© 2017 IJSRSET | Volume 3 | Issue 3 | Print ISSN: 2395-1990 | Online ISSN : 2394-4099
Themed Section : Engineering and Technology

Different Cases of Quick Sort

Chahat Monga[1] and Richa[2]
[1]Guru Nanak College, Department of Computer Science and Applications, Ferozepur, Punjab, India
[2]Punjabi University, Department of Computer Science, Patiala, Punjab, India

ABSTRACT

Sorting algorithms have attracted a great deal of attention and study, as they have numerous applications to Mathematics, Computer Science and related fields. In this thesis, we first deal with the mathematical analysis of the Quick sort algorithm and its variants. Specifically, we study the time complexity of the algorithm and we provide a complete demonstration of the variance of the number of comparisons required, a known result but one whose detailed proof is not easy to read out of the literature. We also examine variants of Quick sort, where multiple pivots are chosen for the partitioning of the array. The rest of this work is dedicated to the analysis of finding the true order by further pair wise comparisons when a partial order compatible with the true order is given in advance. We discuss a number of cases where the partially ordered sets arise at random. To this end, we employ results from Graph and Information Theory. Finally, we obtain an alternative bound on the number of linear extensions when the partially ordered set arises from a random graph, and discuss the possible application of Shell sort in merging chains.
Keywords : Sorting, Pivot, Keys, Pointer

I. INTRODUCTION

Sorting an array of items is clearly a fundamental problem, directly linked to efficient searching with numerous applications. The problem is that given an array of keys, we want to rearrange these in non-decreasing order. Note that the order may be numerical, alphabetical or any other transitive relation defined on the keys. In this work, the analysis deals with numerical order, where the keys are decimal numbers and we particularly focus on Quick sort algorithm and variants of it. Quick sort was invented by C. A. R. Hoare. Here is the detailed definition.

Definition: The steps taken by the Quick sort algorithm are:

1. Choose an element from the array, called pivot.
2. Rearrange the array by comparing every element to the pivot, so all elements smaller than or equal to the pivot come before the pivot and all elements greater than or equal to the pivot come after the pivot.
3. Recursively apply steps 1 and 2 to the sub array of the elements smaller than or equal to the pivot and to the sub array of the elements greater than or equal to the pivot.

Note that the original problem is divided into smaller ones, with (initially) two sub arrays, the keys smaller than the pivot, and those bigger than it. Then recursively these are divided into smaller sub arrays by further pivoting, until we get trivially sorted sub arrays, which contain one or no elements. Given an array of n distinct keys A = {a1, a2, . . . , an} that we want to quick sort, with all the n! Permutations equally likely, the aim is to finding the unique permutation out of all the n! possible permutations, such that the keys are in increasing order. The essence of Quick sort is the partition operation, where by a series of pair wise comparisons, the pivot is brought to its final place, with smaller elements on its left and greater elements to the right. Elements equal to pivot can be on either or both sides.

As we shall see, there are numerous partitioning schemes, and while the details of them are not central to this thesis, we should describe the basic ideas. A straightforward and natural way uses two pointers – a left pointer, initially at the left end of the array and a right pointer, initially at the right end of the array. We pick the leftmost element of the array as pivot and the right pointer scans from the right end of the array for a key less than the pivot. If it finds such a key, the pivot is swapped with that key. Then, the left pointer is increased by one and starts its scan, searching for a key greater than 7 the pivot: if such a key is found, again the pivot is exchanged with it. When the pointers are crossed, the pivot by repeated exchanges will "float" to its final position and the keys which are on its left are smaller and keys on its right are greater. The data movement of this scheme is quite large, since the pivot is swapped with the other elements.

A different partitioning scheme, described in is the following. Two pointers i (the left pointer, initially 1) and j (the right pointer, initially n) are set and a key is arbitrarily chosen as pivot. The left pointer goes to the right until a key is found which is greater than the pivot. If one is found, its scan is stopped and the right pointer scans to the left until a key less than the pivot is found. If such a key is found, the right pointer stops and those two keys are exchanged. After the exchange, both pointers are stepped down one position and the lower one starts its scan. When pointers are crossed, i.e. when $i \geq j$, the final exchange places the pivot in its final position, completing the partitioning. The number of comparisons required to partition an array of n keys is at least $n - 1$ and the expected number of exchanges is $n/6 + 5/6n$.

A third partitioning routine, called Lomuto's partition, this involves exactly $n - 1$ comparisons, which is clearly best possible, but the downside is the increased number of exchanges. The expected number of key exchanges of this scheme is $(n-1)/2$.

We now consider the worst case and best case, analysis of Quick sort. Suppose we want to sort the following array, $\{a1 < a2 < . . . < an\}$ and we are very unlucky and our initial choice of pivot is the largest element an. Then of course we only divide and conquer in a rather trivial sense: every element is below the pivot, and it has taken us $n - 1$ comparisons with an to get here. Suppose we 8 now try again and are unlucky again, choosing $an-1$ as pivot this time. Again the algorithm performs $n - 2$ comparisons and we are left with everything less than $an-1$. If we keep being unlucky in our choices of pivot, and keep choosing the largest element of what is left, after i recursive calls the running time of the algorithm will be equal to
$(n - 1) + (n - 2) + . . . + (n - i)$ comparisons, so the overall number of comparisons made is
$1 + 2 + . . . + (n - 1) = n \cdot (n - 1)/ 2$.

Thus Quick sort needs quadratic time to sort already sorted or reverse-sorted arrays if the choice of pivots is unfortunate.

If instead we always made good choices, choosing each pivot to be roughly in the middle of the array we are considering at present, then in the first round we make $n - 1$ comparisons, then in the two sub arrays of size about $n/2$ we make about $n/2$ comparisons, then in each of the four sub arrays of size about $n/4$ we make $n/4$ comparisons, and so on. So we make about n comparisons in total in each round. The number of rounds will be roughly log2 (n) as we are splitting the arrays into roughly equally-sized sub arrays at each stage, and it will take log2 (n) recursions of this to get down to trivially sorted arrays.

Thus, in this good case we will need $O(n \log_2 n)$ comparisons. This is of course a rather informal argument, but does illustrate that the time complexity can be much smaller than the quadratic run-time in the worst case

II. RANDOM SELECTION OF PIVOT

The mathematical analysis of Quick sort is presented, under the assumption that the pivots are uniformly selected at random. Specifically, the major expected costs regarding the time complexity of the algorithm and the second moment are computed.

Expected number of comparisons: This discussion of lucky and unlucky choices of pivot suggests the idea of selecting the pivot at random, as randomisation often helps to improve running time in algorithms with bad worst-case, but good average-case complexity. For example, we could choose the pivots randomly for a discrete uniform distribution on the array we are looking at each stage. Recall that the uniform distribution on a finite set assigns equal probability to each element of it.

C_n is the random variable giving the number of comparisons in Quick sort of n distinct elements when all the n! Permutations of the keys are equi-probable. It is clear that for n = 0 or n = 1, C0 = C1 = 0 as there is nothing to sort. These are the initial or "seed" values of the recurrence relation for the number of comparisons, given in the following Lemma.

- The random number of comparisons C_n for the sorting of an array consisting of $n \geq 2$ keys, is given by

$$C_n = C_{U_n - 1} + C^{\star}_{n-U_n} + n - 1,$$

Where U_n follows the uniform distribution over the set {1, 2, . . . , n} and C^{*}_{n-Un} is identically distributed to C_{un-1} and independent of it conditional on U_n.

- The expected number an of comparisons for Quick sort with uniform

Selection of pivots is $a_n = 2(n + 1)H_n - 4n$.

- Suppose that a pivot is chosen independently and uniformly at random from an array of n keys, in which Quick sort is applied. Then, for any input, the expected number of comparisons made by Randomised Quick sort is $2n \log_e(n) + O(n)$.

Expected number of exchanges

Here we consider the number of exchanges or swaps performed by the algorithm, which is mentioned by Hoare as a relevant quantity. We assume that each swap has a fixed cost and as in the previous section, we assume that the keys are distinct and that all n! permutations are equally likely to be the input: this in particular implies that the pivot is chosen uniformly at random from the array.

We should specify the partitioning procedure. Assume that we have to sort n distinct keys, where their locations in the array are numbered from left to right by 1,2,3,......, n. Set two pointers i <- 1 and j <- n-1 and select the element at Location n as a pivot. First, compare the element at location 1 with the pivot. If this key is less than the pivot, increase i by one until an element greater than the pivot is found. If an element greater than the pivot is found, stop and compare the element at location n-1 with the pivot. If this key is greater than the pivot, then decrease j by one and compare the next element to the pivot. If an element less than the pivot is found, then the j pointer stops its scan and the keys that the two pointers refer are exchanged. Increase i by one, decrease j by one and in the same manner continue the Scanning of the array until i>= j. At the end of the partitioning operation, the pivot is placed in its final position k, where 1<= k <=n, and Quick sort is Recursively invoked to sort the sub array of k- 1 keys less than the pivot and the sub array of n- k keys greater than the pivot.

Note that the probability of a key being greater than the pivot is (n-k)/(n-1)

The number of keys which are greater than pivot, and were moved during Partition is

((n-k)/(n-1)).(k-1)

Therefore, considering also that pivots are uniformly chosen and noting that we have to count the final swap with the pivot at the end of partition operation, we obtain

$$\sum_{k=1}^{n} \frac{(n-k)(k-1)}{n(n-1)} + 1 = \frac{n}{6} + \frac{2}{3}.$$

III. DIVIDE AND CONQUER RECURRENCES

We have computed the mean and variance of the number of comparisons made by Quick sort that mainly contribute to its time complexity. Because of the simple structure of the algorithm (dividing into smaller sub problems) we can in fact approach many other related problems in the same spirit. Let F (n) denote the expected value of some random variable associated with randomised Quick sort and T(n) be the average value of the "toll function", which is the needed cost to divide the problem into two simpler sub problems. Then F (n) is equal to the contribution T(n), plus the measures required sort the resulting sub arrays of (i-1) and (n-i) elements, where the pivot i can be any key of the array with equal probability.

Thus, the recurrence relation is

$$F(n) = T(n) + \frac{1}{n}\sum_{i=1}^{n}(F(i-1) + F(n-i))$$
$$= T(n) + \frac{2}{n}\sum_{i=1}^{n} F(i-1).$$

This is the general type of recurrences arising in the analysis of Quick sort, which can be manipulated using the difference method or by generating functions.

Since an empty array or an one having a unique key is trivially solved, the initial values of the recurrence is

F(0) = F(1) = 0

IV. QUICK SORT ON TWO PIVOTS

Along the following lines, we present a variant of Quick sort, where 2 pivots are used for the partitioning of the array. Let a random permutation of the keys {1, 2,....., n} to be sorted, with all the n! Permutations equally likely and let their locations in the array be numbered from left to right by {1,2,......., n}. The keys at locations 1 and n are chosen as pivots and since all the n! Permutations are equally likely to be the input then all the (n/2) pairs are equi-probable to be selected as pivots. At the beginning, the pivots are compared each other and are swapped, if they are not in order. If elements i < j are selected as pivots, the array is partitioned into three sub arrays: one with (i-1) keys smaller than i, a sub array of (j- i-1) keys between two pivots and the part of (n- j) elements greater than j.

The algorithm then is recursively applied to each of these sub arrays. The number of comparisons during the first stage is

$$A_{n,2} = 1 + \big((i-1) + 2(j-i-1) + 2(n-j)\big)$$
$$= 2n - i - 2,$$

for i = 1,2,3,.......,n-1, and j = i + 1,......, n. Note that in the specific partitioning scheme, each element is compared once to i and elements greater than i are compared to j as well. The average number of comparisons for the partitioning of n distinct keys

$$\mathbb{E}(A_{n,2}) = \frac{1}{\binom{n}{2}} \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\Big(2n-i-2\Big) = \frac{2}{n(n-1)}\left(\frac{5}{6}n^3 - 2n^2 + \frac{7}{6}n\right)$$
$$= \frac{5n-7}{3}.$$

V. CONCLUSION AND FUTURE DIRECTIONS

The conclusions of the research and possible future directions are being discussed. In the first section, we consider the sorting of partially ordered sets and in the second section, the fast merging of chains.

- **Sorting partially ordered arrays:** Central to the analysis of the time complexity of sorting partially ordered sets, was the number of linear extensions, as a measure of the 'pre-sorted-ness' of the array. Recall that the quantity log2(n!) is the lower bound of comparisons needed to sort an array of n keys, with no prior information. In all cases of partially ordered sets, the constant 1=2 log2(e) appeared to the asymptotic number of comparisons. A future direction to research might be the sharpening of these results. For example, one might ask, what is the average number of key exchanges or the computation of exact expected costs. Generalising Albert–Frieze argument and using entropy arguments, a new result was the lower bound on the number of linear extensions of a random graph order. However, we have seen that it does not directly compete the bounds of Alon *et al.*, thus there is space for further improvement of this bound or to derivation of new sharper ones and this might be a suitable topic for further research.

- **Merging chains using Shellsort:** Here, we discuss some preliminary ideas, that might be worthwhile for further study. Specifically, we propose the application of Shellsort for the merging of linearly ordered sets. Shellsort was invented by Donald Shell in 1959 and is based on insertion sort. The algorithm runs from left to right, by com paring elements at a given gap or increment d 2 N and exchanging them, if they are in reverse order, so in the array fa1; a2; : : : ; ang the d subarrays

faj ; aj+d; aj+2d; : : :g, for j = 1; 2; : : : ; d are separately sorted. At the second pass, Shellsort runs on smaller increment, until after a number of passes, the increment becomes d = 1. This final insertion sort completes the sorting of the array. The sequence of the increments is crucial for the running time of the algorithm, as the pivot selection is important to Quicksort. Thus the application of Shellsort might constitute an alternative choice for the merging of chains.

VI. REFERENCES

[1]. Abramowitz, M. and Stegun, I. A. (1972) "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables." Dover Publications.

[2]. Albacea, E. A. (2012) "Average-case analysis of Leapfrogging samplesort." Philipp. Sci. Lett. 5 (1): 14-16.

[3]. Albert, M. and Frieze A. (1989) "Random graph orders." Order 6 (1): 19-30.

[4]. Alon, N., Bollobás, B., Brightwell, G. and Janson, S. (1994) "Linear Extensions of a Random Partial Order." Ann. Appl. Probab. 4 (1): 108-123.

[5]. Bell, D. A. (1958) "The Principles of Sorting." Comput. J. 1 (2): 71-77.

[6]. Bentley, J. L. (2000) "Programming Pearls." Addison-Wesley Publishing, second edition.

[7]. Bentley, J. L. and McIlroy, M. D. (1993) "Engineering a Sort Function." Software Pract. Exper. 23 (11): 1249-1265.

[8]. Billingsley, P. (2012) "Probability and measure." John Wiley & So., third edition.

[9]. Boyce, W. E., DiPrima, R. C. (2001) "Elementary Differential Equations and Boundary Value Problems." John Wiley & So., seventh edition.