Quicksort

Divide-and-Conquer algorithm for sorting an array. It works as follows:

1. If the input array has less than two elements, nothing to do.
   Otherwise, do the following partitioning subroutine: Pick a particular key called the pivot and divide the array into two subarrays as follows:

   \[
   \begin{array}{cc}
   \leq \text{pivot} & \text{pivot}\geq \end{array}
   \]

2. Sort the two subarrays recursively.

Partitioning

Algorithm Partition(A, p, r)
1. \( \text{pivot} \leftarrow A[r] \)
2. \( i \leftarrow p - 1 \)
3. for \( j \leftarrow p \) to \( r - 1 \) do
4.    if \( A[j] \leq \text{pivot} \) then
5.        \( i \leftarrow i + 1 \)
6.        exchange \( A[i], A[j] \)
7.    exchange \( A[i + 1], A[r] \)
8. return \( i + 1 \)

Same version as [CLRS]
Analysis of Quicksort

▶ The size of an instance \((A, p, r)\) is \(n = r - p + 1\).
▶ Basic operations for sorting are comparisons of keys. We let

\[
C(n)
\]

be the worst-case number of key-comparisons performed by \textsc{Quicksort}(A, p, r). We shall try to determine \(C(n)\) as precisely as possible.
▶ It is easy to verify that the worst-case running time \(T(n)\) of \textsc{Quicksort}(A, p, r) is \(\Theta(C(n))\) if a single comparison requires time \(\Theta(1)\).

\[
T(n) = \Theta(C(n) \cdot \text{cost per comparison}).
\]

\textit{ADS: lect 7 – slide 5 – 11th October, 2013}

Worst-case Analysis of Quicksort

▶ We get the following recurrence for \(C(n)\):

\[
C(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
\max_{1 \leq k \leq n} \left( C(k-1) + C(n-k) \right) + n & \text{if } n \geq 2 
\end{cases}
\]
▶ Intuitively, worst-case seems to be \(k = 1\) or \(k = n\), i.e., everything falls on one side of the partition. This happens, e.g., if the array is sorted.

\textit{ADS: lect 7 – slide 7 – 11th October, 2013}

Analysis of Partition

▶ \textsc{Partition}(A, p, r) does exactly \(n - 1\) comparisons for every input of size \(n\).
▶ There is also an initial comparison \(p < r\) done on line 1 of \textsc{Quicksort}.
▶ Therefore in our theoretical analysis, we can assume, that apart from recursive calls, we do \(n\) comparisons on every input.

\textit{ADS: lect 7 – slide 6 – 11th October, 2013}

Worst-Case Analysis (cont’d)

▶ Lower Bound: \(C(n) \geq \frac{1}{2}n(n+1) = \Omega(n^2)\).

\textit{Proof:}

\[
C(n) \geq C(n-1) + n \\
\geq C(n-2) + n + n - 1 \\
\vdots \\
\geq \sum_{i=0}^{n-1} i + 1 = \sum_{i=1}^{n} i = \frac{1}{2}n(n+1).
\]

▶ Upper Bound: \(C(n) \leq O(n^2)\).

\textit{CLASS?}
▶ Thus

\[
C(n) = \Theta(n^2).
\]

\textit{ADS: lect 7 – slide 8 – 11th October, 2013}
Best-Case Analysis

- \( B(n) \) = number of comparisons done by Quicksort in the best case.
- **Recurrence:**
  \[
  B(n) = \begin{cases} 
  1 & \text{if } n \leq 1 \\
  \min_{1 \leq k \leq n} \left( B(k-1) + B(n-k) \right) + n & \text{if } n \geq 2
  \end{cases}
  \]
- Intuitively, the best case is if the array is always partitioned into two parts of the same size. This means
  \[
  B(n) \approx 2B(n/2) + \Theta(n),
  \]
  which implies \( B(n) = \Theta(n \lg(n)) \).

Average Case Analysis

- \( A(n) \) = number of comparisons done by Quicksort on average if all input arrays of size \( n \) are considered equally likely.
- **Intuition:** The average case is closer to the best case than to the worst case, because only repeatedly very unbalanced partitions lead to the worst case.
- **Recurrence:**
  \[
  A(n) = \begin{cases} 
  1 & \text{if } n \leq 1 \\
  \sum_{k=1}^{n} \frac{1}{n} \left( A(k-1) + A(n-k) \right) + n & \text{if } n \geq 2
  \end{cases}
  \]
- **Solution:**
  \[
  A(n) \approx 2n \ln(n).
  \]

Average Case Analysis in Detail

We shall prove that for all \( n \geq 1 \) we have
\[
A(n) \leq 2(\ln(n) + 1/3)(n + 1).
\]
Clearly, (⋆) holds for \( n = 1 \). So assume that \( n \geq 2 \). We have
\[
A(n) = \sum_{1 \leq k \leq n} \frac{1}{n} \left( A(k-1) + A(n-k) \right) + n
\]
\[
= 2 \sum_{k=0}^{n-1} A(k) + n.
\]
Thus
\[
nA(n) = 2 \sum_{k=0}^{n-1} A(k) + n^2. \quad (⋆⋆)
\]
Note that (⋆⋆) does not hold for \( n = 1 \). (For \( n = 1 \) the l.h.s. \( 1A(1) = 1 \), but the r.h.s. is 3.)
Average Case Analysis in Detail (cont’d)

\[
\frac{A(n)}{n+1} \leq \frac{A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n-1}
\]

\[
\vdots
\]

\[
\leq \frac{A(2)}{3} + 2 \sum_{k=3}^{n} \frac{1}{k}
\]

\[
= \frac{4}{3} + 2 \sum_{k=3}^{n} \frac{1}{k} = 1/3 + 2 \sum_{k=2}^{n} \frac{1}{k}
\]

It is easy to verify this result by induction. Thus

\[
\frac{A(n)}{n+1} \leq 1/3 + 2 \sum_{k=3}^{n} \frac{1}{k} = 1/3 + 2 \sum_{k=2}^{n} \frac{1}{k} \leq 1/3 + 2 \int_{1}^{n} \frac{1}{x} = 1/3 + 2 \ln(n).
\]

Multiplying by \((n + 1)\) completes the proof of \((*)\).

Median-of-Three Partitioning

Idea: Use the median of the first, middle, and last key as the pivot.

Algorithm M3Partition\((A, p, r)\)
4. Partition\((A, p + 1, r - 1)\)

Note that \(\text{M3Partition}\((A, p, r)\) only requires 1 more comparison than \(\text{Partition}\((A, p, r)\)

Median-of-Three Partitioning (cont’d)

Algorithm M3Quicksort\((A, p, r)\)
1. if \(p < r\) then
2. \(q \leftarrow \text{M3Partition}\((A, p, r)\)
3. \(\text{M3Quicksort}(A, p, q - 1)\)
4. \(\text{M3Quicksort}(A, q + 1, r)\)

In can be shown that the worst-case running time of \(\text{M3Quicksort}\) is still \(\Theta(n^2)\), but at least in the case of an almost sorted array (and in most other cases that are relevant in practice) it is very efficient.

Improvements

- Use insertion sort for small arrays.
- Iterative implementation.

Main Question

Is there a way to avoid the bad worst-case performance, and in particular the bad performance on sorted (or almost sorted) arrays?

Different strategies for choosing the pivot-element help (in practice).
Randomized Quicksort

Idea: Use key of random element as the pivot.

Algorithm RPartition(A, p, r)
1. k ← Random(p, r) ⊞ choose k randomly from {p, ..., r}
2. exchange A[k], A[r]
3. Partition(A, p, r)

Algorithm Randomized Quicksort(A, p, r)
1. if p < r then
2. q ← RPartition(A, p, r)
3. Randomized Quicksort(A, p, q − 1)
4. Randomized Quicksort(A, q + 1, r)

Analysis of Randomized Quicksort

The running time of Randomized Quicksort on an input of size n is a random variable.

An analysis similar to the average case analysis of Quicksort shows:

Theorem
For all inputs (A, p, r), the expected number of comparisons performed during a run of Randomized Quicksort on input (A, p, r), is at most 2(ln(n) + 1/3)(n + 1), where n = r − p + 1.

Corollary
Thus the expected running time of Randomized Quicksort on any input of size n is Θ(n lg(n)).