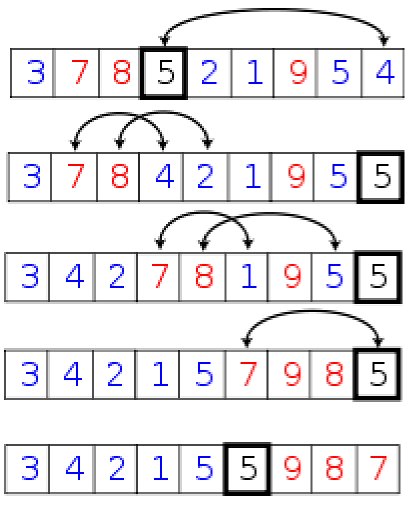
Lecture 11
Quicksort



EECS 281: Data Structures & Algorithms

Quicksort Basics

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
- In-place sort (no extra memory for data)
- Additional memory for 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 the 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(Item a[], size_t left, size_t right) {
  if (left + 1 >= right)
    return;
  size_t pivot = partition(a, left, right);
  quicksort(a, left, pivot);
  quicksort(a, pivot + 1, right);
}  // quicksort()
```

- Range is [left, right)
- If base case, return
- Else divide (partition and find pivot)
  and conquer (recursively quicksort)

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

```
size_t partition(Item a[], size_t left, size_t right) {
  size_t pivot = --right;
  while (true) {
    while (a[left] < a[pivot])
      ++left;
    while (left < right && a[right - 1] >= a[pivot])
      --right;
    if (left >= right)
      break;
    swap(a[left], a[right - 1]);
  }  // while
  swap(a[left], a[pivot]);
  return left;
}  // partition()
```

- Choose last item as pivot
- Scan…
  - from left for >= pivot
  - from right for < pivot
- Swap left & right pairs and continue scan until left & right cross
- Move pivot to ‘middle’

Example: pick-the-last

{2 9 3 4 7 5 8 6}

Another Partition

```
size_t partition(Item a[], size_t left, size_t right) {
  size_t pivot = left + (right - left) / 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 (left < right && a[right - 1] >= a[pivot])
      --right;
    if (left >= right)
      break;
    swap(a[left], a[right - 1]);
  }  // while
  swap(a[left], a[pivot]);
  return left;
}  // partition()
```

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

Example: worst-case (min/max)

{1 2 3 4 5}

Quicksort Basics

Data Structures & Algorithms

Quicksort Analysis & Performance

Data Structures & Algorithms

Time Analysis

- Cost of partitioning n elements: $O(n)$
- Worst case: pivot always leaves one side empty
  - $T(n) = n + T(n - 1) + T(0)$
  - $T(n) = n + T(n - 1) + c$ [since $T(0)$ is $O(1)$]
  - $T(n) \sim n^2/2 \Rightarrow O(n^2$ [via substitution]
- 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 substitution]
- Average case: tricky
  - Between $2n \log n$ and $\sim 1.39\, n \log n \Rightarrow O(n \log n)$

Memory Analysis

- Requires stack space for recursive calls
- The first recursive call is NOT tail recursive, requires a new stack frame
- The second recursive call IS tail recursive, which reuses the current stack frame
- When pivoting is going terribly:
  - $O(n)$ stack frames if split is $(n - 1)$, pivot, (0)
  - $O(1)$ stack frames if split is (0), pivot, $(n - 1)$

Sort Smaller Region First

```
void quicksort(Item a[], size_t left, size_t right) {
  if (left + 1 >= right)
    return;
  size_t pivot = partition(a, left, right);
  if (pivot - left < right - pivot) {
    quicksort(a, left, pivot);
    quicksort(a, pivot + 1, right);
  } else {
    quicksort(a, pivot + 1, right);
    quicksort(a, left, pivot);
  }  // else
}  // quicksort()
```

- Worst memory requirement?
- Both sides equal: $O(\log n)$

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

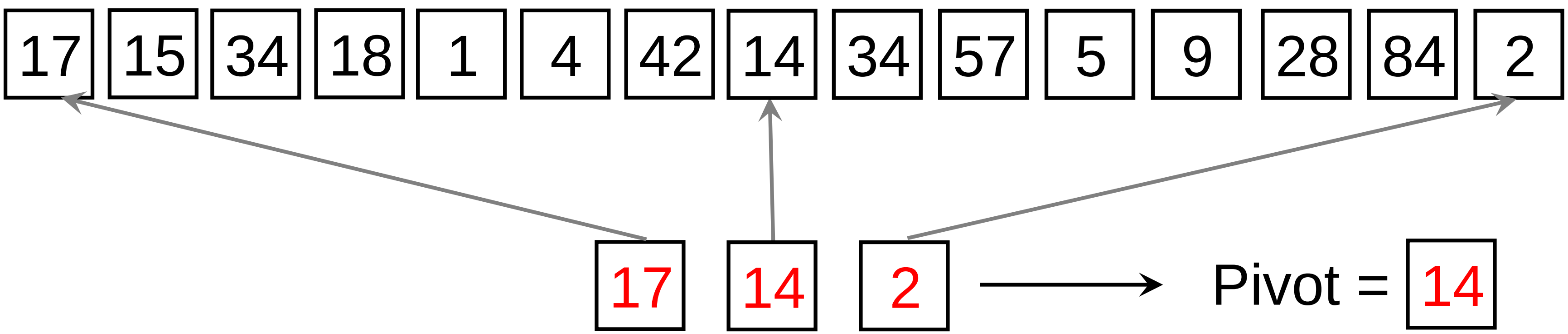
- Worst case, n^2 time to sort n items
- Not stable; making it stable sacrifices time and/or memory
- Partitioning is fragile (simple mistakes will either segfault or not sort properly)

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!

Median Sampling: Fixed

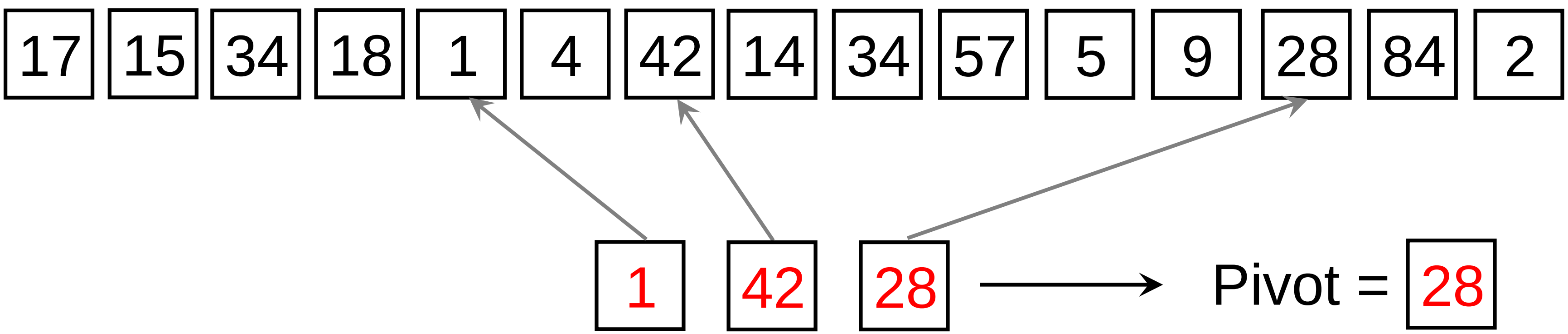
Find median of first, middle, and last elements



Runtime: ***O*(1)**

Median Sampling: Random

Find median of three (five, seven,…) random elements



Runtime: ***O*(1)**

Other Improvements

- Divide and conquer
  - – Most sorts are “small” regions
  - – Lots or recursive calls to small regions
- Reduce the cost of sorting small regions
  - – Insertion sort is faster than quicksort on small arrays
  - – Bail out of quicksort when size < *k*
    - For some small, fixed *k*, usually around 16 or 32
  - – Insertion sort each small sub-array

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

Quicksort Analysis & Performance

Data Structures & Algorithms

Sorting Summary

Data Structures & Algorithms

Sorting Algorithms: Time

- Bubble sort
- Insertion sort
- Selection sort

elementary sorts (worst-case $O(n^2)$)

- Heapsort

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

- Quicksort

divide-and-conquer

Average case: $\boldsymbol{O(n \log n)}$ depending on pivot selection

Sorting Algorithms: Memory

- Bubble sort
- Insertion sort
- Selection sort
- Heapsort
- Quicksort?

In-place sorts - $O(1)$ extra memory

Sorting Algorithms: Stability

A sorting algorithm is stable if output data having the **same key values** remain in the **same relative locations** after the sort

- Bubble sort ✓
- Insertion sort ✓
- Selection sort ✗
- Heapsort ✗
- Quicksort ✗

Introsort

- Introspective Sort
  - Introspection means to think about oneself
- Used by g++ and Visual Studio

```
Algorithm introsort(a[], n):
  if (n is small)
    insertionSort()
  else if (quicksort.recursionDepth is large)
    heapsort()
  else
    quicksort()
```

Sorting Summary

Data Structures & Algorithms

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