CSE 373: Sorting
(QuickSort)

Chapter 7

QuickSort

QuickSort:
- Another recursive divide-and-conquer sorting algorithm
- In practice, the fastest known sorting algorithm
Partitioning

Partitioning: Quicksort’s main operation
- given a list...
- choose a pivot element, \( p \), from the list
- divide the rest of the values into two sets:
  - those less than \( p \)
  - those greater than \( p \)
  - (for now, we’ll ignore those that are equal to \( p \))

Partitioning Example

(Assume we’ll use the first element as a pivot):

\[
\begin{array}{ccccccc}
7 & 4 & 8 & 6 & 9 & 2 & 5 & 3 \\
\end{array}
\]

Running time of partition?
QuickSort Overview

QuickSort: given a list of values...
- if the list contains zero or one elements, return it
- otherwise, partition the list
- call QuickSort recursively on each half
- concatenate the results of the recursive calls:
  QuickSort(small values) :: pivot :: QuickSort(big values)

QuickSort Example

input = 7, 4, 8, 6, 9, 2, 5, 3 (using first element as pivot)
**QuickSort Call Tree**

```
QuickSort (7, 4, 8, 6, 9, 2, 5, 3)
  /       \
QuickSort (4, 6, 2, 5, 3)  QuickSort (8, 9)
  |       |
QS (2, 3)  QS (6, 5)  QS ()  QS (9)
  |       |
QS ()  QS (3)  QS (5)  QS ()
```

**Running Time** *(Approximate & Optimistic)*

Assuming all pivots result in even partitions...

- ~linear work per step
- ~logn steps

```
= \(O(n) \times O(\log n) = O(n \log n)\)
```
Worst-Case Analysis

• What would be a worst-case partition step?

• What input would cause this worst case at every step (assuming pivot is first element)?

• What’s the running time of this worst-case?

Design Decision: Choosing Pivot

• first element – should never, never be used

• random element

• median

• median of three (first, middle, last?)

• middle element
In-Place Partitioning

1) swap the pivot \( p \) with the last element
2) set a pointer \( i \) to the first element
3) set a second pointer \( j \) to the second-to-last
4) walk \( i \) up the array until a value \( > p \) is found
5) walk \( j \) down the array to a value \( < p \)
6) swap elements pointed to by \( i \) and \( j \)
7) continue until \( i \) and \( j \) pass one another
8) when they do, swap \( i \)'s element with \( p \)

In-Place Partitioning Example

input = 7, 4, 8, 6, 9, 2, 5, 3 (using median-of-three pivot)
Quicksort Best-Case Analysis

Use a recurrence relation:

T(0) = k
T(1) = k
T(n) = 2T(n/2) + c \cdot n

Solve using repeated substitution:

Quicksort Overview

- **Running Times:**
  - Best Case: \(O(n \log n)\)
  - Worst Case: \(O(n^2)\) – *but very unlikely*
  - Average Case: \(O(n \log n)\) – shown in book

- **Space Requirement:** sorts in-place
**Design Details**

- Sort small arrays \((n < 20?)\) using insertion sort
  - insertion sort faster for small problems
  - all Quicksorts on big lists must also sort small lists

- **How to handle elements equal to pivot?**
  - annoying detail; see book

- **Quickselect** – a modification of Quicksort to do selection in \(O(n)\) time (on average)

---

**Bucket sort**

Useful for sorting integers of a fixed range:

- Declare an array: `int count[range]`
- Initialize `count[\cdot]\)` to all 0’s
- Iterate over the input list
- For each value \(v\), increment `count[v]`
- Once done, print out `count[0]'s, `count[1]'s, ...

**Running time?**