Alcom-FT Technical Report Series
ALCOMFT-TR-03-50

Partial Quicksort *

Conrado Martínez†

Abstract

This short note considers the following common problem: given an array with n elements, rearrange it so that its first m components contain the m smallest elements in the array in ascending order. We propose here a simple variation of quicksort that efficiently solves the problem, and show and quantify how it outperforms other straightforward alternatives.

1 Introduction

In many applications, what we need is to obtain a sorted list of the smallest m elements of a set of n given elements. This is known as *partial sorting*. Sorting the whole array is an obvious solution, but it clearly does more work than necessary.

A usual solution to this problem is to make a heap with the n given elements (in linear time) and then perform m successive extractions of the minimum element. Historically, this has been the way to implement C++ STL's `partial_sort` function [5]. Its most prominent feature is that it guarantees $\Theta(n + m \log n)$ worst-case performance.

Another solution, begins building a max-heap with the first m elements of the given array, then scans the remaining $n-m$ elements and updates the heap as necessary, so that at any given moment the heap contains the m smallest elements seen so far. Finally, the heap is sorted. Its worst-case cost is $\Theta((m + n) \log m)$ and it is not an interesting alternative unless m is quite small or we had to process the input on-line.

Last but not least, we can solve the problem by first using a selection algorithm to find the m-th smallest element. Most selection algorithms also rearrange the array so that the elements which are smaller than the sought element are to its left, the sought element is at the m-th component, and the elements which are larger are to the right. Hence, we finish just sorting the subarray to the left of the m-th element. Using efficient algorithms for both tasks (selection and sort) the total cost is $\Theta(n + m \log m)$. The obvious choice is to use Hoare's

*The research of the author was supported by the Future and Emergent Technologies programme of the EU under contract IST-1999-14186 (ALCOM-FT) and the Spanish Min. of Science and Technology project TIC2002-00190 (AEDRI II).

†Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain. E-mail: `conrado@lsi.upc.es`.

quickselect and quicksort algorithms [3, 4]; the cost stated above is then only guaranteed on average, but in practice this combination should outperform most other choices. The Copenhagen STL group implements `partial_sort` that way—actually, using finely tuned, highly optimized variants of these algorithms (`http://www.cphstl.dk`). For convenience, we call this combination of the two algorithms *quickselsort*.

In this paper we propose *partial quicksort*, a simple and elegant variant of quicksort that solves the partial sorting problem, by combining selection and sorting into a single algorithm. To the best of the author's knowledge the algorithm hadn't been formally proposed before, but because of its simplicity, it probably has been around for many years. Partial quicksort has the same asymptotic average complexity as quickselsort, namely, $\Theta(n + m \log m)$, but does less work. In particular, if we consider the standard variants of quickselsort and partial quicksort, the latter saves $2m - 4 \ln m + \mathcal{O}(1)$ comparisons and $m/3 - 5/6 \ln m + \mathcal{O}(1)$ exchanges.

The rest of this short note is devoted to present the algorithm and to the analysis of the average number of comparisons and exchanges of the basic variant and to compare them with the corresponding values for the quickselsort algorithm.

2 The algorithm

Partial quicksort (see Algorithm 1) works as follows: if the array contains one or no elements, then we are done. Otherwise, one of the elements is chosen as the pivot, say p, and the array $A[i..j]$ is partitioned so that $A[i..k-1]$ contains the elements that are smaller than p, $A[k]$ contains the pivot and $A[k+1..j]$ contains the elements which are larger than the pivot[1]. We will assume that exactly $n-1$ element comparisons are necessary to carry out the partitioning of the array and that partitioning preserves randomness [16]. Now, if $k \geq m$ then all the elements that we want are in the left subarray, so we only need to make a recursive call on that subarray. Otherwise, if $k < m$ then we have to call the function on both subarrays: notice that the function will behave exactly as quicksort in the left subarray and sort those elements, whereas the recursive call on the right subarray has still the job to put the m-th element into its final place.

Algorithm 1 depicts the basic or standard variant of the algorithm. Usual optimizations like recursion cutoff, using samples to select the pivots, recursion removal, optimized partition loops [1, 14, 15, 16], etc. can be applied here and presumably yield benefits similar to those for the quicksort and quickselect algorithms. However, we do not analyze these refined variants in this paper and stick to the simpler case.

[1]We assume for simplicity that all the elements in the array are distinct. Only minor modifications are necessary to cope with duplicate elements.

Algorithm 1 Partial quicksort

```
void partial_quicksort(vector<Elem>& A,
                       int i, int j, int m) {
   if (i < j) {
      int pidx = select_pivot(A, i, j);
      int k;
      partition(A, pidx, i, j, k);
      // A[i..k-1] <= A[k] <= A[k+1..j]
      partial_quicksort(A, i, k - 1, m);
      if (k < m) // use "k < m - 1" for arrays starting at index 0
        partial_quicksort(A, k + 1, j, m);
   }
```

3 The average number of comparisons

Let $P_{n,m}$ denote the average number of (key) comparisons made by partial quicksort to sort the m smallest elements out of n. Let $\pi_{n,k}$ denote the probability that the chosen pivot is the k-th element among the n given elements. We assume, as it is usual in the analysis of comparison-based sorting algorithms, that any permutation of the given distinct n elements is equally likely. Then

$$P_{n,m} = n - 1 + \sum_{k=1}^{m} \pi_{n,k} \left(P_{k-1,k-1} + P_{n-k,m-k} \right) + \sum_{k=m+1}^{n} \pi_{n,k} P_{k-1,m}. \quad (1)$$

But, as we already have said, partial quicksort behaves exactly as quicksort when $m = n$, so that $P_{n,n} = q_n = 2(n+1)H_n - 4n$, where $H_n = \sum_{1 \le k \le n} 1/k = \ln n + \mathcal{O}(1)$ denotes the n-th harmonic number [9, 10, 14]. Let

$$t_{n,m} = n - 1 + \sum_{k=0}^{m-1} \pi_{n,k+1} q_k.$$

Hence, we get the recurrence

$$P_{n,m} = t_{n,m} + \sum_{k=1}^{m} \pi_{n,k} P_{n-k,m-k} + \sum_{k=m+1}^{n} \pi_{n,k} P_{k-1,m}, \quad (2)$$

which, except for the *toll function* $t_{n,m}$, has the same form as the recurrence for the average number of comparisons made by quickselect (see [7, 10]). The recurrence above holds for whatever pivot selection scheme we use: median-of-three [6, 18], median-of-$(2t+1)$ [13], proportional-from-s [12], pseudomedian-of-9 [1],.... However, as we have already pointed out, we will only consider the basic variant, hence, for the rest of this paper we take $\pi_{n,k} = 1/n$, for $1 \le k \le n$.

Our next step is to introduce the bivariate generating functions (BGFs) associated to the quantities $P_{n,m}$ and $t_{n,m}$. Our path to the sought answer (a closed formula for $P_{n,m}$) is fairly standard, but for readers who are not familiar with the use of generating functions in the analysis of algorithms, Sedgewick and Flajolet's book [17] and Knuth's *The Art of Computer Programming* [8, 9] are excellent starting points which describe in great detail the techniques that we will apply here.

Once the BGFs

$$P(z,u) = \sum_{n\geq 0} \sum_{1\leq m\leq n} P_{n,m} z^n u^m,$$
$$T(z,u) = \sum_{n\geq 0} \sum_{1\leq m\leq n} t_{n,m} z^n u^m,$$

are defined, we can translate Equation (2) into a functional relation over BGFs, namely,

$$\frac{\partial P}{\partial z} = \frac{P(z,u)}{1-z} + \frac{u}{1-uz} P(z,u) + \frac{\partial T}{\partial z}, \tag{3}$$

whose solution is (see for instance [11])

$$P(z,u) = \frac{1}{(1-z)(1-uz)} \left\{ \int (1-z)(1-uz) \frac{\partial T}{\partial z}\, dz + K \right\},$$

subject to $P(0,u) = 0$, as $P_{0,m} = 0$ for any m.

Since $t_{n,m} = n - 1 + \frac{1}{n} \sum_{0\leq k<m} q_k$, we can decompose $P(z,u)$ as $P(z,u) = F(z,u) + S(z,u)$, where the first BGF accounts for the selection and the second one for the sorting. That is, with $T(z,u) = T_F(z,u) + T_S(z,u)$, we have

$$T_F(z,u) = \sum_{n\geq 0} \sum_{1\leq m\leq n} (n-1) z^n u^m,$$
$$T_S(z,u) = \sum_{n\geq 0} \sum_{1\leq m\leq n} \left(\frac{1}{n} \sum_{0\leq k<m} q_k \right) z^n u^m,$$
$$F(z,u) = \frac{1}{(1-z)(1-uz)} \left\{ \int (1-z)(1-uz) \frac{\partial T_F}{\partial z}\, dz + K_F \right\},$$
$$S(z,u) = \frac{1}{(1-z)(1-uz)} \left\{ \int (1-z)(1-uz) \frac{\partial T_S}{\partial z}\, dz + K_S \right\}.$$

Also, from the combinatorics of the problem, we have $F(0,u) = 0$ and $S(0,u) = 0$. This means that $K_F = -4$ and therefore $F(z,u)$ is exactly the same BGF as the one for the average number of comparisons made by quickselect to find the m-th element out of n elements. Namely,

$$F(z,u) = \frac{4u-2}{(1-u)(1-z)(1-uz)} \ln \frac{1}{1-z} + \frac{2u-4}{(1-u)(1-z)(1-uz)} \ln \frac{1}{1-uz} + \frac{2}{(1-u)(1-z)^2(1-uz)} + \frac{2}{(1-u)(1-z)(1-uz)^2}.$$

Extracting coefficients we get the well-known result [7]

$$F_{n,m} = [z^n u^m]F(z,u) = 2\big(n+3+(n+1)H_n - (n+3-m)H_{n+1-m} \\ - (m+2)H_m\big), \qquad \text{if } 1 \le m \le n. \quad (4)$$

On the other hand,

$$\begin{aligned}\frac{\partial T_S}{\partial z} &= \sum_{n>0} z^{n-1} \sum_{0\le m<n} u^{m+1} \sum_{k=0}^{m} q_k \\ &= u \sum_{m\ge 0} u^m \sum_{k=0}^{m} q_k \sum_{n\ge m} z^n = \frac{u}{1-z} \sum_{m\ge 0} (uz)^m \sum_{k=0}^{m} q_k \\ &= \frac{u}{1-z}\frac{Q(uz)}{1-uz},\end{aligned}$$

with $Q(z) = \sum_{n\ge 0} q_n z^n$. It is not difficult to show that

$$Q(z) = \frac{2}{(1-z)^2}\left(\ln\frac{1}{1-z} - z\right),$$

and thus

$$\begin{aligned}S(z,u) &= \frac{1}{(1-z)(1-uz)}\left\{\int u\,Q(uz)\,dz + K_S\right\} \\ &= \frac{2}{(1-uz)^2(1-z)}\ln\frac{1}{1-uz} + \frac{2}{(1-z)(1-uz)}\ln\frac{1}{1-uz} \\ &\quad - \frac{4}{(1-uz)^2(1-z)} + \frac{4}{(1-uz)(1-z)},\end{aligned}$$

since $K_S = 4$.

Extracting coefficients from $S(z,u)$ is straighforward from

$$[z^n u^m]\frac{1}{(1-uz)^2(1-z)}\ln\frac{1}{1-uz} = \begin{cases}(m+1)H_m - m, & \text{if } 1\le m\le n,\\ 0, & \text{otherwise.}\end{cases}$$

$$[z^n u^m]\frac{1}{(1-uz)(1-z)}\ln\frac{1}{1-uz} = \begin{cases}H_m, & \text{if } 1\le m\le n,\\ 0, & \text{otherwise.}\end{cases}$$

We have then

$$[z^n u^m]S(z,u) = 2(m+1)H_m - 6m + 2H_m, \quad (5)$$

whenever $1 \le m \le n$. Finally, adding the two contributions (4) and (5) we get

$$P_{n,m} = 2n + 2(n+1)H_n - 2(n+3-m)H_{n+1-m} - 6m + 6, \quad (6)$$

if $1 \le m \le n$. As a further check, the reader can easily verify that $P_{n,n} = q_n$.

Now we can compare the average number of comparisons of partial quicksort $P_{n,m}$ with that of quickselsort, that is, $F_{n,m} + q_{m-1}$. And it turns out that partial quicksort makes

$$2m - 4H_m + 2$$

comparisons less than its alternative; this is probably irrelevant for small m, but significative enough when m is large, say $m = \Omega(\sqrt{n})$.

4 The average number of exchanges

In order to investigate the average number of exchanges and other quantities of interest, we generalize our analysis in the previous section to cope with toll functions of the form

$$t_{n,m} = an + b + \sum_{k=0}^{m-1} \pi_{n,k} q'_k,$$

where a and b are arbitary constants and q'_n is the solution of

$$\begin{aligned} q'_n &= an + b + 2 \sum_{0 \le k < n} \pi_{n,k} q'_k, \\ q'_0 &= 0. \end{aligned}$$

If we wish to compare partial quicksort with quickselsort, it is now quite clear that we have only to compute the coefficients of $S(z, u)$, but now we have to use the generating function of the sequence $\{q'_n\}_{n \ge 0}$:

$$Q(z) = \frac{1}{(1-z)^2} \left(2a \ln \frac{1}{1-z} + (b-a)z \right).$$

Hence,

$$\begin{aligned} S(z,u) &= \frac{1}{(1-z)(1-uz)} \left\{ \int u\, Q(uz)\, dz + K_S \right\} \\ &= \frac{2a}{(1-uz)^2(1-z)} \ln \frac{1}{1-uz} + \frac{a-b}{(1-z)(1-uz)} \ln \frac{1}{1-uz} \\ &\quad - \frac{b-3a}{(1-uz)^2(1-z)} + \frac{3a-b}{(1-uz)(1-z)}, \end{aligned}$$

since $K_S = 3a - b$.

The quantity of interest is then

$$[z^n u^m] S(z,u) = 2amH_m + (3a-b)H_m + (b-5a)m, \qquad \text{if } 1 \le m \le n,$$

and if we compare q'_{m-1} with the result above, we get that partial quicksort makes

$$2am + (b-3a)H_m + a - b \tag{7}$$

operations less than quickselsort. For instance, for the partitioning method given in [14], the average number of exchanges made during a single partitioning stage is $n/6 - 1/3$. Substituting these values ($a = 1/6$, $b = -1/3$) in (7), we find that we save $\frac{1}{3}m - \frac{5}{6}H_m + \frac{1}{2}$ exchanges by using partial quicksort.

5 Conclusions

In this short research note we have conducted a preliminary analysis of partial quicksort which exhibits its competitive advantage over common, straitghforward alternatives for the partial sort problem. It also shows that partial quicksort can be a rich source of new and appealing mathematical problems. Sections 3 and 4 are witness of the power of the standard and by now classic tools of the analysis of algorithms.

Partial quicksort can be subject to standard optimization techniques like sampling, recursion removal, recursion cutoff on small subfiles, and so on. Also, the idea behind partial quicksort applies to similar algorithms like radix sort [9] or quicksort for strings [2].

In the author's opinion, a challenging open problem on partial quicksort is to find a combinatorial, intuitive explanation for the difference between partial quicksort and quickselsort. It is not self-evident that partial quicksort does significantly ($\Theta(m)$) less work on the average than quickselsort, even though it is more or less clear that partial quicksort would not do more work.

Partial quicksort sorts the m elements incrementally, in chunks, while it looks for the m-th element, whereas quickselsort makes its initial recursive call to quicksort on the chunk of $m - 1$ elements smaller than the m-th. So it seems that by "breaking" the sorting of the $m - 1$ smallest elements in the way partial quicksort does, it makes bad partitions at early stages more unlikely and thus reduces somewhat the average complexity.

References

[1] J.L. Bentley and M.D. McIlroy. Engineering a sort function. *Software—Practice and Experience*, 23:1249–1265, 1993.

[2] J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 360–369, 1997.

[3] C.A.R. Hoare. FIND (Algorithm 65). *Comm. ACM*, 4:321–322, 1961.

[4] C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[5] N.M. Josuttis. *The C++ Standard Library: A Tutorial and Reference Guide*. Addison-Wesley, 1999.

[6] P. Kirschenhofer, H. Prodinger, and C. Martínez. Analysis of Hoare's FIND algorithm with median-of-three partition. *Random Structures & Algorithms*, 10(1):143–156, 1997.

[7] D.E. Knuth. Mathematical analysis of algorithms. In *Information Processing '71, Proc. of the 1971 IFIP Congress*, pages 19–27, Amsterdam, 1972. North-Holland.

[8] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[9] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 2nd edition, 1998.

[10] H.M. Mahmoud. *Sorting: A Distribution Theory*. John Wiley & Sons, New York, 2000.

[11] C. Martínez, D. Panario, and A. Viola. Analysis of quickfind with small subfiles. In B. Chauvin, Ph. Flajolet, D. Gardy, and A. Mokkadem, editors, *Proc. of the 2nd Col. on Mathematics and Computer Science: Algorithms, Trees, Combinatorics and Probabilities*, Trends in Mathematics, pages 329–340. Birkhäuser Verlag, 2002.

[12] C. Martínez, D. Panario, and A. Viola. Adaptive sampling for quickselect. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004. Accepted for publication.

[13] C. Martínez and S. Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM J. Comput.*, 31(3):683–705, 2001.

[14] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1976.

[15] R. Sedgewick. Implementing quicksort programs. *Comm. ACM*, 21:847–856, 1978.

[16] R. Sedgewick. *Quicksort*. Garland, New York, 1978.

[17] R. Sedgewick and Ph. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, Mass., 1996.

[18] R.C. Singleton. Algorithm 347: An efficient algorithm for sorting with minimal storage. *Comm. ACM*, 12:185–187, 1969.