CS 584 DESIGN AND ANALYSIS OF ALGORITHM PROJECT REPORT

QuickSort using MedianOfMedianGroup5, MedianOfMedianGroup7 and RandomizedQuickSort

PSU ID: 926231001
Shweta Korulkar
Computer Science

March 15, 2019

Contents

1 INTRODUCTION

QuickSort is one of the efficient sorting algorithm which place elements of an array or random access file in an increasing order. The major drawback of naive implementation of quicksort is that it's worst case complexity is $O(n^2)$ with input size n.

Efficiency of quicksort depends on the selection of the pivot element. There are different algorithms available for pivot selection with strategies such as median-of-median using group 5, median-of-median using group 7 and randomized pivot selection which make the time complexity of quicksort $O(n \log n)$. We are going to see in the following sections all these pivot selection strategies in detail and we will see which algorithm is more efficient for quicksort.

The paper is organized as follows: ***In Section 2*** we will be discussing about the Lomuto Partitioning technique. ***In Section 3***, we will be discussing about the naive implementation of quicksort . It will cover the worst case and best case time complexity of quicksort. Then it goes on to show that by selecting the last element as a pivot we will get the worst case running time as $\Theta(n^2)$ and best case running time as $\Theta(n \log n)$. To get better running time than this, ***Section 4*** presents a version of quicksort that uses median of median as a pivot. Using this algorithm we can improve the performance of quicksort. This method gives us quicksort in $O(n \log n)$ worst case time. Even though this method is theoretically important, is has been considered as impractical to use because of it's large constant factor. So to improve this performance, ***Section 5*** presents a version of quicksort that uses randomized sampling. This algorithm has a good expected running time, and no particular input elicits its worst case behaviour. This algorithm will run in $\Theta(n^2)$ time in worst case and $O(n \log n)$ in expected time. ***In Section 6*** we are going to see the execution performance of all these algorithms and comparison between these techniques. ***Section 7*** shows the input and output for source code. ***Section 8*** will be conclusion.

2 LOMUTO PARTITIONING

This partitioning method selects the pivot as a last element of an array, around which to partition the subarray. After selection of pivot, algorithm uses index 'i' and 'j' to scan the array such that elements from index 'p' to 'i' i.e. *A[p..i]* are less than or equal to pivot 'x' and elements from index 'i+1' to 'j-1' i.e. *A[i+1...j-1]* are greater than pivot 'x'. Here 'p' is the leftmost index of an array 'A'. This partitioning method is easy to understand so it is frequently used than Hoare partition even though Hoare partition is more efficient.

Algorithm 1 Partitioning the array

```
1: procedure PARTITION(A, p, r)
2:     x = A[r]
3:     i = p - 1
4:     for j = p to r-1 do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             exchange A[i] with A[j]
8:     exchange A[i + 1] with A[r]
9:     return i+1
```

The procedure *PARTITION(A, p, r)* takes 3 arguments as an input: input array A, leftmost array index 'p' and rightmost array index 'r'. This method selects A[r] i.e last element of an array A as a pivot element in line 2 to partition the subarray A[p..r]. Lines 4-8 will execute till we get the pivot fixed position. To get the pivot's new index it swap the element A[j] which is less than or equal to pivot x with A[i] and final two lines finish up by swapping the pivot with leftmost element greater than x, thereby moving the pivot into it's correct place in the partitioning array, and then returning the pivot's new index.

The running time of partitioning the array A[p..r] is $\Theta(n)$ since lines 4-7 executes n times where n = r- p + 1.

3 QUICKSORT USING LAST ELEMENT AS A PIVOT

QuickSort is a "big-oh" (O) efficient sorting algorithm. It uses divide and conquer paradigm to sort the subarray. Here I am performing QuickSort by choosing the last element of an array as a pivot around which to partition the subarray. To partition the array into subarrays I am using here the Lomuto Partition technique. The following algorithm shows the implementation of quicksort using last element of an array as a pivot:

Algorithm 2 QuickSort

```
1: procedure QUICKSORT(A, p, r)
2:     if p <r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q - 1)
5:         QUICKSORT(A, q + 1, r)
```

The above algorithm implements the quicksort by considering last element as a pivot. To sort the entire array A the initial call is *QUICKSORT(A, 1, A.length)*. QUICKSORT method takes 3 arguments as input: an input array A, leftmost index 'p' of an array A and rightmost index 'r' of an array A. Lines 3-5 will execute till 'p' is less than 'r'. Line 3 calls the partition method which is discussed in section 2 to fix the pivot position such that all elements less than or equal to pivot are on the left half of an array and all elements greater than pivot are on right half of an array. This partitioning method will return the pivot's new index. Lines 4-5 calls the function QUICKSORT recursively for both subarrays.

Time Complexity:

Algorithm *QuickSort* takes constant time at line 2. Lines 4-5 recursively call the QUICKSORT, so this will take $\Theta(\log n)$ time as the recurrence terminates at depth $\log n$ and partitioning takes $\Theta(n)$ time. So overall it will take $\Theta(n \log n)$ time. Like insertion sort, quicksort has tight code and so the hidden constant factor in it's running time is small.

The performance of this algorithm depends on whether the partitioning is

balanced or unbalanced. If partitioning is balanced then algorithm runs as fast as merge sort and if partitioning is unbalanced then algorithm runs asymptotically as slow as the insertion sort. The unbalanced partition is like one subproblem with n-1 elements and one subproblem with 0 elements. The recurrence relation for subproblem with 0 elements is $T(0) = \Theta(1)$ and recurrence for running time is,

$T(n) = T(n-1) + \Theta(n)$ for n >0

We can prove this recurrence relation by using substitution method to get the solution $T(n) = \Theta(n^2)$ in worst case. Thus if the partitioning is unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. For balanced partition where partition produces two subproblems of equal size, the recurrence for running time is then,

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

We can prove this recurrence relation by using master theorem to get the solution $T(n) = \Theta(n \log n)$ in best case time. Therefore using this method quicksort will take $\Theta(n \log n)$ best case time and $\Theta(n^2)$ worst case time.

4 QUICKSORT USING MEDIAN-OF-MEDIAN AS A PIVOT

This section explains the median-of-medians linear time algorithm for pivot selection. The idea used here is to perform quicksort using median of median as a pivot element around which to partition the subarray. So here we can guarantee a good split upon partitioning the array. Using this method we can get the $O(n \log n)$ running time of quicksort. In this project I performed the quicksort using median-of-median by group 5 and group 7. Implementation using group 7 is same as below algorithm, the only difference is that in group 7, we are dividing the n elements of an input array into $\lfloor \frac{n}{7} \rfloor$ group of 7 elements each which also takes $O(n \log n)$ time overall. The Implementation of quicksort using median-of-median by group 5 is shown below.

Algorithm 3 Find Median-of-Median element using group 5

```
1:  procedure MEDIAN_QUICK(q, n)
2:      sublist = [ ]
3:      median = [ ]
4:      pivot = None
5:      if q.length ≥ 1 then
6:          for i in range(0, n, 5) do
7:              sublist.append(q[i : i + 5])
8:          for j in sublist do
9:              s = sorted(j)
10:             if s.length >0 then
11:                 median.append(s[(s.length // 2)])
12:         if median.length ≤ 5 then
13:             sorted(median)
14:             pivot = median[median.length // 2]
15:         else
16:             sorted(median)
17:             pivot = median_quick(median, median.length // 2)
18:     return pivot
```

Algorithm 4 QuickSort

```
1:  procedure QUICKSORT(A, p, r)
2:      if p <r then
3:          q = MEDIAN_QUICK(A[p: r+1], A.length)
4:          par = PARTITION(A, p, r, q)
5:          QUICKSORT(A, p, par - 1)
6:          QUICKSORT(A, par + 1, r)
```

Algorithm 5 Partitioning the array

```
1:  procedure PARTITION(A, l, r, pivot)
2:      for i = 0 to r do
3:          if A[i] = pivot then
4:              break
5:      A[i], A[r] = A[r], A[i]
6:      x = A[r]
7:      i = l - 1
8:      for j = l to r do
9:          if A[j] ≤ x then
10:             i = i + 1
11:             exchange A[i] with A[j]
12:     exchange A[i+1] with A[i]
13:     return i + 1
```

Here we used the median-of-median element as a pivot. We do so by first exchanging last element of array A i.e. A[r] with median-of-median element. To sort the entire array A, the initial call is *QUICKSORT(A, 1, A.length)*. The idea used here is, partition the input array A around the median-of-medians using modified version of PARTITION.

Algorithm QUICKSORT takes 3 arguments as a input: an input array A, leftmost index 'p' of an array A and rightmost index 'r' of an array A. The only difference in this algorithm compared to Section 3 quicksort algorithm is lines 3-4. On line 3, MEDIAN_QUICK method is called to select the median of medians element of an array A. This method will return the pivot element. Line 4 call PARTITION methods with 4 parameters as an input i.e. array A, leftmost index p, rightmost index r and pivot q. Lines 5-6 call QUICKSORT recursively to sort the array A.

In algorithm PARTITION, lines 2-5 exchange the pivot element with A[r] i.e. last element of an array. Lines 6-13 work the same as the partition function described in the Section 2 which places pivot at a fixed position.

Algorithm MEDIAN_QUICK works as follows: It takes two parameters as input i.e. array 'q' and 'n' is the length of array. Line 5-17 will execute if array contains more than 1 element. 'For' loop of lines 6-7, is dividing the n elements of the input array q into $\lfloor \frac{n}{5} \rfloor$ group of 5 elements each and storing these subarrays into

array sublists at line 7. 'For' loop of lines 8-11 is finding the medians of all these sorted subarrays and storing these medians into array *median* at line 11. Lines 12-14 or 15-17 will choose the median of this median array as a pivot element. If length of median array is less than or equal to 5 then lines 12-14 will execute otherwise lines 15-17 will execute. Line 18 returns the pivot.

Time Complexity:

Lines 6-7 of MEDIAN_QUICK takes $O(n)$ time to divide the array into subarrays. lines 8-11 consist of $O(n)$ call of insertion sort on the set of size $O(1)$. Line 17 takes $T(\frac{n}{5})$ times to recursively find the median-of-medians. This algorithm guarantees that $\frac{3}{10}$ of input elements are less than the output pivot and other $\frac{3}{10}$ elements are greater than output pivot. This yields that one partition operation reduces the length of A from n to $\frac{7n}{10}$ in the worst case. Therefore the algorithm MEDIAN_QUICK takes $O(n)$ time to find the median-of-median element. PARTITION algorithm also takes $O(n)$ time. and QUICKSORT algorithm takes $O(n \log n)$ time to sort the array recursively as recurrence terminate at depth $\log n$. Therefore total time complexity of sorting with this method is $O(n \log n)$

The overall recurrence relation is,

$T(n) = 2T(\frac{n}{2}) + O(n) + O(n)$

By solving this recurrence by master theorem we get the solution $T(n) = O(n \log n)$. But using this method we get a large constant. Hence sorting using this method is inefficient.

5 RANDOMIZED VERSION OF QUICKSORT

In this section we are going to see the random sampling technique to perform the quicksort. Here we will select the randomly chosen element from subarray A[p..r] as a pivot. We do so by first exchanging last element of array A i.e. A[r] with element chosen at random from A[p..r]. By randomly sampling the range p...r, we ensure that pivot element x = A[r] is equally likely to be any of the r - p + 1 elements in the subarray. Because we randomly chose the pivot element, we

expect the split of the input array to be reasonably well balanced on average. The change to PARTITION and QUICKSORT are small. In this new partition procedure i.e RANDOMIZED_PARTITION, we simply implement the swap before actually partitioning.

Algorithm 6 QuickSort

```
1: procedure RANDOMIZED_QUICKSORT(A, p, r)
2:     if p <r then
3:         q = RANDOMIZED_PARTITION(A, p, r)
4:         RANDOMIZED_QUICKSORT(A, p, q - 1)
5:         RANDOMIZED_QUICKSORT(A, q + 1, r)
```

Algorithm 7 Selecting Random element as a pivot

```
1: procedure RANDOMIZED_PARTITION(A, p, r)
2:     i = randint(p, r)
3:     exchange A[i] with A[r]
4:     return PARTITION(A, p, r)
```

The procedure RANDOMIZED_PARTITION is selecting random element A[i] from array A as a pivot and exchanging the A[i] with last element of an array i.e A[r]. randint() returns the index of randomly chosen element at line 2. At line 4, PARTITION function is called. This partition function is described in the Section 2.

Time Complexity:

The worst case time for RANDOMIZED_QUICKSORT is $\Theta(n^2)$. The recurrence relation is,

$T(n)$ = max $(T(q) + T(n-q-1) + \Theta(n))$ where q ranges from 0 to n - 1 because the procedure PARTITION produces two subproblems with total size n - 1. By using substitution we can prove this recurrence to get the solution $T(n) = O(n^2)$. The expected running time of RANDOMIZED_QUICKSORT is $O(n \log n)$ because if in each level of recursion the split introduced by RANDOMIZED_PARTITION puts any constant fraction of elements on one side of

the partition, then the recursion tree has the depth $\Theta(\log n)$ and $O(n)$ work performed at each level. Therefore expected running time is $O(n \log n)$ when element values are distinct. We can prove this as follows,

Correctness:

The running time of quicksort depends on the number of comparisons performed in all calls to the RANDOMIZED_QUICKSORT routine.

Let z_i denote the i^{th} smallest element of A[1..n]. Let Z_{ij}=$z_i, \ldots, z_j$ denote the set of elements between z_i and z_j including these elements. Suppose we chose the pivot element in Z_{ij}=$z_i, \ldots, z_j$ then probability of comparing the elements z_i and z_j is

Pr[z_i or z_j is the first pivot chosen from Z_{ij}]
= Pr[z_i is the first pivot chosen from Z_{ij}] + Pr[z_j is the first pivot chosen from Z_{ij}]
= (1 / j-i+1) + (1 / j-i+1) = (2 / j-i+1)

So the expected number of comparison is,

$$E(x) = \sum_{i \to 1}^{n-1} \sum_{j \to iplus1}^{n} (2 / j\text{-}i+1)$$

let's put k = j - i

$$E(x) = \sum_{i \to 1}^{n-1} \sum_{k \to 1}^{n-i} (2 / k+1)$$

$$E(x) < \sum_{i \to 1}^{n-1} \sum_{k \to 1}^{n} (2 / k)$$

$$= \sum_{i \to 1}^{n-1} O(\log n) = O(n \log n)$$

Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.

6 EXPERIMENT

I carried out the experiments to estimate the efficiency of the 4 variants of quicksort algorithms i.e. naive implementation of quicksort, median-of-median using group 5 and group 7 as a pivot selection strategy and randomized pivot selection.

I used random sequence of distinct integers of various sizes as an input array. I executed each algorithm 5 times for each input size, obtained the mean and performed comparison between these algorithms. Figure 1 shows the execution time taken by each algorithm (log scale) and figure 2 shows the comparison graph between these 4 pivot selection algorithms. X-axis represents the length of an input array in logarithmic scale and Y-axis represents the execution time in seconds.

Input_array_length	NaiveQuickSort	RandomizedVersion	MoM5	MoM7
100	6.11E-07	5.74E-07	3.77E-06	1.865E-06
1000	2.364E-06	4.247E-06	4.247E-06	6.225E-05
5000	1.3248E-06	1.934E-05	1.934E-05	0.001512
10000	2.97062E-05	4.6422E-05	222	160
50000	0.22277	0.25739	511	363
100000	0.771011	1.1557	1575	1232

Figure 1: Execution_Time

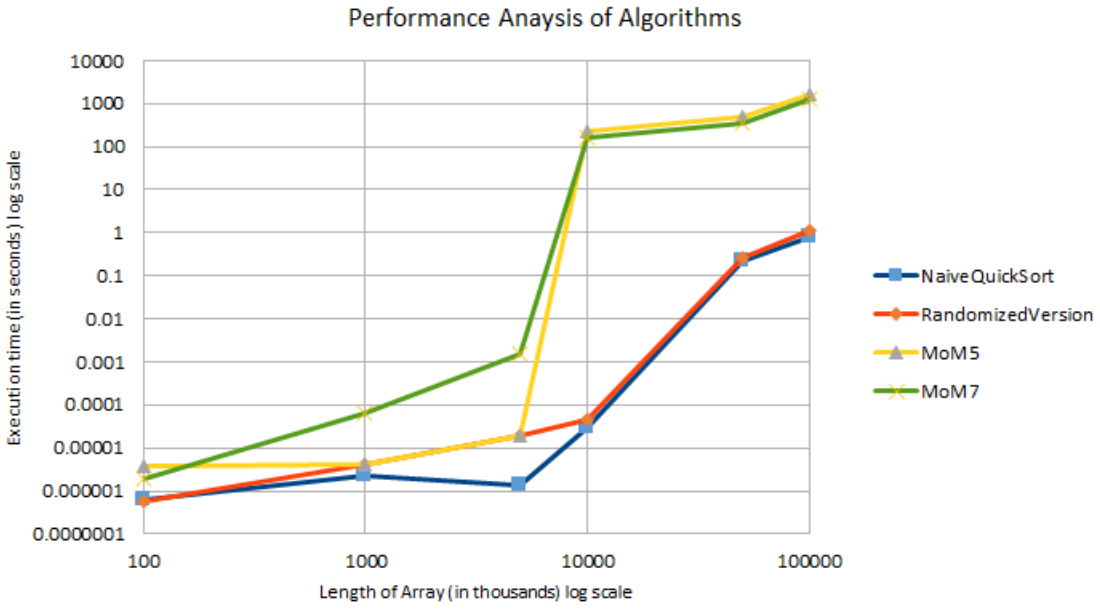

Figure 2: Performance Analysis and comparison of 4 algorithm

In above graph we can see that quicksort using median-of-median by group 5

and group 7 is not good option because of it's expensive pivot selection method. Pivot selection using these methods takes lot of time for the large input array. On the other hand, randomized version of quicksort and naive implementation of quicksort produces excellent results. They beat both median-of-median pivot selection methods.

I did the verification of the sorted array by comparing it with the output of python's inbuilt sort method in order to check the correctness of the sorted array. As we were expecting the better performance with median-of-median pivot selection method but after seeing this graph we can say that median-of-median pivot selection methods are not as good for large input array. We can see that on randomly generated input, performance of quicksort using median-of-median by group 5 is not good because of it's pivot selection method. In naive implementation of quicksort we are getting $O(n^2)$ worst case time because of occasional unbalanced partition but in randomized version of quicksort we will almost get balanced partition so, the performance is very good with randomized version of quicksort.

7 IMPLEMENTATION: INPUT AND OUTPUT TO SOURCE CODE

The source code is implemented in Python. I generated the random numbers in Input.txt file and copied these numbers in an array to provide this array as an input to the the main sorting code. The length of the input arrays used are: 100, 1000, 5000, 10,000, 50,000 and 1,00,000. I executed all 4 algorithms on each input for performance evaluation. To execute all 4 algorithm files, we first need to generate the random numbers in the Input.txt file by executing RandomNumber-Generator.py file. Output generated by these algorithms is a sorted array which is written in the output.txt file.

8 CONCLUSION

In this report, I show the different pivot selection methods for quicksort. After studying all these methods and comparing them, we can say that quicksort using randomized version is always a better choice for pivot selection to get an upper bound $O(n \log n)$ in expected time. It is unlikely that this randomized version of pivot selection algorithm will choose a terribly unbalanced partition each time, so the performance is very good almost all the time.

REFERENCES

1] Introduction to Algorithms Third Edition by CLRS.

2] Wikipedia

Source code link: https://github.com/sk192/AlgorithmProject