QuickSort (Ch. 7)

A divide-and-conquer algorithm

**Divide:** Rearrange the array \(A[1..n]\) into two (one possibly empty) subarrays \(A[1..q-1]\) and \(A[q+1..n]\) such that each element of \(A[1..q-1]\) \(\leq A[q]\) and each element of \(A[q+1..n]\) > \(A[q]\) after computation of index \(q\).

**Conquer:** Sort the two subarrays \(A[1..q-1]\) and \(A[q+1..n]\) recursively.

**Combine:** No work is needed to combine subarrays since they are sorted in-place.

**Correctness of Quicksort**

**Claim:** Partition satisfies the specifications of the Divide step.

**Loop invariant:** At the beginning of each iteration of the for loop (lines 3-6), for any array index \(k\),

1. If \(p \leq k \leq j\), then \(A[k] \leq x\).
2. If \(i+1 \leq k \leq j-1\), then \(A[k] > x\).
3. If \(k=r\), then \(A[k] = x\).

**Initialization:** \(i = p - 1\) and \(j = p\).

**Inductive Hypothesis:** Assume the invariant holds through iteration \(j = k < n - 1\).

**Inductive Step:** At the start of iteration \(k+1\), either \(A[j] > x\) or \(A[j] \leq x\). If \(A[j] > x\), \(j\) is incremented and cond 2 holds for \(A[j] > x\) with no other changes. If \(A[j] \leq x\), \(A[j+1]\) and \(A[j]\) are swapped, and then \(j\) is incremented. Cond 1 holds for \(A[j]\) after swap by the IH. By the IH, the item in \(A[j]\) was in \(A[i+1]\) during the last iteration, and was \(> x\) then, so cond 2 holds at the end of iteration \(k+1\).

**Termination:** At termination, \(j = r\) and \(A\) has been partitioned into 3 sets: items \(\leq x\), items \(> x\), and \(A[j] = x\). The item in \(A[i+1]\) > \(x\) in the last iteration, so it remains > \(x\) at termination, when it is swapped with the element in \(A[r] = x\).

QED

**QuickSort**

**Input:** An \(n\)-element array \(A\) (unsorted).

**Output:** An \(n\)-element array \(A\) in non-decreasing order.

**Quicksort(A, p, r)**
1. If \(p < r\)
2. \(q = \text{Partition}(A, p, r)\)
3. \(\text{Quicksort}(A, p, q-1)\)
4. \(\text{Quicksort}(A, q+1, r)\)

**Partition(A, p, r)**
1. \(x = A[r]\) // choose pivot
2. \(i = p - 1\)
3. for \(j = p\) to \(r - 1\)
4. if \(A[j] \leq x\)
5. \(i = i + 1\)
6. swap \(A[i]\) and \(A[j]\)
7. swap \(A[i+1]\) and \(A[r]\)
8. return \(i + 1\)

**Initial call:**
\(\text{Quicksort}(A, 1, \text{A.length})\)

**Divide:** Rearrange the array \(A[p..r]\) into two (one possibly empty) subarrays \(A[p..q-1]\) and \(A[q+1..r]\) such that each element of \(A[p..q-1]\) \(\leq A[q]\) and each element of \(A[q+1..r]\) > \(A[q]\) after computation of index \(q\).
Quicksort Running Time

\[ T(n) = T(q - p) + T(r - q) + O(n) \]

The value of \( T(n) \) depends on the location of \( q \) in the array \( A[p..r] \).

Since we don't know this in advance, we must look at worst-case, best-case, and average-case partitioning.

Quicksort Best-case

\[ T(n) = T(q - p) + T(r - q) + O(n) \]

Best-case partitioning: Each partition results in a \([n/2] : [n/2] - 1\) split (i.e., close to balanced split each time), so recurrence is

\[ T(n) = 2T(n/2) + \Theta(n) \]

By the master theorem, this recurrence evaluates to \( \Theta(n \log n) \).

Quicksort Average-case

Intuition: Some splits will be close to balanced and others close to unbalanced ⇒ good and bad splits will be randomly distributed in recursion tree.

The running time will be (asymptotically) bad only if there are many bad splits in a row.

- A bad split followed by a good split results in a good partitioning after one extra step.
- Implies a \( \Theta(n \log n) \) running time (with a larger constant factor to ignore than if all splits were good).

Randomized Quicksort

How can we modify Quicksort to get good average case behavior on all inputs?

2 techniques:
1. randomly permute input prior to running Quicksort. Will produce tree of possible executions, most of them finish fast.
2. choose partition randomly at each iteration instead of choosing element in highest array position.

Randomized-Partition(A, p, r)

1. \( i = \text{Random}(p, r) \)
2. swap \( A[r] \leftrightarrow A[i] \)
3. return Partition(A, p, r)

In section 7.4, a probabilistic analysis is presented, showing that the expected running time of Randomized-Quicksort is \( O(n \log n) \).

Quicksort Running Time

Worst-case partitioning: Each partition results in a \( 0 : n-1 \) split \( T(0) = \Theta(1) \) and the partitioning costs \( \Theta(n) \), so the recurrence is

\[ T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) \]

This is an arithmetic series which evaluates to \( \Theta(n^2) \). So worst-case for Quicksort is no better than Insertion sort!

What does the input look like in Quicksort's worst-case?

Sorting Algorithms

- A sorting algorithm is comparison-based if the only operation we can perform on keys is to compare them.
- A sorting algorithm is in place if only a constant number of elements of the input array are ever stored outside the array.

Running Time of Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst-case</th>
<th>average-case</th>
<th>best-case</th>
<th>in place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>( n^2 )</td>
<td>( n^2 )</td>
<td>( n )</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>( n^2 )</td>
<td>( n \log n )</td>
<td>( n \log n )</td>
<td>yes</td>
</tr>
</tbody>
</table>