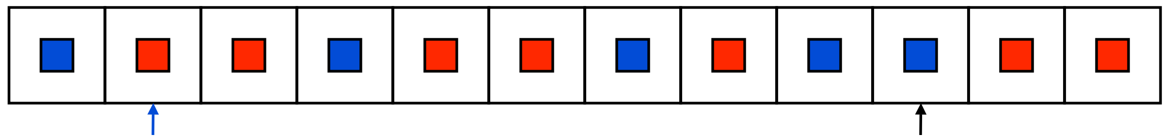




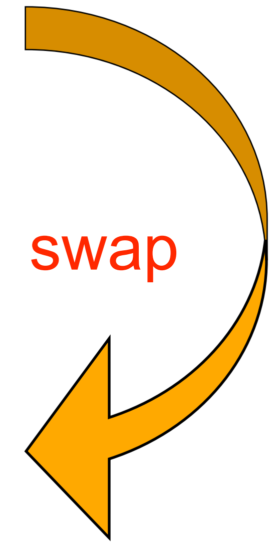


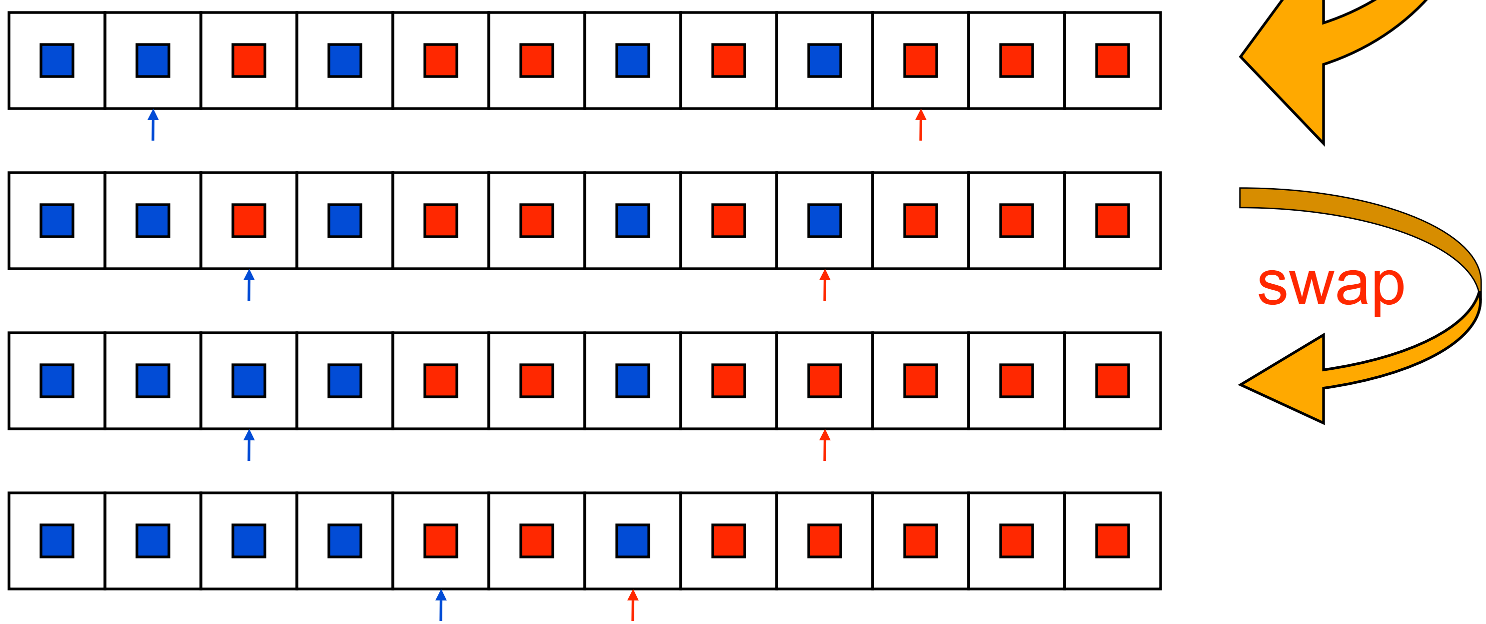
Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

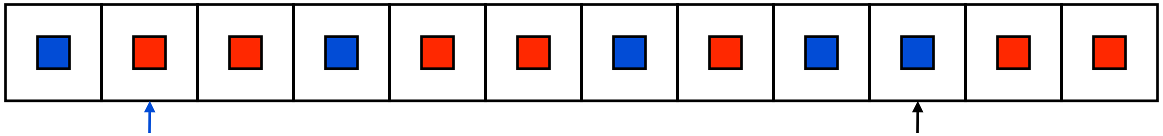




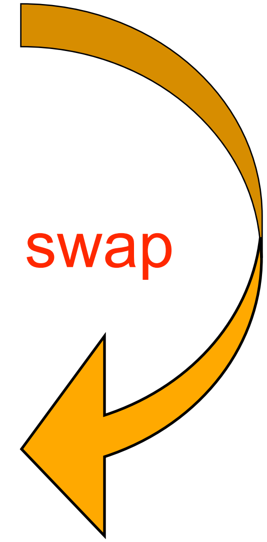
Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

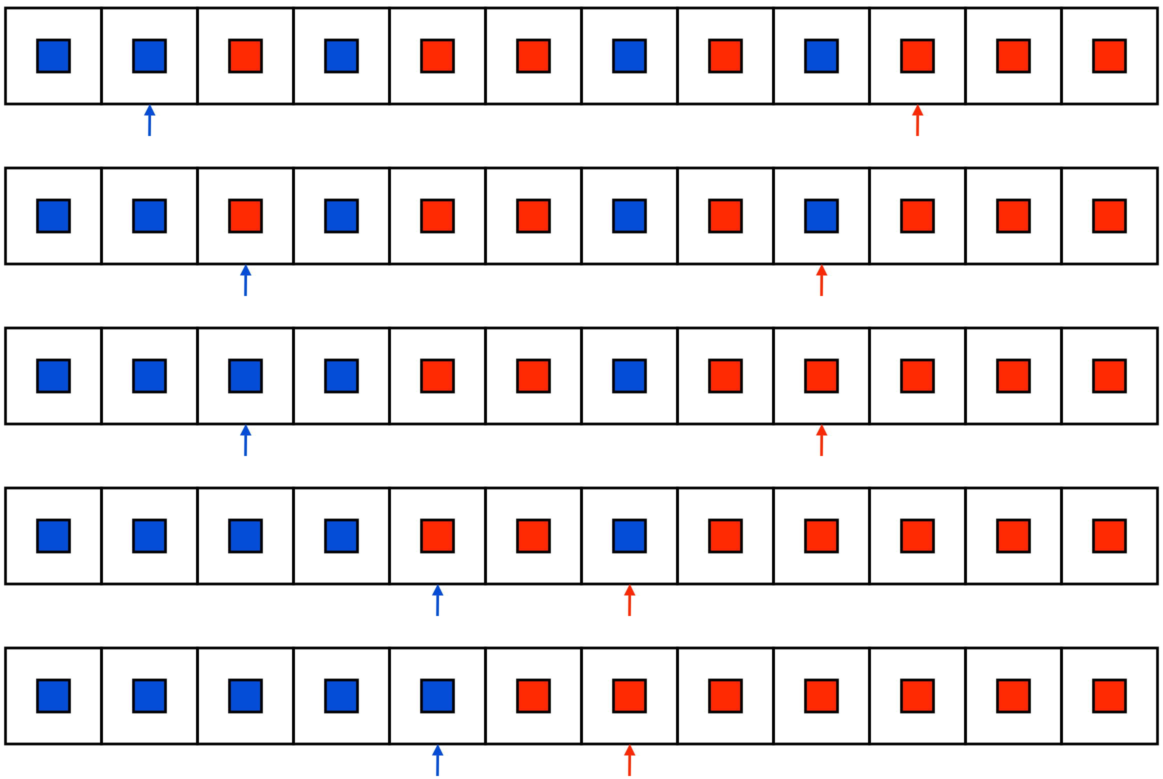


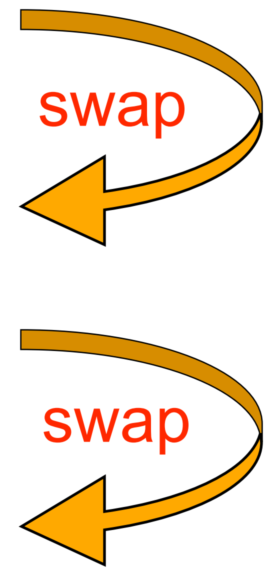


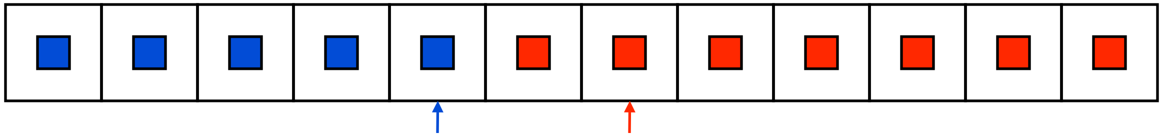


Now neither LEFT nor RIGHT can advance and maintain invariant.
We can swap red and blue pointed to by LEFT and RIGHT indices.
After swap, indices can continue to advance until next conflict.

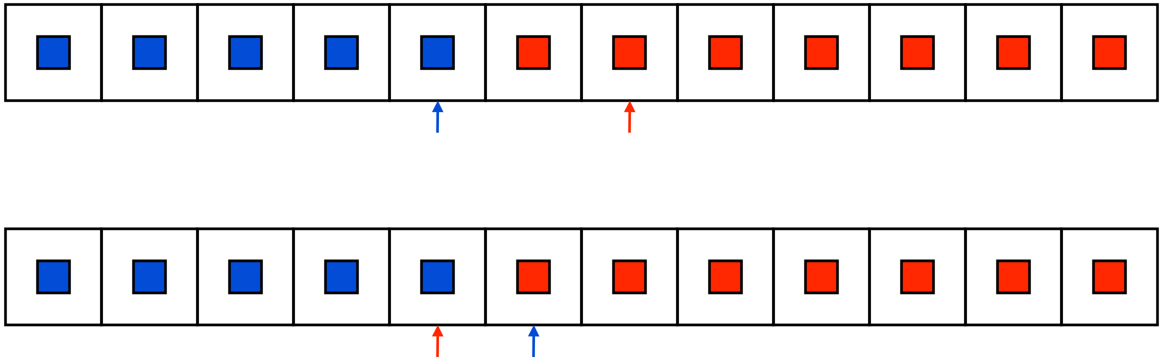








- Once indices cross, partitioning is done
- If you replace blue with ≤ **`p`** and red with ≥ **`p`**, this is exactly what we need for QuickSort partitioning
- Notice that after partitioning, array is partially sorted
- Recursive calls on partitioned subarrays will sort subarrays
- No need to copy/move arrays, since we partitioned in place



- Once indices cross, partitioning is done
- If you replace blue with ≤ `p` and red with ≥ `p`, this is exactly what we need for QuickSort partitioning
- Notice that after partitioning, array is partially sorted
- Recursive calls on partitioned subarrays will sort subarrays
- No need to copy/move arrays, since we partitioned in place

`QuickSort` Analysis

- Runtime analysis (worst-case)
  - Partition can work badly, producing this:

p	$\geq$ **p**

  - Runtime recurrence
    - T(n) = T(n–1) + n
  - This can be solved to show worst-case T(n) is $O(n^2)$
- Runtime analysis (expected-case)
  - More complex recurrence
  - Can solve to show *expected* T(n) is O(n log n)
- Improve constant factor by avoiding **`QuickSort`** on small sets
  - Switch to **`InsertionSort`** (for example) for sets of size, say, ≤ 9
  - Definition of *small* depends on language, machine, etc.

Sorting Algorithm Summary

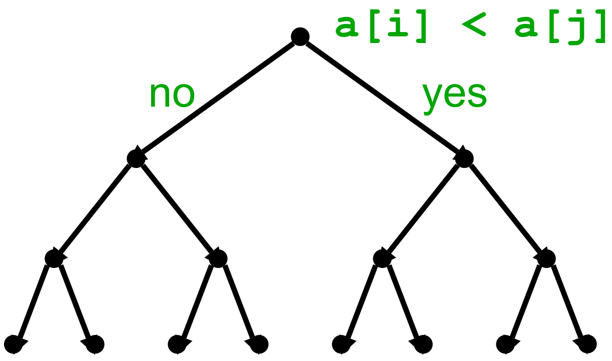
- The ones we have discussed
  - `InsertionSort`
  - `SelectionSort`
  - `MergeSort`
  - `QuickSort`
- Other sorting algorithms
  - `HeapSort` (will revisit this)
  - `ShellSort` (in text)
  - `BubbleSort` (nice name)
  - `RadixSort`
  - `BinSort`
  - `CountingSort`
- Why so many? Do computer scientists have some kind of sorting fetish or what?
  - Stable sorts: `Ins, Sel, Mer`
  - Worst-case O(n log n): `Mer, Hea`
  - Expected O(n log n): `Mer, Hea, Qui`
  - Best for nearly-sorted sets: `Ins`
  - No extra space needed: `Ins, Sel, Hea`
  - Fastest in practice: `Qui`
  - Least data movement: `Sel`

Lower Bound for Comparison Sorting

- Goal: Determine the minimum time *required* to sort *n* items
- Note: we want *worst-case,* not *best-case* time
  - Best-case doesn't tell us much; for example, we know Insertion Sort takes O(n) time on already-sorted input
  - Want to know the *worst-case time* for the *best possible* algorithm
- But how can we prove anything about the *best possible* algorithm?
  - We want to find characteristics that are common to *all* sorting algorithms
  - Let's limit attention to *comparison-based algorithms* and try to count number of comparisons

Comparison Trees

- Comparison-based algorithms make decisions based on comparison of data elements
- This gives a *comparison tree*
- If the algorithm fails to terminate for some input, then the comparison tree is infinite
- The height of the comparison tree represents the *worst-case number of comparisons* for that algorithm
- Can show that *any* correct comparison-based algorithm must make at least n log n comparisons in the worst case



Lower Bound for Comparison Sorting

- Say we have a correct comparison-based algorithm
- Suppose we want to sort the elements in an array **`B[]`**
- Assume the elements of **`B[]`** are distinct
- Any permutation of the elements is initially possible
- When done, **`B[]`** is sorted
- But the algorithm could not have taken the same path in the comparison tree on different input permutations

Lower Bound for Comparison Sorting

- How many input permutations are possible? $n! \sim 2^{n \log n}$
- For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree
- to have at least $n! \sim 2^{n \log n}$ leaves, it must have height at least n log n (since it is only binary branching, the number of nodes at most doubles at every depth)
- therefore its longest path must be of length at least n log n, and that it its worst-case running time

`java.lang.Comparable<T>` Interface

- **`public int compareTo(T x);`**
  - Returns a negative, zero, or positive value
    - negative if **`this`** is before **`x`**
    - 0 if **`this.equals(x)`**
    - positive if **`this`** is after **`x`**

- Many classes implement **`Comparable`**
  - **`String`**, **`Double`**, **`Integer`**, **`Character`**, **`Date`**,…
  - If a class implements **`Comparable`**, then its **`compareTo`** method is considered to define that class's *natural ordering*

- Comparison-based sorting methods should work with **`Comparable`** for maximum generality