# QuickSort

**CSE 373 Winter 2020**

**Instructor:** Hannah C. Tang

**Teaching Assistants:**

<table>
<thead>
<tr>
<th>Aaron Johnston</th>
<th>Ethan Knutson</th>
<th>Nathan Lipiarski</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amanda Park</td>
<td>Farrell Fileas</td>
<td>Sam Long</td>
</tr>
<tr>
<td>Anish Velagapudi</td>
<td>Howard Xiao</td>
<td>Yifan Bai</td>
</tr>
<tr>
<td>Brian Chan</td>
<td>Jade Watkins</td>
<td>Yuma Tou</td>
</tr>
<tr>
<td>Elena Spasova</td>
<td>Lea Quan</td>
<td></td>
</tr>
</tbody>
</table>
Approximately how long did HW7: HuskyMaps take?

A. 0-4 hours
B. 5-9 hours
C. 10-14 hours
D. 15-19 hours
E. 20-24 hours
F. 25-29 hours
G. 29+ hours
H. I’m not done / I don’t want to say ...
Announcements

❖ HW8 (Seam Carving) has been released!
  ❖ 🚨 NOTE 🚨: We are NOT offering late days for this homework. If you think you’ll need extra time, pretend it’s due on Tuesday instead of Friday

❖ You can always make appointments with staff (TAs or me) to discuss anything: homework, concepts, imposter syndrome, and more

❖ Your health is more important than this class!
  ❖ COVID-19 announcements/updates: [https://uw.edu/coronavirus](https://uw.edu/coronavirus)
  ❖ Will adjust the in-class participation (PollEverywhere) policy so that you can remain at home for the rest of the quarter
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ QuickSort Intro

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
An (Oversimplified) Summary of Sorting Algorithms So Far

- **SelectionSort**: find the smallest item and put it in the front
- **HeapSort**: SelectionSort, but use a heap to find the smallest item
- **MergeSort**: Merge two sorted halves into one sorted whole

- **QuickSort**:
  - Much stranger core idea: *Partitioning*
  - Invented by Sir Tony Hoare in 1960, at the time a novice programmer
  - Interview: [https://www.bl.uk/voices-of-science/interviewees/tony-hoare/audio/tony-hoare-inventing-quicksort](https://www.bl.uk/voices-of-science/interviewees/tony-hoare/audio/tony-hoare-inventing-quicksort)
  - “I thought, that’s a nice exercise: how would I program sorting?”
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ QuickSort Intro

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
Partitioning Definition

- **Partitioning** an array `a[]` on a pivot `x=a[i]` rearranges `a[]` so that:
  - `x` moves to position `j` (may be the same as `i`)
  - All entries to the left of `x` are `<= x`
  - All entries to the right of `x` are `>= x`

- Which of these are valid partitions?

A.  
```
4 5 9 10 10 330 550
```

B.  
```
5 4 9 10 10 550 330
```

C.  
```
5 9 10 4 10 550 330
```

D.  
```
5 9 10 4 10 550 330
```

The correct answer is **B**.
Your Turn! Implement Partitioning

- Write pseudocode to implement the following:
  - Given an array of elements, rearrange the array so that all the less-than-0\(^{th}\)-value elements are to the left of the 0\(^{th}\) value and all greater-than-0\(^{th}\)-value elements are to the right

- Constraints:
  - Your algorithm must complete in \(O(N \log N)\) time, but ideally \(\Theta(N)\)
  - Your algorithm must use \(O(N)\) space, but ideally \(\Theta(1)\)
  - You may use any data structure (eg, BSTs, stacks/deques/queues, etc)
    - Please don’t copy the two solutions discussed in the reading: sort and copy-less-than-then-copy-greater-than
  - Relative order does NOT need to stay the same
Describe your implementation in a sentence or two

Constraints:
- Your algorithm must complete in $O(N \log N)$ time, but ideally $\Theta(N)$
- Your algorithm must use $O(N)$ space, but ideally $\Theta(1)$
- You may use any data structure (eg, BSTs, stacks/deques/queues, etc)
  - Please don’t copy the two solutions discussed in the reading: sort and copy-lessthan-then-copy-greaterthan
- Relative order does NOT need to stay the same

Input:

| 6 | 8 | 3 | 1 | 2 | 7 | 4 |

Valid outputs:

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Partitioning Implementations

❖ Sort the elements (described in the reading)
  ▪ Note: this implies that **partitioning reduces to sorting**

❖ Three-pass: copy “less than”s, then copy pivots, finally copy “greater-than”s
  ▪ Described in reading
  ▪ Demo:
    [https://docs.google.com/presentation/d/16pOLboxhtJlaDxF7iRT5XcItDKmwab_wbvZ4wPmJYk/edit](https://docs.google.com/presentation/d/16pOLboxhtJlaDxF7iRT5XcItDKmwab_wbvZ4wPmJYk/edit)
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ **QuickSort Intro**

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
Context for QuickSort’s Invention

❖ In 1960, exchange student (!!) Tony Hoare worked on a translation program between Russian and English

Sentence of N words

“The cat wore a beautiful hat.”

Dictionary of D english words

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>beautiful</td>
<td>красивая</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>cat</td>
<td>кошка</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

❖ O(N log D) if we binary search the dictionary: not bad!

❖ ... alas, the dictionary was stored on magnetic tape. Seeks require very slow physical movement of a tape head
Constants Matter (Sometimes)

- Iterating through an in-memory array != iterating through a magnetic tape

**Question 5:**

[3 pts] Let’s map the latency of common computer operations to the human-scale operations required for studying for the 333 final. You may use the following:

A. Reading a sticky note on your monitor (0.5 secs)
B. Finding the right page/paragraph in the textbook kept next to your monitor (2 mins)
C. Asking on Piazza (36 mins)
D. Texting another 333 student for the answer (1 hour)
E. Requesting a scanned article from UW Libraries (2 days)
F. Buying the physical textbook *without Amazon Prime* (1 week)
G. Re-taking CSE 351 and then re-taking CSE 333 (20 weeks)
H. Buying the physical textbook *currently on Jupiter* (6 years)
I. Buying the physical textbook *currently in the Alpha Centauri system* (78,000 years)

<table>
<thead>
<tr>
<th>Computer Operation</th>
<th>Human Analogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>A</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>B</td>
</tr>
<tr>
<td>Packet round trip within same datacenter</td>
<td>F</td>
</tr>
<tr>
<td>Disk seek</td>
<td>G</td>
</tr>
<tr>
<td>Packet round trip across a submarine cable</td>
<td>H</td>
</tr>
</tbody>
</table>
Context for QuickSort’s Invention

- O(N log D) if we binary search the dictionary: not bad!
  - ... alas, the dictionary was stored on magnetic tape. Seeks require very slow physical movement of a tape head. Moving the head N times was too slow

- Better solution: sort the sentence and scan the dictionary (ie, the tape) in a single pass

- Named the resultant algorithm “QuickSort”, although “PartitionSort” may be clearer
QuickSort is Partitioning

- After partitioning on 5:
  - 5 is in its “correct place” (i.e., where it’d be if the array were sorted)
  - Can now sort two halves separately (e.g., through recursive use of partitioning)
QuickSort is Partitioning

QuickSort(a[]): p = SelectPivot(a)
a1, a2 = Partition(a, p)
QuickSort(a1)
QuickSort(a2)

- For Naïve QuickSort:
  - SelectPivot() selects the 0th element
  - Partition() copies into a new array using three-pass method (see reading)

- Demo: https://docs.google.com/presentation/d/1QjAs-zx1i0_XWllqsKtxeb-ieuao9jNLkN-gW9QxAD0/present?ueb=true&slide=id.g463de7561_042
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ QuickSort Intro

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
Best Case: Pivot Always Lands in the Middle

Only size 1 problems remain, so we’re done.
**Best Case: Runtime**

Only size 1 problems remain, so we’re done

Total work at each level:

\[ \approx N \]

\[ \approx \frac{N}{2} + \approx \frac{N}{2} = \approx N \]

\[ \approx \frac{N}{4} \times 4 = \approx N \]

\[ h = \Theta(\log N) \]

So best-case runtime is \( \Theta(N \log N) \)
Worst Case: Pivot Always Lands at Beginning

- Give an example of an array that would follow the pattern to the right.
  - 1 2 3 4 5 6

- What is the runtime $\Theta(\cdot)$?
  - $N^2$
Randomized Case

❖ Suppose pivot always ends up at least 10% from either edge

❖ Work at each level: \( O(N) \) and Runtime is \( O(NH) \)
  ▪ \( H \) is approximately \( \log_{10/9} N = O(\log N) \)

❖ Randomized Case: \( O(N \log N) \)
  ▪ Even if you’re unlucky enough to have a pivot that never lands anywhere near the middle but is at least 10% from one edge, runtime is still \( O(N \log N) \)
QuickSort Runtime, Empirically

- For N items:
  - Mean number of compares to complete Quicksort: $\sim 2N \ln N$
  - Standard deviation: $\sqrt{(21 - 2\pi^2)/3N} \approx 0.6482776N$

Lots of arrays take 12,000ish compares to sort with Quicksort

A very small number take 15,000ish compares to sort with Quicksort.

Empirical histogram for quicksort compare counts (10,000 trials with N = 1000)

Chance of taking 1,000,000ish compares is effectively zero.

- For more, see:
QuickSort Performance

❖ Theoretical analysis:
  ▪ Best case: \( \Theta(N \log N) \)
  ▪ Worst case: \( \Theta(N^2) \)
  ▪ **Randomized case**: \( \Theta(N \log N) \) expected

❖ Compare this to Mergesort
  ▪ Best case: \( \Theta(N \log N) \)
  ▪ Worst case: \( \Theta(N \log N) \)

❖ Why is QuickSort empirically faster than MergeSort in the best and randomized cases?
  ▪ No obvious reason why, just need to run experiments to show that constants are better
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ QuickSort Intro

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
Avoiding QuickSort’s Worst Case

❖ If pivot lands “somewhere good”, Quicksort is $\Theta(N \log N)$ 🥂

❖ However, the very rare $\Theta(N^2)$ cases do happen in practice ⌒
  - **Bad ordering**: Array already in (almost-)sorted order
  - **Bad elements**: Array with all duplicates

❖ What can we do to avoid worst case behavior?

❖ Three philosophies:
  1. **Randomness**: pick a random pivot; shuffle before sorting
  2. **Smarter Pivot Selection**: calculate or approximate the median
  3. **Introspection**: switch to safer sort if recursion goes too deep
#1: Randomness

- Dealing with Bad Ordering:
  - Strategy 1: Pick pivots randomly
  - Strategy 2: Shuffle before you sort

- Dealing with Bad Elements (ie, duplicates):
#2a: Smarter Pivot Selection (Constant Time)

- Any algorithm for picking a pivot which requires constant time and determinism (ie, not random) has a corresponding family of dangerous inputs

- Dealing with Bad Elements (ie, duplicates):
  - 😭
#2b: Smarter Pivot Selection (Linear Time)

- Dealing With Bad Ordering:
  - We can calculate the actual median in linear time!
  - Worst-case is $\Theta(N \log N)$, but constants make it slower than MergeSort 😭
  - Note: we can adapt QuickSort into QuickSelect
    - Selects the k-th element in $\Theta(N)$ time; we can use it to find the N/2 aka median element

- Dealing with Bad Elements (ie, duplicates):
#3: Introspection

- If recursion depth exceeds some threshold (e.g., 10 log N), switch to MergeSort
  - Reasonable, but not common in practice

- Dealing With Bad Ordering:
  - \_\_\_(ツ)_/\_

- Dealing with Bad Elements (i.e., duplicates):
  - \_\_\_(ツ)_/\_
Ultimately ...

- As we saw with LLRB trees and B-trees, having a “100% guarantee” against worst-case input came with a cost
  - Here, our “100% guarantee” changed QuickSort’s constants so that it became slower than MergeSort in the cases where it used to be faster: best-case and randomized-case

- Ultimately, most QuickSort implementations choose a few “reasonable protections” against pessimal input to maintain its performance against MergeSort in best-case and randomized-case
  - If you, the implementer, need a “100% guarantee” against worst-case input you should choose MergeSort instead. You should also recognize that you’re paying for that guarantee with a slower runtime in most other cases
Lecture Outline

❖ Comparison Sorts Review

❖ Partitioning

❖ QuickSort Intro

❖ Analyzing QuickSort’s Runtime

❖ Avoiding QuickSort’s Worst Case

❖ QuickSort in Practice
Decisions When Implementing QuickSort

❖ How to select pivot?
  ▪ Naïve QuickSort uses 0th element
  ▪ Dual-pivot QuickSort uses 1/3rd-2/3rd

❖ How to partition?
  ▪ Naïve QuickSort uses a stable three-pass partition
  ▪ Dual-pivot QuickSort uses three-way partition

QuickSort(a[]):
  p = SelectPivot(a)
  a1, a2 = Partition(a, p)
  QuickSort(a1)
  QuickSort(a2)
Pivot Selection: Median-of-Three

❖ Median-of-Three \textit{approximates} the true median in $\Theta(1)$ time
  ▪ Pick 3 items and take the median of the sample

❖ Options for picking 3:
  ▪ Randomly choose 3 indices
  ▪ Pick first, middle, last
  ▪ …?

❖ “Good enough” for protecting against bad ordering
  ▪ Intuitively: it’s not-that-hard to one bad pivot, but it’s pretty-hard to pick three bad pivots simultaneously

\begin{verbatim}
if (a < b)
  if (b < c) return b;
  else if (a < c) return c;
  else return a;
else
  if (a < c) return a;
  else if (b < c) return c;
  else return b;
\end{verbatim}
Partitioning: Hoare Partitioning

- This is the original QuickSort partitioning algorithm
  - Good constants: single-pass and in-place
  - Yields an unstable sort

- Idea: initialize two pointers, L and R
  - L loves small items < pivot
  - R loves large items > pivot
  - Walk towards each other, swapping anything they don’t like

- Demo:
  https://docs.google.com/presentation/d/1DOnWS59PJOa-LaBfttPRselpwLGefZkn450TMSSUiQY/pub?start=false&loop=false&delayms=3000&slide=id.g463de7561_042
Partitioning: Three-Way Partition

- Pick *two* pivots
  - Same intuition as median-of-three: it’s hard to pick two bad pivots simultaneously

- Like Hoare Partitioning, use two pointers walking to the middle
  - But split array into three pieces, not two
  - Good constants: single-pass and in-place; $\log_3 N$ vs $\log_2 N$
  - Still results in an unstable sort
Case Study: Dual-Pivot QuickSort

- In 2009, Dual-Pivot QuickSort was introduced to the world by a previously-unknown guy in a Java developers’ forum

- It is now the de-facto QuickSort implementation for many languages, including Java’s Arrays.sort(), Python’s unstable sort, etc
Case Study: Dual-Pivot QuickSort

- Dual-Pivot QuickSort combines several ideas:
  - InsertionSort when array length < 48 elements
    - Provides some protection against bad ordering and bad elements
  - Three-way partition
    - Good constants: single-pass and in-place; $\log_3 N$ vs $\log_2 N$
    - Dual “middle pivots” provides some protection against bad ordering
      - 1/3\textsuperscript{rd} and 2/3\textsuperscript{rd} elements instead of “the end elements” (first and last)
**tl;dr**

- Constants matter in the real world, even if they don’t matter asymptotically!

<table>
<thead>
<tr>
<th></th>
<th>Best-Case Time</th>
<th>Worst-Case Time</th>
<th>Space</th>
<th>Stable?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>In-Place HeapSort</td>
<td>(\Theta(N))</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td>Slow in practice</td>
</tr>
<tr>
<td>MergeSort</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N))</td>
<td>Yes</td>
<td>Fastest stable sort</td>
</tr>
<tr>
<td>In-Place InsertionSort</td>
<td>(\Theta(N))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(1))</td>
<td>Yes</td>
<td>Best for small or partially-sorted input</td>
</tr>
<tr>
<td>Naïve QuickSort</td>
<td>(\Theta(N \log N))</td>
<td>(\Theta(N^2))</td>
<td>(\Theta(N))</td>
<td>Yes</td>
<td>&gt;=2x slower than MergeSort</td>
</tr>
<tr>
<td>Dual-Pivot QuickSort</td>
<td>(\Omega(N))</td>
<td>(\Omega(N^2))</td>
<td>(\Theta(1))</td>
<td>No</td>
<td>Fastest comparison sort</td>
</tr>
</tbody>
</table>