

CSC 212: Data Structures and Abstractions

Quick Sort

Jonathan Schrader

[credit Marco Alvarez]

Department of Computer Science and Statistics
University of Rhode Island

Fall 2022



Quick Sort

- **Divide** the array into **two** partitions (subarrays)
 - ✓ need to pick a *pivot* and rearrange the elements into two partitions
- Conquer **Recursively** each half
 - ✓ call Quick Sort on each partition (i.e. solve 2 smaller problems)
- **Combine** Solutions
 - ✓ there is no need to combine the solutions

Quick Sort: pseudocode

```
if (hi <= lo) return;
```

```
int p = partition(A, lo, hi);
```

```
quicksort(A, lo, p-1);
```

```
quicksort(A, p+1, hi);
```

Partition

10	12	3	7	4	13	11	9
----	----	---	---	---	----	----	---

10

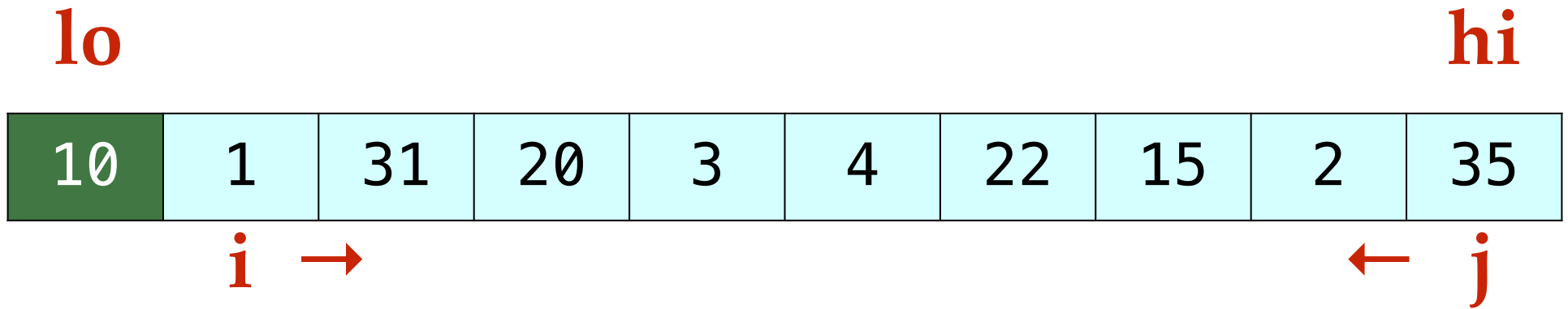
 ← pick a **pivot** (it can be the first element)

4	9	3	7	10	13	11	12
---	---	---	---	----	----	----	----

\leq pivot

\geq pivot

Partition: algorithm



```
while (true)
    scan i left-to-right (while  $a[i] < a[lo]$ )
    scan j right-to-left (while  $a[j] > a[lo]$ )
    if i and j crossed then
        break
    swap  $a[i]$  with  $a[j]$ 
swap  $a[lo]$  with  $a[j]$ 
```

Partition: do it yourself

12	1	31	20	10	11	8	2	23	1
----	---	----	----	----	----	---	---	----	---

```
while (true)
    scan i left-to-right (while a[i] <
a[lo]) scan j right-to-left (while a[j]
> a[lo]) if i and j crossed then
        break
    swap a[i] with a[j]
swap a[lo] with a[j]
```

Partition: implementation

```
int partition(int *A, int lo, int hi) {  
    int i = lo;  
    int j = hi+ 1;  
    while (1) {  
        // while A[i] < pivot, increase i  
        while (A[++i] < A[lo]) if (i == hi) break;  
        // while A[i] > pivot, decrease j  
        while (A[lo] < A[--j]) if (j == lo) break;  
        // if i and j cross exit the loop  
        if (i >= j) break;  
        // swap A[i] and A[j]  
        std::swap(A[i], A[j]);  
    }  
    // swap the pivot with A[j]  
    std::swap(A[lo], A[j]);  
    // return pivot's position  
    return j;  
}
```

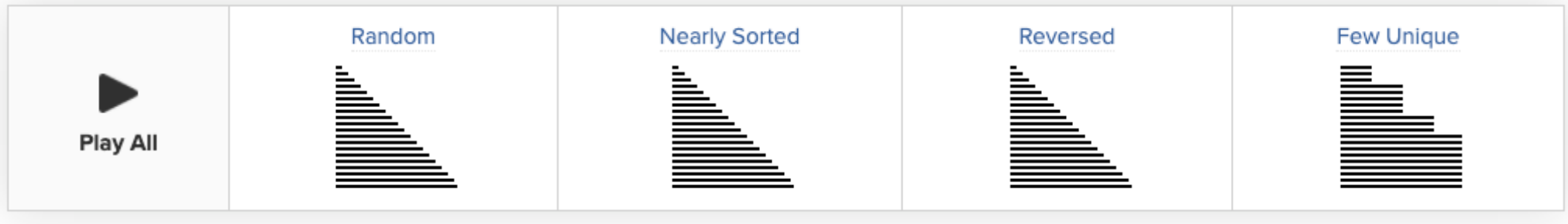
Quick Sort: implementation

```
void r_quicksort(int *A, int lo, int hi) {  
    if (hi <= lo) return;  
    int p = partition(A, lo, hi);  
    r_quicksort(A, lo, p-1);  
    r_quicksort(A, p+1, hi);  
}
```

```
void quicksort(int *A, int n, int m) {  
    // shuffle the array  
    std::random_shuffle(A, A+n);  
    // call recursive quicksort  
    r_quicksort(A, 0, n-1);  
}
```


Animation

<https://www.toptal.com/developers/sorting-algorithms/quick-sort>



Analysis of Quick Sort

- **Best-case**

- ✓ pivot partitions array evenly (almost never happens)

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\&= \dots \\&= \Theta(n \log n)\end{aligned}$$

Analysis of Quick Sort

▸ Worst-case

✓ input sorted, reverse order, equal elements

$$\begin{aligned}T(n) &= T(n - 1) + T(0) + \Theta(n) \\&= T(n - 1) + \Theta(1) + \Theta(n) \\&= T(n - 1) + \Theta(n) \\&= \dots \\&= \Theta(n^2)\end{aligned}$$

can shuffle the array (to avoid the worst-case)

Analysis of Quick Sort

▸ Average-case

- ✓ analysis is more complex (assumes distinct elements)

- ✓ Consider a 9-to-1 proportional split
- ✓ Even a 99-to-1 split yields same running time
- ✓ Faster than merge sort in practice (less data movement)

$$\begin{aligned} T(n) &= T(9n/10) + T(n/10) + \Theta(n) \\ &= \dots \\ &= \Theta(n \log n) \end{aligned}$$

Comments on Quick Sort

▸ Properties

- ✓ it is **in-place** but **not stable**
- ✓ benefits substantially from **code tuning**

▸ Improvements

- ✓ use insertion sort for small arrays
 - avoid overhead on small instances (~10 elements)
- ✓ median of 3 elements
 - estimate true median by inspecting 3 random elements
- ✓ three-way partitioning
 - create three partitions $< \text{pivot}$, $== \text{pivot}$, $> \text{pivot}$

Sorting Algorithms

	Best-Case	Average-Case	Worst-Case	Stable?	In-place?
Selection Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	No	Yes
Insertion Sort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	Yes	Yes
Merge Sort	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n \log n)$	Yes	No
Quick Sort	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n^2)$	No	Yes

Empirical Analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.