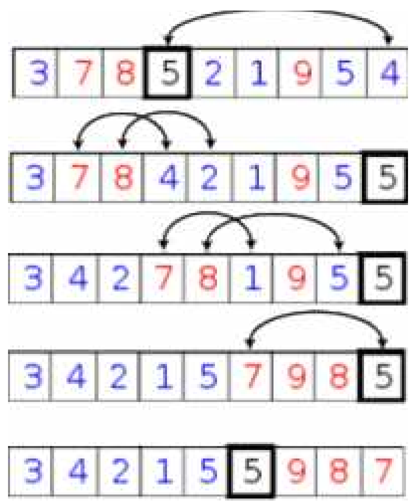
# Lecture 11
# QuickSort




EECS 281: Data Structures & Algorithms

# Two Problems with Simple Sorts

- They might compare every pair of elements
  - Learn only one piece of information/comparison
  - Contrast with binary search: learns $N/2$ pieces of information with first comparison
- They often move elements one place at a time (bubble and insertion)
  - Even if the element is “far” out of place
  - Contrast with selection sort: moves each element exactly to its final place
- Faster sorts attack these two problems

# Quicksort: Background

- ‘Easy’ to implement
- Works well with variety of input data
- $O(1)$ additional memory (plus memory for the stack frames!)

# Quicksort: Divide and Conquer

- Base case:
  - Arrays of length 0 or 1 are trivially sorted
- Inductive step:
  - Guess an element *elt* to partition array
  - Form array of [LHS]elt[RHS] (divide)
    - $\forall x \in$ LHS, x <= elt
    - $\forall y \in$ RHS, y >= elt
  - Recursively sort [LHS] and [RHS] (conquer)

# Quicksort with Simple Partition

```
void quicksort(int a[], int left, int right) {
    if (left >= right)
        return;
    int pivot = partition(a, left, right);
    quicksort(a, left, pivot - 1);
    quicksort(a, pivot + 1, right);
}
```

- If base case, return
- Else divide (partition and find pivot)
- And conquer (recursively quicksort)
- Note: pivot is not part of either recursive call. Range is inclusive [left, right]

# How to Form [LHS]elt[RHS]?

- Divide and conquer algorithm
  - Ideal division: equal-sized LHS, RHS
- <u>Ideal division is the *median*</u>
  - **How does one find the median?**
- Simple alternative: just pick any element
  - (a) array is random
  - (b) otherwise
  - Not *guaranteed* to be a good pick
  - Quality can be averaged over such choices

# Simple Partition

```
int partition(int a[], int left, int right) {
    int pivot = right--;
    while (true) {
        while(a[left] < a[pivot])
            left++;
        while(a[right] >= a[pivot]        &  left   < right)
            right--;                      &
        if (left >= right)
            break;
        swap(a[left], a[right]);
    }
    swap(a[left], a[pivot]);
    return left;
}
```

- Choose last item as pivot
- Do until left & right cross:
  - Scan from left for >= pivot
  - Scan from right for <= pivot
  - Swap them
- Move pivot to ‘middle’

# Another Partition

```
int partition(int a[], int left, int right) {
    int pivot = (left + right) / 2;     // pivot is middle
    swap(a[pivot], a[right]);           // swap with right
    pivot = right--;                    // pivot is right

    while (true) {
        while (a[left] < a[pivot])
            left++;
        while (a[right] >= a[pivot] && left < right)
            right--;
        if (left >= right)
            break;
        swap(a[left], a[right]);
    }
    swap(a[left], a[pivot]);
    return left;
}
```

- Choose middle item as pivot
- Swap it with the right end
- Repeat as before

# Analysis

- Cost of partitioning $N$ elements: $O(N)$
- Worst case: pivot always leaves one side empty
  - $T(N) = N + T(N - 1) + T(1)$
  - $T(N) = N + T(N - 1)$ [since T(1) is O(1)]
  - $T(N) \sim N^2/2 \Rightarrow O(N^2)$ [with summation trick]
- Best case: pivot divides elements equally
  - $T(N) = N + T(N / 2) + T(N / 2)$
  - $T(N) = N + 2T(N / 2) = N + 2(N / 2) + 4(N / 4) + \ldots + O(1)$
  - $T(N) \sim N \log N \Rightarrow O(N \log N)$ [master theorem or telescoping]
- Average case: tricky
  - Between $2N \log N$ and $\sim 1.39\, N \log N \Rightarrow O(N \log N)$

# Quicksort: Pros and Cons

## Advantages

- On average, $n \log n$ time to sort $n$ items
- Short inner loop $O(n)$
- Efficient memory usage
- Thoroughly analyzed and understood

## Disadvantages

- Worst case, $n^2$ time to sort $n$ items
- Not stable, and incredibly difficult to make stable
- Fragile (simple implementation mistakes very hard to fix)

# Improving Splits

- Key to performance: a “good” split
  - Any single choice could always be worst one
  - Too expensive to actually compute best one (median)
- Rather than compute median, sample it
  - Simple way: pick three elements, take their medians
  - Very likely to give you better performance
- Sampling is a *very* powerful technique!

# Other Improvements

- Divide and conquer: most sorts are little
- Reduce the cost of “little” sorts
  - Insertion sort is faster than quicksort on small arrays
  - Bail out of quicksort when size < *k*
  - Either insertion-sort each small array or use a single (fast!) insertion pass at the end
- What if many elements are equal?

# Summary: Quicksort

- On average, $O(n \log n)$-time sort
- Efficiency based upon selection of pivot
  - randomly choose middle or last key in partition
  - sample three keys
  - other creative methods
- Other methods of tuning
  - use another sort when partition is ‘small’
  - three-way partition

# Sorting Algorithms: Performance

**Large-scale classification:** [use big-*O* analysis]

Sorting algorithms that use worst-case $O(n^2)$ time

- Bubble sort
- Insertion sort
- Selection sort

elementary sorts

- Heapsort

heap-based sort, O(n log n) worst-case

- Quicksort

divide-and-conquer

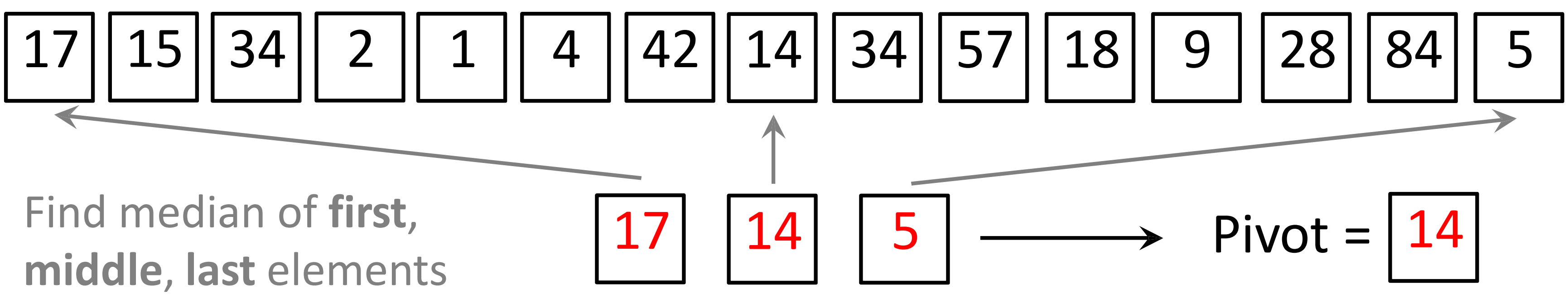
Common case: $O(n \log n)$ depending on pivot selection

# Sorting Algorithms: Performance

**<u>Smaller-scale classification</u>:** [implementation-specific]

<u>Example: Quicksort and pivot selection</u>

**Method 1: Sampling (Best of 3, Best of 5, etc.)**




<u>Runtime</u>: $O(1)$

# Sorting Algorithms: Performance

<u>Quicksort pivot selection</u>

**Method 1: Sampling (Best of 3, Best of 5, etc.)**

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | 15 | 34 | 2 | 1 | 4 | 42 | 14 | 34 | 57 | 18 | 9 | 28 | 84 | 5 |

Find median of 3 **random** elements

<u>Runtime</u>: $O(1)$

1 42 28 → Pivot = 28

# Sorting Algorithms: Efficiency

**Memory usage – how much extra room do you need?**

- Bubble sort
- Insertion sort
- Selection sort

$\left.\begin{array}{l}\text{Bubble sort}\\ \text{Insertion sort}\\ \text{Selection sort}\end{array}\right\}$ $O(1)$ overhead – in-place sorting

- Heapsort?
- Quicksort?

# Sorting Algorithms: Stability

**A sorting algorithm is <u>stable</u> if data with the same key remains in the same relative locations**

**Example Input:** (already sorted by first name)
{"Sheen, Charlie", "Berry, Halle", "Liu, Lucy", "Sheen, Martin"}

Sort by **Last Name**

**Example Stable Output:** (two "Sheen" stay in relative locations)
{"Berry, Halle", "Liu, Lucy", "Sheen, Charlie", "Sheen, Martin"}

**Example Unstable Output:**
{"Berry, Halle", "Liu, Lucy", "Sheen, Martin", "Sheen, Charlie"}

# Sorting Algorithms: Stability

**A sorting algorithm is <u>stable</u> if the output data having the same key value remains in the same relative locations**

- Is bubble sort stable?
- Is insertion sort stable?
- Is selection sort stable?
- Is heapsort stable?
- Is quicksort stable?

<u>How can you use an unstable sorting algorithm but ensure stable output?</u>

# Questions for Self-study

- Illustrate worst case input for quicksort
- Explain why best-case runtime for quicksort is not linear
  - Give two ways to make it linear
    (why is this not done in practice ?)
- Normally, pivot selection takes $O(1)$ time, what will happen to quicksort's complexity if pivot selection takes $O(n)$ time?
- Improve quicksort with $O(n)$-time median selection
  - Must limit median selection to linear time in all cases