Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 7.056

IJCSMC, Vol. 9, Issue. 11, November 2020, pg.107 – 117

Breaking the O(n^2) Barrier: Novel Approaches to Time Complexity Analysis in Quick Sort

Pijush Kanti Kumar
Asst. Professor (Information Technology), Government College of Engineering and Textile Technology, Serampore-712201, India
Pijush752000@yahoo.com

DOI: https://doi.org/10.47760/ijcsmc.2020.v09i11.010

Abstract: *The most common type of sorting algorithm used is quicksort. As the name suggests it is the one of the most fastest sorting algorithm used since the innovation of sorting. However, this sort has often been criticised for its worst-case time complexity that is O(n^2). Practically the quick sort tends to follow its average case and best-case scenarios most of the time. So practically the quick sort is the most efficient practical sorting algorithm. In this paper we will fine tune this algorithm and remove its worst-case time complexity of O(n^2) and make this the best sorting algorithm both theoretically and practically. Various techniques will be discussed from pivot fixing to randomisation and further using medians of medians.*

1.RESEARCH OBJECTIVES

To analyse and research about reducing the worst-case time complexities of quick sort using techniques such as randomisation which will make it almost 99% worst case prone and further median of medians for 100% average case or best-case accuracy. In this paper we are going to reduce the worst-case time complexity of quick sort from O(n^2) to O(nlogn). No permutation of input sequence will be able to go to O(n^2) T.C.

© 2020, IJCSMC All Rights Reserved

2. LITERATURE REIVEW

Sorting is a fundamental operation in computer science and data processing for several important reasons like data organization, search efficiency, data retrieval and many more algorithms. It would take less than 1 minute for the internet to collapse if sorting not there.

Various types of sorting algorithms exits like bubble sort, selection sort, quick sort, merge sort , insertion sort and others. Among these sorting algorithms the best worst-case time complexity is given by merge sort which is O(nlogn). So the merge sort is theoretically better than the quick sort when both are judged on the run times in worst case scenarios. Then why quick sort is named “Quick”?

Reason is simple. Even the worst-case scenario of the quick sort is O(n^2), in most of the practical cases the quick sort is **constant time faster** than the merge sort and this gives an edge over all over sorting algorithms. So, if the worst-case scenario is somehow adjusted to its best case or average case then quick sort will be the most practical & theoretical based sorting algorithm. In this paper we will discuss the entire reduction of the worst-case time complexity in full details,

2.1 EXPLORING QUICK SORT

Quick sort is an application of Divide & Conquer technique. "Divide and conquer" is a fundamental algorithmic technique used to solve problems by breaking them down into smaller sub problems, then solving the sub problems independently, and then combining their solutions to solve the original problem. The quick sort is an emplace and unstable sorting algorithm. The actual code of quick sort contains a function called “Partition function”. The partition function is based on pivot selection. A pivot is an element which may be chosen as any of the elements randomly and after partition algo the elements lesser than the pivot element are to the left of the pivot and elements more than the pivot element are to the right of it. Now recursively we can call quick sort for the left and right parts. This is the core logic behind the working of the quick sort. A small psedo-code has been shown below for an example.

The pseudo code for quick sort is:

```
quickSort(arr[], low, high) { if (low < high) {
  p = partition(arr, low, high);
```

© 2020, IJCSMC All Rights Reserved

quickSort(arr, low, p – 1); quickSort(arr, p + 1, high);
}
}

In the pseudo code, low and high are the starting and ending indexes of the array. P is the position of the pivot after applying the partition algo which is mentioned below:

partition (arr[], low, high) { pivot = arr[low]
k = high
for (i = high; i > low; i–) { if (arr[i] > pivot){
swap arr[i] and arr[k]; k–;
}
}
swap arr[k] and arr[low] return k-1;
}

There exists many types of partition algorithms among which in the above case the first element is chosen as pivot. So, after the first partition algorithm the partition algorithm. returns the position of the pivot in such a way that all elements greater than the pivot are to the right of it and elements smaller than the pivot are to the left of it. Then quicksort is called recursively for the two parts.

2.2 UNDERSTANDING THE TIME & SPACE COMPLEXITIES OF THE QUICK SORT

We will in details analyse the best case, average case and worst case time complexities of the quick sort.

BEST CASE

The best case will occur only if the split is always n/2 and n/2. If the split is always like this, then the recurrence relation will be

$T(n) = 2 * T(n/2) + n$ {The extra n for partition algo} $T(n) = 2*(2*T(n/4) + n/2) + n$

$T(n) = 2^k * T(n/2^k) + k *n$

⇨ $2^k=n$

⇨ $k = \log n$

$T(n) = n*T(1) +n*\log n$ $T(n) = n\log n$

Alternatively solving by master's theorem the time complexity for T(n) is also O(nlogn).

WORST CASE

The worst case is achieved when the split always occurs in 1 and n-1 or 2 and n-2 like that. If the first element is chosen as pivot the worst-case scenario is achieved if the array is already sorted or all the elements are same.

$T(n) = T(n-1) + T(1) + n*constant$
$T(n) = T(n-2) + (n-1)*constant + n*constant$
.

© 2020, IJCSMC All Rights Reserved

.

T(n) = T(n-k) + k*n*constant – (k-1)*constant - .. – 2*constant- constant T(n) = T(n-k) + k*N*constant – constant*(k*(k-1))/2

Now putting k=n

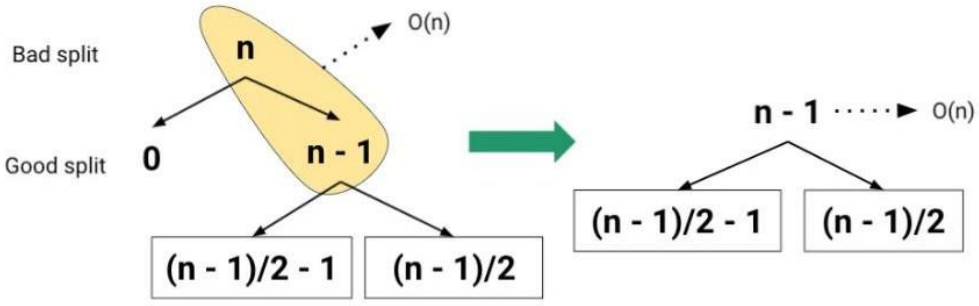
T(n) = T(0) + n*n *constant – constant*(n*(n-1)/2) T(n)= n2/2+n/2

T(n)= O(n^2)

AVERAGE CASE

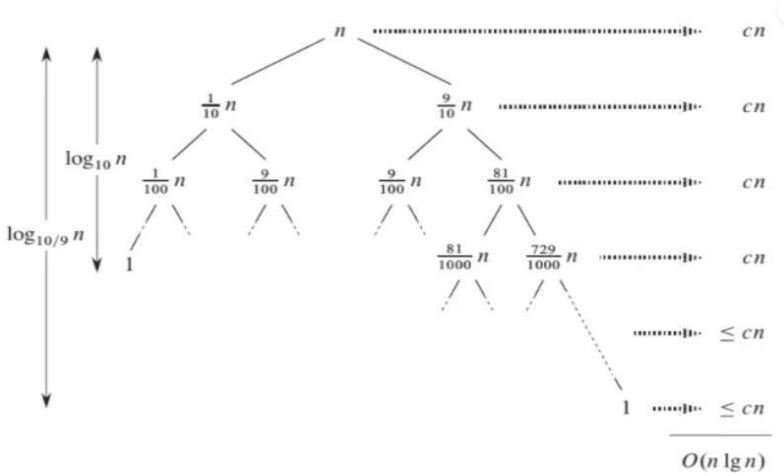
To discuss the average case let's say the array gets divided into k and n-k.

It's better to use intuitive assumption rather than mathematical derivation for the average case. When we select pivot randomly during the partition process, partitioning is highly unlikely to happen in the same way at every level, i.e. some of the splits will be close to a balanced scenario, and some will be close to an unbalanced scenario. Here, we assume that the probability of choosing each element as the pivot is equally likely.



In the average case, the partition process will generate a combination of good (balanced partition) and bad (unbalanced partition) splits, where good and bad splits will be distributed uniformly in the recursion tree. Suppose the partition algorithm always generates a partially unbalanced spit in the ratio of 9 to 1.

T(n) = T(9n/10) + T(n/10) + constant * n.



© 2020, IJCSMC All Rights Reserved

The left part of the recursion tree is decreasing fast with a factor of 1/10 and the right part is decreasing slowly with a factor of 9/10. So in the worst case, the maximum depth of the recursion tree will be equal to log10/9(n), which is O(logn). The partitioning cost is at most constant*n at every level of the recursion tree. After doing level by level sum, the total cost of the quick sort will be constant* n*O(logn) = O(nlogn). So for an unbalanced split of 9-to-1, quicksort works in O(nlogn) time complexity.

In general, time complexity will be O(nlogn) whenever split has constant proportionality. For example, 90-to-10 split will also produce O(nlogn) time complexity for quick sort.

The quick sort is an inplace sorting algorithm. The recursion will take a stack size of log(n) and hence the space complexity is O(logn)

2.3 HOARE'S VS LOMUTO PARTITION SCHEME

The basic concept of the partition algo was to partition the array into 2 splits such that elements lesser than pivot are to the left of it and elements to the right of the pivot are to the right of it. Majorly two types of partition schemes exists.

LOMUTO'S PARTITION SCHEME

Lomuto's algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element, then first swap it with the last element. Now initialize two variables i as low and j also low, iterate over the array and increment i when arr[j] <= pivot and swap arr[i] with arr[j] otherwise do only j++. After coming out from the loop swap arr[i] with arr[high]. This i stores the pivot element.

partition(arr[], lo, hi) pivot = arr[hi]

i = lo-1 // place for swapping for j := lo to hi – 1 do

if arr[j] <= pivot then i = i + 1

swap arr[i] with arr[j] swap arr[i+1] with arr[hi] return i+1

This i stores the pivot element. Generally, a random element of the array is located and picked and then exchanged with the first or the last element to give initial pivot values. In the algorithm, the last element of the list is considered as the initial pivot element. It always fixes the pivot element in the correct position.

HOARE'S PARTITION SCHEME

Hoare's algorithm works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on the left side and greater value on the right side) found. When an inversion is found, those two values are swapped and

© 2020, IJCSMC All Rights Reserved

the process is repeated until i>j.

```
partition(arr[], lo, hi) pivot = arr[lo]
  i = lo - 1  // Initialize left index
  j = hi + 1  // Initialize right index do
  i ++;
  while arr[i] < pivot
  do
  j--;
  while (arr[j] > pivot); if i >= j then
  return j
  swap arr[i] with arr[j]
```

Generally, the first item or the element is assumed to be the initial pivot element. Some choose the middle element and even the last element. It is generally faster but does not fix the pivot in the correct position.

2.4 BREAKING THE O(n^2) BARRIER

So, whatever the partition algorithm we get the worst case of quick sort as O(n^2). Till now we have discussed the core working of quick sort and observed that its only demerit is the worst case time complexity. Observing the average case time complexity we observed that only some input sequences were troublesome and that's the only reason why the quick sort took a time complexity of O(n^2). Actual reason behind these sequences are the pivot selection. As in CLRS the pivot is initially fixed at first position. So for an already sorted number sequence as input like 1 2 3 4 5 6

The partition will be as follows
(1) (2 3 4 5 6)
(1)((2)(3 4 5 6))
(1)((2)((3)(4 5 6)))
(1)((2)((3)((4)(5 6))))
(1)((2)((3)((4)((5)(6)))))
So we can observe that every time the partition is n and n-1. This sequence will lead to a worst case sequence of O(n^2).

Similarly sequences like 7 6 5 4 3 2 1 and 7 7 7 7 7 7 7 will lead to worst case scenarios. Above we say that partition algorithm has no role, so it must be the pivot selection. When we take the first or last element as the pivot the worst cases are achieved in the followings:

- ⇨ Already sorted sequence in ascending order
- ⇨ Already sorted sequence in descending order
- ⇨ All elements are same

© 2020, IJCSMC All Rights Reserved

2.5 SELECTING MIDDLE ELEMENT AS PIVOT & RANDOMISED ALGORITHM

Now if the middle element is selected as pivot every time does that mean quicksort will never go to it's worst case?

Let's try with an already sorted input like :
1 2 3 4 5 6 7 8
The partitions will be as follows:
(1 2 3 4) (5 6 7 8)
((1 2)(3 4))((5 6)(7 8))
(((1)(2))((3)(4)))(((5)(6))((7)(8))
So we can observe that a sorted input sequence is sorted within logn levels so the time complexity is O(nlogn).

But this procedure has it's own demerits like for cases where imagine selecting the middle element always splits the array into n and n-1. Also for a sequence where all numbers are same, selecting the middle element as pivot will lead to O(n^2) time complexity. So, this process is better than selecting first and last element as pivot but not efficient.

So, we can conclude that **"for any fixed pivot algorithm there exits an input sequence such that the quick sort goes to its worst-case scenarios"**. To solve this problem a random sequence of pivots can be chosen like for example where before first partition the first element can be chosen as pivot, then before 2^{nd} partition the middle element can be chosen as pivot and so on. This algorithm is called as the **"Randomized algorithm for quick sort"**.

This will ensure that there is no fixed pivot selection, but will it guarantee that quick sort will not go to the worst-case scenarios? Answer is **"NO".** Even the randomized algorithm can also choose the first element as pivot randomly every time. Choosing a position randomly also leads to a possibility where the same number is chosen every time. That makes randomized algo a fixed pivot algorithm and worst case again leads to O(n^2).

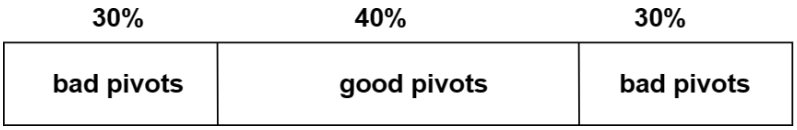
3. FINDING PIVOT USING MEDIANS OF MEDIANS

If we know the median element of an array and select the median element as the pivot, we can guarantee that the split will be n/2 and n/2. Using the above time complexity discussions we can ensure that such a split will always be O(nlogn). No matter even if the array is sorted or not or all the elements are same or not if the split itself is into n/2 and n/2 then even the worst case scenario of O(n^2) will boil down to O(nlogn).

© 2020, IJCSMC All Rights Reserved

The median and middle element of an array is different. The middle element is the element which is present in the middle of any array maybe sorted or unsorted but the median element is the element which is present in the centre of the sorted array only. So if median element is taken as pivot then it implies quick sort is implemented with O(nlogn) in it's worst case also, no matter whatever be the input sequence. But to achieve this the median of an array has to be calculated in O(n) time else the entire working procedure will fail.

The median of medians is a divide and conquer algorithm in that, it returns a pivot that in the worst case will divide a list of unsorted elements into sub-problems of size 3*n/10 and 7*n/10 assuming we choose a sublist size of 5. This will guarantee a good pivot that in the worst case will give a pivot in the range between 30th and 70th percentile of the list of size n. For a pivot to be considered good it is essential for it to be around the middle, 30-70% guarantees the pivot will be around the middle 40% of the list. So, the split will never be of the type n and n-k and ensure that it never goes to its worst cases.

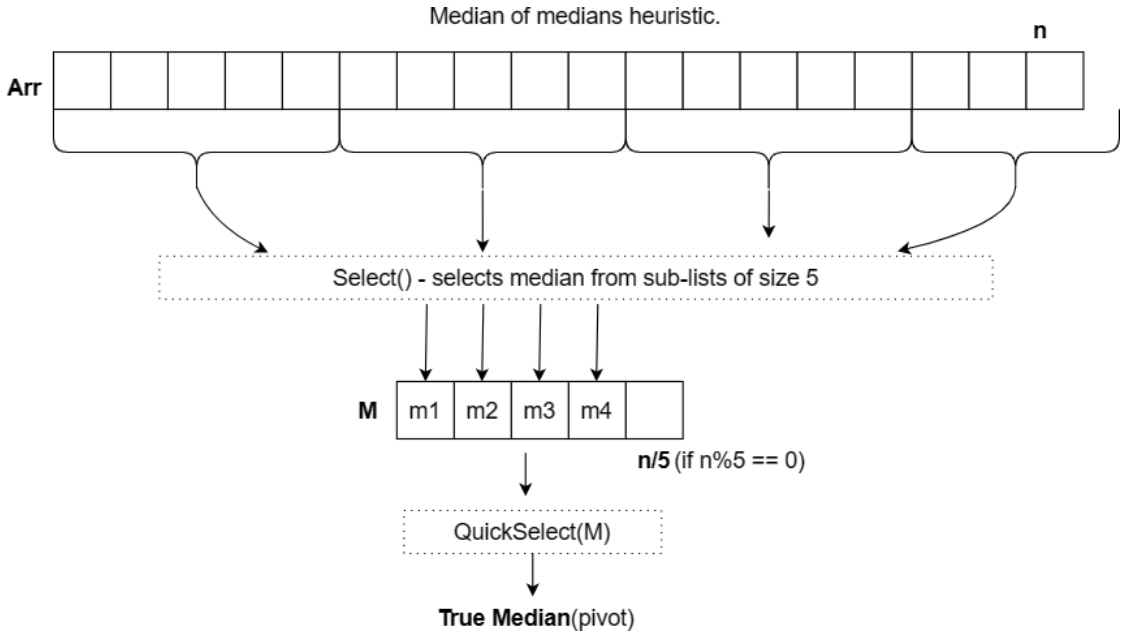


Let us understand the working of the median of medians algorithm in detail. The algorithm is as follows,

- Divide the list into sub lists or sub arrays of size n, assume 5 (This can be any odd number to prevent additional computations).
- Initialize an empty array Med to store medians we obtain from smaller sub lists.
- Loop through the whole list in sizes of 5, assuming our list is divisible by 5. (The last list may not have 5 elements but that does not matter since we will use brute force technique for calculating the median)
- For n/5 sub lists, use select brute-force subroutine to select a median m, which is in the 3rd rank out of 5 elements. Now this is median we can use brute force comparison here to find the 3^{rd} largest element or simple sort 5 elements using bubble sort and return the middle element, Sorting 5 elements will take constant time or O(1) time complexity.
- Append medians obtained from the sub lists to the array Med.

© 2020, IJCSMC All Rights Reserved

- Use quickselect recursively to find the true median from array Med, The median obtained is the viable pivot.
- Terminate the algorithm once the base case is hit, that is, when the sub list becomes small enough. Use Select brute-force subroutine to find the median.



Think of the following set of numbers:
5 2 6 3 1
The median of these numbers is 3. Now if you have a number n, if n > 3, then it is bigger than at least half of the numbers above. If n < 3, then it is smaller than at least half of the numbers above. Basically, this is the idea.

Now if you get the median of those numbers (call it m), it is bigger than half of them and smaller than the other half (by definition of median!). In other words, m is bigger than n / 10 numbers (which themselves were medians of small 5 element groups) and bigger than another n / 10 numbers (which again were medians of small 5 element groups).

In the example above, we saw that if the median is k and you have m > k, then m is also bigger than 2 other numbers (that were themselves smaller than k). This means that for each of those smaller 5 element groups where m was bigger than its median, m is also bigger than two other numbers. This makes it at least 3 numbers (2 numbers + the median itself) in each of those n / 10 small 5 element groups, that are smaller than m. Hence, m is at least bigger than 3n/10 numbers.

Are we always getting n/2 ad n/2 then?

© 2020, IJCSMC All Rights Reserved

Psedocode is as follows:
medianOfMedians(arr[1...n])
if(n < 5 return Select(arr[1...n], n/2)) Let M be an empty list

For i from 0 to n/5 - 1:
Let m = Select(arr[5i + 1...5i+5], 3) Add m to M

Return QuickSelect(M[1...n/5], n/10) End medianOfMedians

Answer is “NO” we are not getting a half-half split but it always gives somewhere between 30-70 and 70-30. For quicksort to give O(nlogn) time complexity, it needs to be able to divide the array into partitions proportional to the size of the array. That's what gives the logn recursion depth. 30-70 division already gives a 3n/10 and 7n/10 division which is proportional to n. So even though it's not a perfect 50-50, it will still yield logarithmic recursion depth (except it's not log in base 2, but base 10/7). This has already been discussed in this paper during the analysis of average case time complexity.

The median of medians will return a pivot element that is greater than and less than at least 30% of all elements in the whole list.

T(n/5) means recursive call when finding medOfmed value when we use medOfmed as a pivot, after partitioning, assure that at least $3((n/5)/2-2)$ elements must be less or larger than medOfmed. (where, -2 term means except for medOfmed group and last group.)

Therefore, next recursive call for KthSmallest selcect array size is $n-3((n/5)/2-2)=7n/10+6$.

$T(n)=T(n/5)+T(n-3((n/5)/2-2))+O(n)$

$=T(n/5)+T(7n/10+6)+O(n)$
$=T(n/5)+T(7n/10)+O(n)$
$=O(n)$

From recurrence tree or master theorem, it can be shown that the median can be calculated in O(n) time. Now that median of medians can be calculated in O(n) time and the pivot always lies in 30-70% range we can say that the worst-case runtime of quick sort will be O(n*logn) and not O(n^2).

© 2020, IJCSMC All Rights Reserved

CONCLUSIONS AND FUTURE WORK

In this paper successfully reduced the worst-case time complexities of Quick sort to O(nlogn). But this also slightly increased the overhead calculations resulting in poorer runtime in practical scenarios. In future algorithms can be developed to reduce this overhead. However, till date it can be concluded that quicksort using medians of medians always yields in O(nlogn) time complexity in all three best, average and worst cases respectively. No permutation of input sequence will be able to take quick sort to its O(n^2) scenario.

REFERENCES

[1]. Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
[2]. https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/
[3]. https://sungwookyoo.github.io/algorithms/QuickSortMedian/
[4]. https://medium.com/enjoy-algorithm/quicksort-one-the-fastest-sorting- algorithms-2685fb0910c5
[5]. https://iq.opengenus.org/median-of-medians/
[6]. https://stackoverflow.com/questions/12545795/explanation-of-the-median-of- medians-algorithm
[7]. https://gateoverflow.in/2048/gate-cse-2014-set-3-question-14
[8]. https://en.wikipedia.org/wiki/Quicksort

© 2020, IJCSMC All Rights Reserved