Information Processing Letters 114 (2014) 137–139

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

An intuitive and simple bounding argument for Quicksort

Michael L. Fredman

Rutgers University, New Brunswick, United States

ARTICLE INFO

Article history:
Received 23 September 2013
Received in revised form 27 October 2013
Accepted 30 October 2013
Available online 5 November 2013
Communicated by M. Chrobak

Keywords:
Quicksort
Algorithms
Analysis of algorithms
Randomized
Intuitive

The purpose of this paper is to demonstrate the $O(n \log n)$ expected running time of Hoare's Quicksort algorithm [2] in a manner that is conceptually simple, reflects common intuition about the algorithm, and is nearly free of calculation.

Our treatment of Quicksort begins by defining a closely related algorithm, *Insistent-Q-Sort*, designed to have an easily derived $O(n \log n)$ estimation of expected cost, and then goes on to show that the derivation of the Insistent-Q-Sort bound, with a little refinement, is applicable to Quicksort itself. Moreover, the design of Insistent-Q-Sort captures the intuition that informs the individual to anticipate the $O(n \log n)$ bound for Quicksort. The standard treatments of Quicksort (e.g. see [4, Section 5.2.2] and [1, Section 7.4.2]) either center on a non-trivial recurrence equation, or focus on the individual probabilities that specified comparisons get executed. These treatments are not difficult and culminate in strong estimates, but they involve calculation and are not intuitive. We concede, however, that our method is too simple to provide a tight result, being off by a constant factor.

1. Insistent-Q-Sort

We consider the randomized version of Quicksort, for which the pivot is randomly selected from the input array. Assume (for now) that the input consists of n distinct data values – the usual assumption. For $n > 1$, two recursive sub-calls are spawned: one acting on values less than the pivot, and the other on values greater than the pivot. For $n = 0$ or 1, the call is terminal, and no comparisons are executed. Define the main invocation of Quicksort to be a *level*-0 call, and generally let the level-k calls, k positive, be those invoked by level-$(k-1)$ calls. There can be up to 2^k level-k calls collectively taking place on non-overlapping sub-arrays of the input array. Focusing on the comparisons executed (to implement the partition at a non-terminal sub-call), they total to at most n among the calls on any fixed level. Now in the ideal circumstance the pivot selected by any sub-call would be such that a reasonably balanced partitioning would result; namely, the sizes of each of the resulting sub-arrays would be at most $\frac{3}{4}$ of the size of the parent sub-array. (Any alternative value between $\frac{1}{2}$ and 1 would provide a suitable substitute for $\frac{3}{4}$.) We shall refer to such a partitioning as being *balanced*. That this should hold "often enough" provides the intuition supporting the plausibility of an $O(n \log n)$ expected

E-mail address: fredman@cs.rutgers.edu.

0020-0190/$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2013.10.010

running time for Quicksort. Insistent-Q-Sort uniformly *insists*, in every non-terminal sub-call, that the partitioning be balanced. This is accomplished by repeatedly drawing random candidate pivots until a balanced partitioning is obtained, refraining from executing sub-calls until this occurs. Each candidate for the pivot, successful or otherwise, participates in $n-1$ comparisons.

Observe that the change to Quicksort that defines Insistent-Q-Sort removes considerable uncertainty with respect to the number of sub-call levels: The size of the sub-array acted upon by any level-k sub-call is at most $(\frac{3}{4})^k \cdot n$ since all partitionings are balanced, so that maximum possible value for k, given that sub-array sizes 0 or 1 constitute terminal sub-calls, is $O(\log n)$. Moreover, the probability that a given pivot candidate proves suitable is at least $1/2$. Consequently, the expected amount of work to produce a suitable pivot is at most twice the size sub-array being acted upon by a sub-call – reflecting the expected two (at most) candidate trials to produce a suitable pivot. Now for a given fixed k, the sub-array ranges corresponding to the sub-calls ν belonging to level-k are non-overlapping, so that the sum of these sub-array sizes is bounded by n. Define $E(\nu)$ to be expectation of work in a sub-call ν, and C_k to be the expectation of total work among the level-k sub-calls. Then it follows that

$$C_k = \sum_{\nu \in \text{level-}k} E(\nu) \leqslant \sum_{\nu \in \text{level-}k} 2 \cdot \text{sub-array-size}(\nu) \leqslant 2n.$$

Multiplying by the $O(\log n)$ bound on the number of levels completes the derivation of the $O(n \log n)$ bound for the expected cost of Insistent-Q-Sort. To be clear, there is no practical purpose served in implementing Insistent-Q-Sort. We proceed to illustrate, however, that it provides a frame of reference for estimating Quicksort.

It is tempting to anticipate that a bound for Insistent-Q-Sort directly applies to Quicksort, particularly since Insistent-Q-Sort pays the price for, but eschews the progress available from bad pivot candidates. But one cannot a priori dismiss the possibility that a bad pivot sets Quicksort off in a bad (costly) direction that Insistent-Q-Sort avoids. The argument below essentially groups the partitioning steps of Quicksort in a manner making clear that no harm is done, relative to Insistent-Q-Sort, when partitioning with bad pivots.

Let T be the binary recursion tree reflecting a randomized execution of Quicksort. We color the nodes of T, red or blue, as follows. The root of T is colored red, and any node acting on a sub-array having size at most $\frac{3}{4}$ of the size of the sub-array of its parent node is also colored red. All other nodes (having sub-array size exceeding $\frac{3}{4}$ of the size of the parent's sub-array) are colored blue. For any red node in T define its *red*-level to be the number of red proper ancestors it has. The root has red-level 0. Observe that the sub-array acted upon by red node ω with red-level k, $k > 0$, has size at most $\frac{3}{4}$ of the size of the sub-array of the red node ancestor τ of ω with red-level $k-l$, since τ is either the parent of ω, or an ancestor of this parent. The size of the sub-array acted upon by a red node is therefore bounded by $(\frac{3}{4})^\ell \cdot n$ where ℓ is its red-level number.

The following can now be observed.

(a) There are only $O(\log n)$ red-levels.
(b) For any fixed integer k, the sub-arrays associated with the red nodes having red-level k have non-overlapping ranges (since for any pair of such red nodes, one cannot be an ancestor of the other).
(c) Any given node ν in T has at most one blue child. Moreover, the probability that ν has a blue child is at most $\frac{1}{2}$, since a balanced partitioning precludes the presence of a blue child.

Considering (c), we observe that the blue nodes in T form linear sub-paths, and that there can be only one such blue sub-path immediately descending from any red node. Now for any non-terminal red node ν, attribute to ν the totality of the partitioning costs incurred in the sub-call at ν, as well as in the sub-calls represented at the nodes of the (possibly empty) blue sub-path descending from ν. All of the cost encompassed by T is now accounted for by its red nodes, and a red node acting on a sub-array of size m has an expectation of attributed cost bounded by $2m$. This $2m$ bound follows from the intuitive assertion that the expected number of partitioning steps, terminating when a balanced partition occurs, is at most 2, but can be argued formally as follows.

> The conditional probability that a node has a blue child, conditioning relative to its ancestral sub-calls, never exceeds $1/2$. It follows that the probability p_j that its descending blue sub-path has j or more nodes is bounded by 2^{-j}. Summing the p_j gives the expected length of this blue sub-path, yielding a bound of 1. Summing the (at most) m cost contributions from the red node itself, as well as each blue node on the descending blue sub-path, yields the bound $2m$ on the expectation of attributed cost.

Now in parallel with the above analysis of Insistent-Q-Sort, it follows by (b) that the expected total cost attributed to the red nodes at any fixed red-level is bounded by $2n$, so that by (a) the total expected cost of Quicksort is $O(n \log n)$.

The above argument can be recast so that Quicksort is viewed as a systematic modification of Insistent-Q-Sort, to which the same time bound must necessarily apply. Starting with Insistent-Q-Sort, acting on an array A of size n, should a given candidate pivot be unsuitable, resulting in an unbalanced partition of A, proceed with the partitioning *anyway*, but then immediately attend to the larger sub-array. Repeat, partitioning the larger sub-array, and continue iteratively until a balanced partitioning occurs. As the iteration proceeds, append each of the smaller sub-array portions resulting from the unbalanced partitionings to a list Γ of sub-arrays, and also append to Γ both portions resulting from the final balanced partition. The overall effect of this iterative process is a partitioning of the array A into a multitude Γ of non-overlapping sub-arrays, each having size at most a $\frac{3}{4}$ fraction of the size of A. Moreover, the expected number of partitioning steps of the iteration is bounded by 2, including the initial partitioning of A; the

same as the corresponding expected number of trials that Insistent-Q-Sort performs to produce a suitable candidate pivot. Thus the same $2n$ bound applies with respect to the expected cost of an iteration. Our modified algorithm now proceeds by recursively processing each of the sub-arrays of A belonging to Γ. The resulting algorithm necessarily satisfies the same time bound as Insistent-Q-Sort. However, it is plain that this modification of Insistent-Q-Sort is none other than Quicksort, apart from a reordering of the sub-array partitioning steps.

2. Remarks

Abandoning the assumption of distinct data values, consider Quicksort implemented with linear-time 3-way partitioning – three sub-arrays are formed by a partitioning step; one consisting of all values equal to the pivot, and with the other two *active* sub-arrays being acted upon by recursive sub-calls. Then the probability that the two active sub-arrays each have size not exceeding $\frac{3}{4}$ that of the parent sub-array continues to be at least $1/2$, so that the of $O(n \log n)$ cost expectation follows, without need of further discussion.

We also note that Hoare's randomized selection algorithm [3] analogous to Quicksort can be similarly treated to demonstrate a linear cost expectation.

We close with a fanciful observation. Suppose that the random source available for Quicksort is frequently under the control of an adversary, e.g. that for each m, up to, but not more than, $0.99m$ of the first m calls to the source yield malicious values. The remaining source calls yield legitimately random values, though their *timing*, in relation to the bogus-filled calls, is controlled by the adversary, aside from the limitation imposed by the 0.99 constraint. Standard Quicksort cannot weather this situation and no longer retains an $O(n \log n)$ expectation of cost. The above analysis, however, can be readily extended to a modification of Quicksort that preserves the $O(n \log n)$ expectation of cost. For this modification insert the sub-arrays resulting from each partitioning step – more precisely, the respective ranges that reference them – into a priority queue keyed by sub-array sizes, and draw the next sub-array to be partitioned by a call to extract-max. The priority queue can be implemented so that the aggregate cost of the operations invoked upon it is $O(n)$, seen as follows. Insertion of a sub-array having size m into the priority queue is accomplished by placing it at the front of the list of sub-arrays located at position $S[m]$, where S is an array of lists indexed by the key space $[1, n]$ of possible sub-array sizes. The successive returns from extract-max are monotone decreasing. So when the current list, $S[\ell]$ (say), which supplied the preceding extract-max request becomes empty, then the next block of sub-arrays for supplying subsequent extract-max requests is obtained from the next non-empty list in S, found by repeatedly decrementing ℓ. (We make no pretense that this observation has any practical import. Nor should it be viewed as a contribution to the quite distinct and rather developed theory concerning the qualities of random sources necessary for effective implementation of randomized algorithms.)

Acknowledgement

The author appreciates the reviewers' observations that the preceding analysis should extend to the case of duplicate data-values, as well as suggestions for clarifying and improving the exposition.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, Cambridge, 2009.
[2] C.A.R. Hoare, Quicksort, Comput. J. 5 (1) (1962) 10–15.
[3] C.A.R. Hoare, Algorithm 63 (PARTITION) and Algorithm 65 (FIND), Commun. ACM 4 (7) (1962) 321–322.
[4] D.E. Knuth, The Art of Computer Programming: Sorting and Searching, second ed., Addison Wesley Longman, Reading, 1998.