Quicksort Implementation

https://cs.pomona.edu/classes/cs140/
Outline

**Topics and Learning Objectives**

- Learn how quicksort works
- Learn how to partition an array

**Exercise**

- Partitioning
Extra Resources

- https://me.dt.in.th/page/Quicksort/
- https://www.youtube.com/watch?v=ywWBy6J5gz8
- CLRS Chapter 7
Quicksort

• A practical and simple algorithm
• The running time = $O(n \lg n)$
• Superior to other $O(n \lg n)$ in some respects
• The hidden constants are small (hidden by Big-O)
• Our first stochastic algorithm
Quicksort

Input: an array of $n$ elements in any order

Output: a reordering of the input array such that the elements are in non-decreasing order

Key idea of Quicksort: **partition** the array around a pivot element
Key concept of Quicksort

- Pick an element and call it the **pivot**

- **Partition** (rearrange) the elements so that:
  - Everything to the **left** of the pivot is **less than** the pivot
  - Everything to the **right** of the pivot is **greater than** the pivot
  - Let’s ignore ties for now

- This is a partial sorting into “buckets”

- What can you tell me about the pivot?

  - **Pivot** is now in the correct spot (we’ve made progress!)

What would be the running time of calling partition on every element?
Partitioning
Partitioning
Pivot around “hello”

[“hello”, “are”, “you”, “how”, “today”, “doing”, “class”]
Quicksort (NOT IN-PLACE PARTITIONING)

1. FUNCTION BadQuicksort(array)
2.    IF array.length ≤ 1
3.        RETURN array
4.      pivot_index = ChoosePivot(array.length)
5.      left_array, right_array = Partition(array, pivot_index)
6.    left_sorted = BadQuicksort(left_array)
7.    right_sorted = BadQuicksort(right_sorted)
8. RETURN left_sorted ++ array[pivot_index] ++ right_sorted

What is the recurrence equation for Quicksort?
Partitioning the Easy Way

- How would you partition? (how did we perform a merge?)
- Copy all elements to a new array
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array

Original array: 3 8 2 5 1 4 7 6
New array:  

Copy all elements to a new array:
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array

Original array

New array
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array

Original array

```
3  8  2  5  1  4  7  6
```

New array

```
2  3  5  8
```
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array

Original array: 3 8 2 5 1 4 7 6
New array: 2 1 5 8
Partitioning the Easy Way

• How would you partition? (how did we perform a merge?)
• Copy all elements to a new array

Original array

New array
Partitioning the Easy Way

- How would you partition? (how did we perform a merge?)
- Copy all elements to a new array

Original array

| 3 | 8 | 2 | 5 | 1 | 4 | 7 | 6 |

New array

| 1 | 2 | 3 | 6 | 7 | 4 | 5 | 8 |

- This would be like merge sort.
- Lots of memory allocations (one for each node in the recursion tree).
Partitioning the Easy Way

- Nothing inherently wrong with this approach in theory
- But can we do the same thing without the extra memory?

- Note: implementing merge sort “in-place” is possible
- You can do so with an iterative (stack based) approach
Partitioning In-Place

• **For now**, assume that the **pivot** is in the **first** spot of a **subarray**
• (we can swap the pivot with the first spot if needed)

• **Idea**: gradually build up a subarray that is correctly partitioned by scanning through the array

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>&lt; P</th>
<th>&gt; P</th>
<th>Un-partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Partitioning In-Place

Index one to the right of the “smaller-than” partition

Index one to the right of the “larger-than” partition
To which partition does 8 belong?

How do I put it there?

How should we initialize \(i\) and \(j\)?
Index one to the right of the “smaller-than” partition

Index one to the right of the “larger-than” partition

To which partition does 2 belong?

How do I put it there?
To which partition does 2 belong?

How do I put it there?
To which partition does 2 belong?

How do I put it there?

Now what?
To which partition does 5 belong?

How do I put it there?
Index one to the right of the “smaller-than” partition
Index one to the right of the “larger-than” partition

To which partition does 1 belong?

How do I put it there?
To which partition does 1 belong?

How do I put it there?
To which partition does 1 belong?
How do I put it there?
Now what?
<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>&lt; P</th>
<th>&gt; P</th>
<th>Un-partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index one to the right of the "smaller-than" partition

Index one to the right of the "larger-than" partition

Un-partitioned
<table>
<thead>
<tr>
<th>P</th>
<th>&lt; P</th>
<th>&gt; P</th>
<th>Un-partitioned</th>
</tr>
</thead>
</table>

Index one to the right of the “smaller-than” partition

Index one to the right of the “larger-than” partition

Now what?
Index one to the right of the “smaller-than” partition

Index one to the right of the “larger-than” partition
1. **FUNCTION** Partition(array, left_index, right_index)

2. # Partition the subarray array[left_index ..< right_index]

3. # around the value at left_index

4. pivot_value = array[left_index]

5. i = left_index + 1

6. FOR j IN [left_index + 1 ..< right_index]

7. IF array[j] < pivot_value

8. swap(array, i, j)

9. i = i + 1

10. swap(array, left_index, i - 1)

11. RETURN i - 1

What is the asymptotic running time?

1. O(n), where n is right_index - left_index

2. In-place
   no extra memory
1. **FUNCTION** QuickSort(array, left_index, right_index)
2. IF (left_index + 1) ≥ right_index
3. RETURN
4.
5. MovePivotToLeft(left_index, right_index)
6. pivot_index = Partition(array, left_index, right_index)
7.
8. QuickSort(array, left_index, pivot_index)
9. QuickSort(array, pivot_index + 1, right_index)

Our **Partition** function expects the pivot element to be at left_index
FUNCTION QuickSort(array, left_index, right_index)

2. IF left_index >= right_index
3. RETURN

5. MovePivotToLeft(left_index, right_index)
6. pivot_index = Partition(array, left_index, right_index)

8. QuickSort(array, left_index, pivot_index)
9. QuickSort(array, pivot_index + 1, right_index)