Quick Sort

CS 351: Analysis of Algorithms

Lucas P. Cordova, Ph.D.

This lecture covers the QuickSort algorithm, including its design, implementation, and analysis. We will explore the partitioning concept, pivot selection strategies, and the average-case performance of QuickSort. Additionally, we will discuss the theoretical foundations of comparison-based sorting algorithms and their optimality.

Table of contents

0.0.1 Executive Summary

> What We'll Discuss This Week
>
> - QuickSort algorithm design and implementation
> - Partitioning concept and in-place implementation
> - Pivot selection strategies and their impact
> - Running time analysis: worst, best, and average cases
> - Mathematical analysis using probability and expectation

1 Introduction to QuickSort

1.1 Why Another Sorting Algorithm?

We already have MergeSort - why do we need QuickSort?

Professional Opinion: QuickSort appears on most computer scientists' "top 10 algorithms" lists

Practical Advantages:

- Competitive with and often superior to MergeSort
- Default sorting method in many programming libraries
- **Runs in place** - minimal memory overhead
- Operates through element swaps only

Aesthetic Appeal:

- Remarkably beautiful algorithm design
- Equally beautiful running time analysis

1.2 The Sorting Problem Revisited

Problem Definition:

- **Input**: Array of n numbers in arbitrary order
- **Output**: Same numbers sorted smallest to largest

Example:

```
Input:  [3, 8, 2, 5, 1, 4, 7, 6]
Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

Assumption: Distinct elements (no duplicates) for simplicity

Checkpoint 1: Can you think of any advantages an in-place sorting algorithm might have over one that requires additional memory?

2 The Partitioning Concept

2.1 Partitioning Around a Pivot

QuickSort is built around **partial sorting** - partitioning an array around a "pivot element"

Two-Step Process:

Step 1: Choose a pivot element

- Select one element to act as pivot
- For now, let's use the first element

Step 2: Rearrange around pivot

- Everything before pivot $<$ pivot
- Everything after pivot $>$ pivot

2.2 Partitioning Example

Input Array:

```
[3, 8, 2, 5, 1, 4, 7, 6]
```

Choose pivot = 3 (first element)

After partitioning:

```
[2, 1, 3, 6, 7, 4, 5, 8]
       ↑
     pivot
```

Key Insight: Elements before pivot don't need to be in sorted order relative to each other, and neither do elements after pivot!

2.3 Two Key Facts About Partitioning

1. Fast Implementation

- Runs in linear O(n) time
- Can be implemented **in place**
- Minimal memory beyond input array

2. Significant Progress

- Pivot ends up in its **rightful position**
- Reduces sorting to two smaller subproblems:
  - Sort elements $<$ pivot
  - Sort elements $>$ pivot

Checkpoint 2: In the partitioned array `[2, 1, 3, 6, 7, 4, 5, 8]`, what can you say about the final position of the element `3` in the fully sorted array?

3 QuickSort Algorithm

3.1 High-Level Description

```
QuickSort(array A, length n):
    if n   1:
        return  // base case - already sorted

    choose a pivot element p
    partition A around p
    recursively sort first part of A
    recursively sort second part of A
```

Visual representation:

```
[< p] [p] [> p]
  ↓    ↓    ↓
sort  done sort
```

3.2 QuickSort vs MergeSort

Order of Operations Difference:

MergeSort:

1. Recursive calls first
2. Combine step (Merge) after

QuickSort:

1. Partition step first

2. Recursive calls after
3. **No combine step needed!**

Historical Note: QuickSort was invented by Tony Hoare in 1959 when he was just 25 years old. He later won the Turing Award in 1980.

3.3 Implementation To-Do List

Three Main Questions:

1. **Section 5.2**: How do we implement the partitioning subroutine?
2. **Section 5.3**: How should we choose the pivot element?
3. **Sections 5.4-5.5**: What's the running time of QuickSort?

Checkpoint 3: Before we dive into implementation details, predict: what do you think would be the worst way to choose a pivot element?

4 Partition Implementation

4.1 The Easy Way Out

Simple approach using extra memory:

1. Scan input array A once
2. Copy elements to new array B:
    - Elements $<$ pivot go to front of B
    - Elements $>$ pivot go to back of B
    - Place pivot in remaining position

Problem: Uses O(n) extra memory **Goal**: Implement in-place with O(1) extra memory

4.2 In-Place Partitioning Strategy

Key Challenge: Rearrange elements without extra array

Solution Approach:

- Use two pointers moving towards each other
- Maintain invariant: elements in correct relative positions
- Swap elements when they're in wrong regions

Memory Usage: Only constant extra space for:

- Pointer variables
- Temporary storage for swaps

Checkpoint 4: Can you sketch how you might use two pointers (one from left, one from right) to partition an array in place?

5 Pivot Selection Strategies

5.1 The Critical Choice

Algorithm Correctness: Independent of pivot choice **Algorithm Performance**: Heavily dependent on pivot choice

Common Strategies:

1. **First element** (naive approach)
2. **Last element**
3. **Random element**
4. **Median-of-three**

5.2 Worst-Case Scenario

When things go wrong:

- Always pick minimum or maximum element as pivot
- One partition becomes empty
- Other partition has n-1 elements
- **Result**: $O(n^2)$ running time

Example: Sorted array [1,2,3,4,5,6,7,8] with first-element pivot

```
Pick 1: [1] [2,3,4,5,6,7,8]  (unbalanced!)
Pick 2: [2] [3,4,5,6,7,8]    (still unbalanced!)
...
```

5.3 Best-Case Scenario

When things go perfectly:

- Always pick median element as pivot
- Partitions are perfectly balanced
- **Result**: O(n log n) running time

Example: Each partition splits roughly in half

```
Pick median: [smaller half] [median] [larger half]
Recursively: both halves  n/2 size
Depth: log n levels
Work per level: O(n)
Total: O(n log n)
```

Checkpoint 5: If we have an 8-element array and always achieve perfect splits, how many levels of recursion will we have?

5.4 Randomized Pivot Selection

The Magic Solution: Choose pivot uniformly at random

```
1 ChoosePivot(array A, left l, right r):
2     return random element from {l, l+1, ..., r}
```

Why randomization works:

- 50% chance of getting 25%-75% split or better
- Bad splits become unlikely
- **Average performance**: O(n log n)

Real-world impact: Randomized QuickSort is a "for-free primitive"

6 Running Time Analysis

6.1 The Three Scenarios

Worst Case: $O(n^2)$

- Occurs when pivot is always min/max
- Very unlikely with random pivots

Best Case: O(n log n)

- Occurs when pivot is always median
- Optimal for comparison-based sorting

Average Case: O(n log n)

- **Most important**: Expected performance with random pivots
- Only constant factor worse than best case

6.2 Intuitive Analysis

Why randomized QuickSort works well:

Good Split Probability:

- 50% chance of 25%-75% split or better
- Even 25%-75% split makes good progress

Progress Accumulation:

- Good splits happen frequently
- Bad splits are rare and don't dominate
- Overall: logarithmic depth with linear work per level

Remarkable Fact: No matter what your input array is, if you run randomized QuickSort repeatedly, the average running time will be O(n log n)

Checkpoint 6: If we always get a 25%-75% split, what's the maximum depth of recursion for an array of size n?

7 Mathematical Analysis

7.1 Why Should We Care About Counting Comparisons?

Real-world impact: Understanding QuickSort's performance helps us:

- Choose the right sorting algorithm for our data
- Predict how long our program will take
- Optimize our code for better performance

The practical question: "If I have 1 million records to sort, how long will QuickSort take?"

What makes QuickSort slow or fast?

- Comparisons are the expensive operations
- Every time we ask "Is A[i] $<$ pivot?", that costs time
- Other operations (swaps, array access) are much cheaper

Bottom line: Count comparisons $\rightarrow$ predict performance

7.2 A Simple Way to Think About It

Imagine tracking every pair of elements:

Think of your array like a tournament bracket. Every element might get "matched up" against every other element at some point.

Example with array [4, 1, 3, 2]:

- Will 4 and 1 get compared? Maybe!
- Will 1 and 3 get compared? Maybe!

- Will 3 and 2 get compared? Maybe!

Our strategy: Instead of asking "How many total comparisons?", ask "Will these two specific elements get compared?"

Why this is smart:

- One complex question becomes many simple yes/no questions
- We can answer each yes/no question separately
- Add up all the answers to get the total

Checkpoint 7a: In a dating app with 100 users, how many possible "matches" (pairs) could there be? This is similar to counting element pairs in QuickSort.

7.3 When Do Two Elements Actually Meet?

The key insight: Two elements get compared only under specific circumstances.

Let's use a concrete example:

Array after sorting would be: [1, 2, 3, 4, 5, 6, 7, 8]

Question: When do elements 2 and 7 get compared?

Scenario 1: Element 2 becomes a pivot

- All elements get compared to the pivot
- So 2 and 7 definitely get compared

Scenario 2: Element 7 becomes a pivot

- Again, all elements compared to pivot
- So 2 and 7 get compared

Scenario 3: Element 4 becomes a pivot first

- 2 goes to the "less than 4" side
- 7 goes to the "greater than 4" side
- They're now in different subarrays - never to meet again!

7.4 The Separation Principle

Real-world analogy: Think of QuickSort like organizing a library

The library scenario:

- You're organizing books by call number
- You pick a "separator book" (the pivot)
- All books with smaller numbers go left
- All books with larger numbers go right
- Books on different sides never get compared again!

Back to our example with elements 2 and 7:

- If any element between them (3, 4, 5, or 6) becomes pivot first
- Elements 2 and 7 get permanently separated
- They'll never be compared

The general rule: Elements get compared only if one becomes a pivot before any "separator" elements are chosen.

Checkpoint 7b: You're organizing a bookshelf with books numbered 10, 20, 30, 40, 50. If book 30 is chosen as the separator first, will books 10 and 50 ever be compared later?

7.5 Calculating the Odds

Let's make this practical: What are the chances elements 2 and 7 get compared?

The contestants: Who could be chosen as pivot first?

- Elements 2, 3, 4, 5, 6, 7 (that's 6 elements total)
- Each has an equal chance of being picked (random selection)

Winning scenarios: Element 2 or 7 picked first

- That's 2 out of 6 possibilities
- Probability = 2/6 = 1/3 33%

Losing scenarios: Element 3, 4, 5, or 6 picked first

- That's 4 out of 6 possibilities
- These separate elements 2 and 7 forever

Generic formula: For elements i steps apart:

- Probability they get compared = 2/(i+1)
- Closer elements → higher chance of comparison
- Distant elements → lower chance of comparison

7.6 Why This Matters in Practice

Performance prediction: Now we can estimate QuickSort's behavior

For any array size n:

- Count all possible element pairs: roughly $n^2/2$ pairs
- Each pair has some probability of being compared
- Add up all these probabilities
- Result: approximately $2n \times \log(n)$ comparisons on average

What log(n) means in practice:

- Array size 1,000 → log(1000) 7
- Array size 1,000,000 → log(1,000,000) 14

- Grows very slowly!

Real performance:

- 1,000 elements: ~14,000 comparisons
- 1,000,000 elements: ~28,000,000 comparisons
- Very manageable for modern computers!

Checkpoint 7c: If QuickSort makes about 2n×log(n) comparisons, roughly how many comparisons would you expect for an array of 10,000 elements? (Hint: log(10,000) 9)

7.7 The Big Picture

What we discovered:

- Random pivot selection is brilliant
- Most element pairs never get compared (they get separated)
- The few that do get compared don't hurt performance much
- Result: O(n log n) average performance

Why this analysis matters:

- Proves QuickSort is efficient in practice
- Explains why random pivots work so well
- Gives us confidence to use QuickSort on large datasets

Practical takeaway: When you see QuickSort in a library or system, you now know why it's there - it's not just fast, it's probably fast on average!

The Bottom Line: QuickSort's genius isn't just in its simplicity, but in how randomization naturally prevents the worst-case scenarios from happening often. Mathematics proves what programmers observe: it just works well in practice!

8 Advanced Topics

8.1 Median-of-Three Strategy

Practical Improvement: Choose pivot more carefully

Algorithm:

1. Consider first, middle, and last elements
2. Find median of these three values
3. Use median as pivot

Example: Array [8,3,2,5,1,4,7,6]

- Candidates: 8 (first), 5 (middle), 6 (last)

- Median of {5,6,8} = 6
- Use 6 as pivot

Benefits: Much better performance on nearly-sorted arrays

8.2 Comparison-Based Sorting Lower Bound

Fundamental Question: Can we sort faster than O(n log n)?

Theorem 5.5: No comparison-based sorting algorithm has worst-case running time better than O(n log n).

Proof Idea:

- Decision tree has n! leaves (all possible orderings)
- Each comparison gives binary choice
- Tree depth $\log(n!) = \Omega(n \log n)$

Implication: QuickSort (and MergeSort) are asymptotically optimal!

Checkpoint 8: Can you think of sorting algorithms that might beat the O(n log n) lower bound? What would they need to do differently?

9 Summary and Conclusions

9.1 Key Takeaways

Algorithm Design:

- Choose pivot element
- Partition around pivot (O(n) time, in-place)
- Recursively sort subarrays
- No combine step needed

Performance:

- **Worst case**: $O(n^2)$
- **Best case**: O(n log n)
- **Average case**: O(n log n) with random pivots

Practical Impact:

- Runs in place (memory efficient)
- Excellent average performance
- Widely used in practice

9.2 The Beauty of QuickSort

Algorithmic Elegance:

- Simple high-level structure
- Sophisticated performance analysis
- Optimal average-case complexity

Mathematical Beauty:

- Linearity of expectation application
- Exact formula: 2(n-1)log(n) expected comparisons
- Probability theory meets algorithm design

Historical Significance:

- Tony Hoare's masterpiece (1959)
- Foundation for modern sorting implementations
- Gateway to randomized algorithm design

Final Checkpoint: Now that you understand QuickSort, can you explain why it's often preferred over MergeSort in practice, despite having the same average-case complexity?

9.3 Programming Challenge

Implement and Experiment: Try implementing QuickSort with different pivot strategies:

1. First element pivot
2. Random element pivot

3. Median-of-three pivot

Compare: Count comparisons on various input types:

- Random arrays
- Sorted arrays
- Reverse-sorted arrays
- Arrays with duplicates

Observe: How does pivot choice affect performance in practice?

10 Questions and Discussion

10.1 Thank You!

What we covered:

- Partitioning concept and implementation
- Pivot selection strategies

- Complete running time analysis
- Theoretical foundations

Next steps:

- Implement QuickSort yourself
- Explore advanced partitioning schemes
- Study other randomized algorithms
- Apply to real-world data sets