A Noble Randomized Quicksort with a Random Number Generator

Md. Atiqur Rahman Ahad
Department of Applied Physics, Electronics & Communication Engineering,
University of Dhaka, Bangladesh
Kyushu Institute of Technology, Japan
E-mail: atiqahad@univdhaka.edu

Abstract
Sorting is a well-known computational problem. Sorting means arranging a set of records (or a list of keys) in some (increasing or decreasing) order. Randomized algorithms often have low complexity and are associated to robustness bounds, which are generally less conservative than the classical ones. This paper presents a noble randomized Quicksort with a random number generator. This randomized Quicksort can solve the problems in a better way when permutations of the input numbers are not equally likely. The detailed algorithm is presented here with explanation.
Keyword: Quicksort, randomized Quicksort, algorithm.
1. Introduction

Sorting is an important kernel for sequential and multiprocessing computing and a core part of database systems. Donald Knuth [1] reports that “computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spend on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.” As it was expected, sorting is one of the most heavily studied problems in computer science [4]. Of them Quicksort algorithm is probably the most widely used sorting method in computer science [6,7,11] and is one of the most widely used internal sort routines [10]. It is widely believed, the fastest comparison-based, sequential sorting algorithm [15,16] for a large input sets on average.

Quicksort is a sorting algorithm whose worst-case running time is \(\Theta(n^2)\) on an input array of \(n\) numbers [2,5]. In spite of this slow worst-case running time, Quicksort is often the best practical choice for sorting because it is remarkably efficient on the average-case, its expected running time is \(\Theta(n \log n)\), and the constant factors hidden in the \(\Theta(n \log n)\) notation are quite small [1]. It also has the advantage of sorting-in-place, and it works well even in virtual memory environments.

2. Background

There are several variants of quicksort and of them the following variants are well-known [21]. Balanced quicksort choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm. Whereas, the external quicksort is similar to the regular quicksort except the pivot is replaced by a buffer. In this case, initially, read the \(M/2\) first and last elements into the buffer and sort them. Afterwards, read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition [21]. Another well-known variant is called three-way radix quicksort or multikey quicksort, which is a combination of radix sort and quicksort. Note that radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. In the multi-key quicksort method, we pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Then we partition the remaining elements into three sets, namely, those whose corresponding character is less than, equal to, and greater than the pivot’s character. Recursively sort the less than and greater than partitions on the same character. Recursively sort the equal to partition by the next character (key).

In exploring the average-case behavior of Quicksort, it has been assumed that all permutations of the input numbers are equally likely [2]. In an engineering situation, however, we cannot always expect it to hold. We can sometimes add randomization to an algorithm in order to obtain good average-case performance over all inputs [8]. In various situations where deterministic algorithms’ scalability is poor, randomized versions of the same algorithms are easier to implement and provide surprisingly good performance [17,18] and stability [19]. Randomized algorithms often have low complexity and are associated to robustness bounds which are generally less conservative than the classical ones [20,23]. Many people regard the resulting randomized version of Quicksort as the sorting algorithm of choice for large enough inputs. In order to use probabilistic analysis, we need to know something about the distribution on the inputs. In many cases, we know very little about the input distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally. Yet we often can use probability and randomness as a tool for algorithm design and analysis, by making the behavior of part of the algorithm random. More generally, we call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator. Knowing a distribution on the inputs can help to analyze the average-case behavior of an algorithm. Many times, we do not have such knowledge and no average-case analysis is possible and therefore, use of a randomized algorithm can solve the problem. For this algorithm and many other randomized algorithms, no particular input elicits its worst-case behavior, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if random-number generator produces an “unlucky” permutation [2].

3. Randomized Quicksort Algorithm

The choice of pivot elements is the determining factor for the runtime of Quicksort [11]. In reality, there are three basic methods to select the pivot [6]. The first and simplest is to select an element from a fixed position of \(A\), typically the first, as the pivot [12], or the middle element [1]. The second method for selecting the pivot is to try to approximate the median of \(A\) by computing the median of a small subset of \(A\), or the median of three [11,13]. Finally, we can randomly select an element from \(A\) to be the pivot by using a random-number generator [14]. In this paper, the third choice is considered. For the
implementation of Randomized Quicksort, one has to choose the pivot an element \(A[i]\), \(p \leq i \leq q\), in the sub-array \(A[p...q]\) at random. For the case, where one do not have a random number generator which, given \(p\) and \(q\), produces an \(i\) such that \(p \leq i \leq q\), with all numbers \(i\) between \(p\) and \(q\), picked with equal probability (thus with the probability \(1/(q-p+1)\), because there are \(q-p+1\) such numbers). Also assume that we have a random number generator producing \(1\) or \(0\) with equal probability (i.e., one half). Using such random number generator, we will design our own random number generator which for a given \(p\) and \(q\) – chooses a number \(i\), such that \(p \leq i \leq q\), and all numbers in that range are nearly equally probable, i.e., have probability \((1/(q-p+1)) \pm \epsilon\), where \(\epsilon\) is an arbitrary small and fixed number independent of \(p\) and \(q\).

Let the number of bits, i.e., the number of times to run the randomGenerator( ) to produce \(0\) or \(1\) each time is, \(\log(q-p+1)\). Here the ceiling has been considered to make the number of choices as integer so that ( \(2^{\text{integer}}\) ) it starts with all \(0\)'s and all \(1\)'s. Therefore, the number of choices \((\text{nc})\) can be defined as,

\[
\text{nc} = 2^\text{numberOfBits} = 2^{\log(q-p+1)}
\]

Therefore, the portion with proper elements, i.e., the using part (up) is \(q-p+1\). This is the number of elements that has been used up so far. And the not using parts from the total number of choices \((\text{nu}) = \text{nu} = \text{nc} - \text{up} \). Therefore,\n
\[
\therefore \text{nu} = 2^{\log(q-p+1)} - (q-p+1)
\]

3.1 Generation of ‘nc’

Let the using part (up), i.e., the number of elements in the array is six (6). So the number of choices \((\text{nc})\) will be \(2^{\log(6)}\); i.e., 8 (hence 3 bits). So three randomGenerator( ) will produce from 000 to 111 in binary and we have to convert it into decimal. Fig. 1 demonstrates this concept. In this case, the generation number of choices \((\text{nc})\) continues from 000 to 111 according to 000, 001, 010, 011, 100, 101, 110, and 111.

<table>
<thead>
<tr>
<th>up</th>
<th>nu</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
</tr>
</tbody>
</table>
| 111| ...

Figure 1: Generation of number of choices \((\text{nc})\) – from 000 to 111.

Then compare the GeneratedValue with the array index. If it is within up, then take that element as the PIVOT. Otherwise, generate other GeneratedValue with the probability \((1/(q-p+1)) \pm \epsilon\) – where the \(\epsilon\) will be provided as an input in the program.

3.2 Probability to ReRoll to Generate GeneratedValue

The probability to ReRoll in order to generate the new value is the probability to get ‘not using part’ (\(\text{nu}\)). We can define it as below:

\[
\text{Pr(ReRoll)} = 1 - \frac{(q-p+1)}{2^{\log(q-p+1)}}
\]

Let, \((q-p+1) = n\). Therefore, this probability to ReRoll \((P_p)\) becomes,

\[
P_p = 1 - \frac{n}{2^{\log(n)}}
\]

Now, we will define the number of times to ReRoll. If we ReRoll once, then nothing to do. But if we ReRoll twice, then the probability becomes squared as \((P_p)^2\); if thrice, then the probability becomes \((P_p)^3\); and so on. Finally, for \(k\) times ReRoll – we can get that the probability is \((P_p)^k\).

Therefore, based on our previous discussion, we can conclude that,

\[
(P_p)^k \leq \epsilon \implies k \log(P_p) \leq \log(\epsilon)
\]

So, the random generator part of the algorithm will rotate ‘\(k\)’ times based on the given value of \(\epsilon\). This ensures the not using \((\text{nu})\) part to hit smaller than \(\epsilon\). So, for an array, the probability of stopping on a proper choice will be,

\[
1 - 1/(2^{\log(n)})^k \geq 1 - \epsilon
\]

For an array with 3 elements (say, 5, 10,15), we get,

<table>
<thead>
<tr>
<th>up</th>
<th>nu</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
| 15 | ...
| 000| 001|
| 010| 011|
| 111| ... |

Let, the probability distribution of the entire algorithm is (100%).

Assume that after one generation of random number, the choice was not proper. So, we have to ReRoll again and assume that this time, it is not in the proper choice. Hence, the probability distribution is,
Afterwards, ReRoll once again and assume that even this time, it is not in the proper choice. Hence the probability distribution goes to narrower as,

<table>
<thead>
<tr>
<th>10</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

If this continues, then it will be smaller slice, and the distribution narrow-downs unless the condition ($\leq \epsilon$) satisfies. The above discussed algorithm is stated below:

**RandomNumberGenerator** ($p, q, \epsilon$)

if ($q - p + 1 = 1$), then return 1;
if ($p < q$) then
// Generate the number of bits
NoOfBits $\leftarrow \left\lceil \log(q - p + 1) \right\rceil$
NoOfChoice $\leftarrow 2^{\text{NoOfBits}}$
// If no 'unused part', then
if ($\text{NoOfChoices} = (q - p + 1)$) then
   i $\leftarrow \text{randomGenerator}(\text{NoOfBits})$
   return i;

// There are some 'unused part', so need to re-roll.
else
   // Compute Pr(ReRoll) $= p_r$
   $p_r \leftarrow (1 - (q - p + 1) / \text{NoOfChoices})$
   // Compute rotation invariant 'k'
   $k \geq (\log \epsilon) / \log (p_r)$
   for $j \leftarrow 1$ to $k$
      i $\leftarrow \text{randomGenerator}(\text{NoOfBits})$
      // If the 'i' is within the proper choice
      if ($i \leq (q - p + 1)$) then
         return i;
      else
         // Increase the counter
         $k \leftarrow k + 1$;

The random generator will work according to the following pseudocode:

**randomGenerator**(NoOfBits)

// Return the index
for i $\leftarrow 0$ to (NoOfBits $-$ 1) do
   Binary[i] $\leftarrow \text{random}()$;
//random() produces 0 or 1 and put them in Binary[ ] array
// Convert this to decimal using some method
   index $\leftarrow \text{Binary2Decimal}(\text{Binary}[])$;
   return index;

4. **Randomized Version of the Quicksort**

In Quicksort, if there are some elements equal to each other, i.e., there is number of repetitions in the array of the elements (for example, 7 2 3 5 2 7 2 3 2 5 9), then the basic Quicksort algorithm’s running time is very high and in the worst case scenario, the overhead becomes more. So to mitigate this problem, instead of taking the last element of the array as the PIVOT, we can take any element within the array for the PIVOT, randomly employing some random number generator. Below the algorithm is given. Here the Quicksort() is the conventional function that are used for the basic Quicksort algorithm.

**RandomizedPartition** ($A, p, r, \epsilon$)

x $\leftarrow \text{RandomNumberGenerator}(p, r, \epsilon)$ // PIVOT
   // This extra work produces a pivot randomly.
   // The rest of the program will be just like the basic Quicksort algorithm. Therefore, these are not covered in this paper.

**RandomizedQuickSort** ($A, p, r, \epsilon$)

if $p < r$
   q $\leftarrow \text{RandomizedPartition}(A, p, r, \epsilon)$
   QuickSort($A$, $p$, $q-1$)
   QuickSort($A$, $q+1$, $r$)

Randomized quicksort has the desirable property that, for any input, it requires only $\Theta(n \log n)$ expected time (averaged over all choices of pivots). Even if pivots aren’t chosen randomly, quicksort still requires only $\Theta(n \log n)$ time averaged over all possible permutations of its input. Because this average is simply the sum of the times over all permutations of the input divided by $n$ factorial, it’s equivalent to choosing a random permutation of the input. When we do this, the pivot choices are essentially random, leading to an algorithm with the same running time as randomized quicksort [21]. Quicksort has a space complexity of $\Theta(\log n)$, even in the worst case, when it is carefully implemented ensuring that (i) in-place partitioning is used. This requires $\Theta(1)$ space. And (ii) After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $\Theta(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn’t add to the call stack [21-22].

5. **Conclusions**

Quicksort remains the sorting routine of choice except when we have more detailed information about the input, in which case other algorithms could outperform Quicksort [6]. The divide-and-conquer strategy has three phases. First, divide the problem into several sub-problems (typically two for sequential algorithms and more for parallel algorithms) of almost equal sizes. Second, solve independently the resulting sub-problems. Third, merge the solutions of the sub-problems into a solution for the original problem. This strategy efficiently depends on finding efficient procedures to partition the problem during the initial phase and to merge the solutions during the last phase. Two points,
namely, method for selecting the pivot and the method for partitioning the input once the pivot is selected – are crucial. This paper has presented this pivot selection when some elements have more frequency in the inputs. This randomized version of the Quicksort improves the running time in worst case scenario.

Though there are varieties of random number generation methods [3], this paper have not used any of them, rather introduced a new proposal that will fit well in Quicksort algorithm when randomized algorithm is necessary. This represents a fundamental approach for designing efficient combinatorial algorithms that will remain a source of inspiration for researchers for many years to come. In particular, the full potential of the technique in handling higher-dimensional problems in computational geometry and in developing parallel algorithms for combinatorial problems is yet to be fully exploited [6]. In the future, we can anticipate vigorous research progress along these directions.

Acknowledgement

The author especially thanks to Aleksandar Ignjatovic, University of New South Wales, Australia for his correction and guidance on this work.

References


