COURSE NAME: DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE AND SECTION: CSE5311

A Report On

Implement the median of medians (median-of-3, median-of-5, and median-of-7) algorithm and the randomized median finding algorithm and compare their performances.

(Submitted in partial fulfilment of the requirements for the award of Degree)

MASTER'S

In

COMPUTER SCIENCE

BY

Aman Bheemisetty(1002087953)

Rudranshi Dave(1001633017)

Under the Guidance of

Md Hasanuzzaman Noor **(Professor)**
&
Jacob Devasier (**GTA**)

Academic Session: Fall 22

The University of Texas, Arlington

Department of Computer Science and Engineering

Abstract

The quickselect,which chooses the kth smallest element from an initially unsorted array, commonly uses the median of medians, an approximation (median) selection procedure, as an useful pivot. The median of medians only locates a rough median in linear time, which is constrained but adds extra overhead for quickselect. The worst-case complexity of quickselect decreases dramatically from quadratic to linear when this approximate median is employed as an improved pivot; this is also the asymptotically ideal worst-case complexity of any selection process.

Problem Statement

Implement the median of medians (median-of-3, median-of-5, and median-of-7) algorithm and the randomized median finding algorithm and compare their performances.

Introduction

Median of median(median-of-3):

A point that is guaranteed to be between the 33.33th and 66.66th percentiles is the approximate median that the median-of-medians algorithm calculates (in the middle 4 deciles). Thus, there is a minimum 33.33% reduction in the search set. The problem is fixed proportionally smaller at 66.66% of its initial magnitude. The cost of using the same procedure again on the now-smaller set until only one or two elements are left is n / 1-0.9, or approximately 10.

The real median-of-medians algorithm is subroutine pivot. It first divides its input into groups of three pieces, then uses a subroutine to calculate the median of each of those groups. Finally, it recursively calculates the true median of the n/3 medians discovered in the previous step.

Median of median(median-of-5):

A point that is guaranteed to be between the 30th and 70th percentiles is the approximate median that the median-of-medians algorithm calculates (in the middle 4 deciles). Thus, there is a minimum 30% reduction in the search set. The problem is fixed proportionally smaller at 66.66% of its initial magnitude. The cost of using the same procedure again on the now-smaller set until only one or two elements are left is n / 1-0.7, or approximately 3.33.

The real median-of-medians algorithm is subroutine pivot. It first divides its input into groups of five pieces, then uses a subroutine to calculate the median of each of those groups. Finally, it recursively calculates the true median of the n/5 medians discovered in the previous step.

Median of median(median-of-7):

The real median-of-medians algorithm is subroutine pivot. It first divides its input into groups of seven pieces, then uses a subroutine to calculate the median of each of those groups. Finally, it recursively calculates the true median of the n/7 medians discovered in the previous step.

Randomized median finding :

Choosing a pivot element p, computing its rank, and splitting the list into two sublists—one having elements smaller than p and the other containing elements larger than p—is a natural method. We now provide a straightforward probabilistic strategy. Simply choose one element from the list that was consistently chosen at random to serve as the pivot point. It's really easy to put this into practice. In the worst-case scenario, we might execute nearly cn2 comparisons since we were unlucky at every stage, but it is a very unlikely occurrence.

Implementation:

- Python 3.x+

Libraries used:

- Python time library

- Python numpy library
- Python pandas library

Code:

Median of median(median-of-3/mo3):

```
def nthSmallest(a, l, r, k):
    if (k > 0 and k <= r - l + 1):
        n = r - l + 1
        median = []
        i = 0
        while (i < n // 3):
            median.append(findMedian(arr, l + i * 3, 3))
            i += 1
        if (i * 3 < n):
            median.append(findMedian(a, l + i * 3, n % 3))
            i += 1
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = nthSmallest(median, 0, i - 1, i // 2)
        pos = partition(a, l, r, medOfMed)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return nthSmallest(a, l, pos - 1, k)
        return nthSmallest(a, pos + 1, r,k - pos + l - 1)

    return 999999999999
```

```
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp


def partition(a, l, r, x):
    for i in range(l, r):
        if arr[i] == x:
            swap(a, r, i)
            break
    x = a[r]
    i = l
    for j in range(l, r):
        if (a[j] <= x):
            swap(a, i, j)
            i += 1
    swap(a, i, r)
    return i
```

```
def findMedian(a, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(a[i])
    lis.sort()
    return lis[n // 2]
```

Running time :O(nlogn)

Median of median(median-of-5/mo5):

```
def nthSmallest(a, l, r, k):
    if (k > 0 and k <= r - l + 1):
        n = r - l + 1
        median = []
        i = 0
        while (i < n // 5):
            median.append(findMedian(a, l + i * 5, 5))
            i += 1
        if (i * 5 < n):
            median.append(findMedian(a, l + i * 5, n % 5))
            i += 1
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = nthSmallest(median, 0, i - 1, i // 2)
        pos = partition(a, l, r, medOfMed)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return nthSmallest(a, l, pos - 1, k)
        return nthSmallest(a, pos + 1, r,k - pos + l - 1)
    return 999999999999
```

```
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp


def partition(a, l, r, x):
    for i in range(l, r):
        if a[i] == x:
            swap(a, r, i)
            break
    x = a[r]
    i = l
    for j in range(l, r):
        if (a[j] <= x):
            swap(a, i, j)
            i += 1
    swap(a, i, r)
    return i
```

```
def findMedian(a, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(a[i])

    lis.sort()

    return lis[n // 2]
```

Running time :O(n)

Median of median(median-of-7/mo7):

```
def nthSmallest(a, l, r, k):
    if (k > 0 and k <= r - l + 1):
        n = r - l + 1
        median = []
        i = 0
        while (i < n // 7):
            median.append(findMedian(a, l + i * 7, 7))
            i += 1
        if (i * 7 < n):
            median.append(findMedian(a, l + i * 7, n % 7))
            i += 1
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = nthSmallest(median, 0, i - 1, i // 2)
        pos = partition(a, l, r, medOfMed)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return nthSmallest(arr, l, pos - 1, k)
        return nthSmallest(arr, pos + 1, r,k - pos + l - 1)
    return 999999999999
```

```
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp


def partition(a, l, r, x):
    for i in range(l, r):
        if a[i] == x:
            swap(a, r, i)
            break
    x = a[r]
    i = l
    for j in range(l, r):
        if (a[j] <= x):
            swap(a, i, j)
            i += 1
    swap(a, i, r)
    return i
```

```
def findMedian(a, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(a[i])
    lis.sort()
    return lis[n // 2]
```

Running time :O(n)

Randomized median(RA) :

```
def find_median_randomized(S):
    print(S)
    assert (isinstance(S, list) and len(S) > 0)
    for num in S:
        assert isinstance(num, int)
    n = len(S)
    random.shuffle(S)
    number_of_samples = int(math.ceil(n ** (3.0 / 4.0)))
    R = random.sample(S, number_of_samples)
    R.sort()
    d_index = int(math.floor(((n ** (3.0 / 4.0)) / 2.0) - math.sqrt(n)))
    d = R[d_index]
    u_index = int(math.ceil(((n ** (3.0 / 4.0)) / 2.0) + math.sqrt(n)))
    u = R[u_index]
    c = [x for x in S if d <= x and x <= u]
    ld = len([x for x in S if x < d])
    lu = len([x for x in S if x > u])
    assert not (ld > (n / 2) or lu > (n / 2))
    assert len(c) <= 4.0 * (n ** (3.0 / 4.0))
    c.sort()
    median_index = int(math.floor(n / 2) - ld + 1)
    print(c)
    return c[median_index]
```

Running time :O(n^2)

Running Time as per code executed:

Number of Elements	MO3	MO5	MO7	RA
500	0.00400	0.00212	0.00109	0.0098
1000	0.0117	0.0060	0.0060	0.0498
2000	0.0238	0.00801	0.014	0.069
5000	0.05183	0.0193	0.0200	0.101
10000	0.1246	0.0279	0.0709	0.139

Time Complexity :

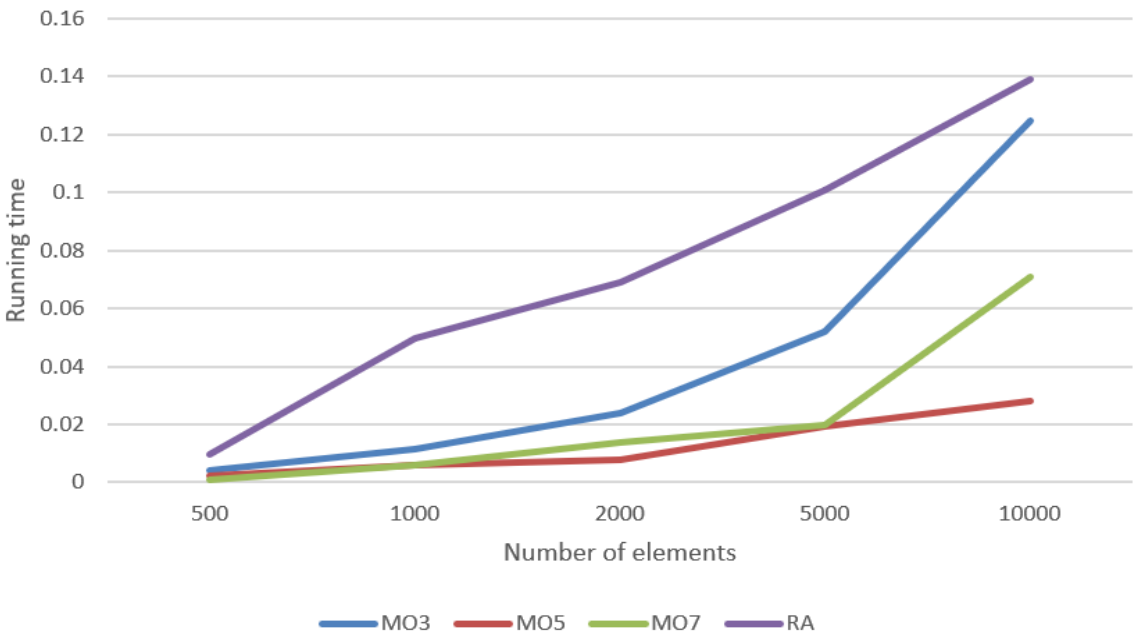
Randomized Algorithm
The randomized method, like quicksort, performs well on average but is sensitive to the pivot that is selected. By induction (or adding the geometric series), one can see that performance is linear because each step is linear and the total time is a constant multiplied by this. If good pivots are chosen, that is, ones that consistently reduce the search set by a given fraction, then the search set decreases in size exponentially (depending on how quickly the search set reduces). The worst-case performance, however, is **quadratic: O(n^2**) if terrible pivots are repeatedly chosen, such as reducing by only one element every time. This happens, for instance, when a set's largest element is sought after after being sorted and the first element is used as the pivot.
Best case: O(n)
Worst case: O(n^2)

Median of Median (grouping of 3 elements)
When we divide the input elements into groups of 3 elements each, we can still compute the number of elements that are greater than $m*$ and smaller than $m*$ similarly. We get that at least $2([1/2\ [\ n\ /3]\ ] - 2)$ elements are greater than $m*$ and a like number are smaller than $m*$. Thus, the size of the subproblem is at most $n - (\ [n/\ 3] - 4) = [2n/\ 3] + 4$. The recurrence relation for the running time becomes $T(n) \leq T([n/3]) + T(\ [2n/\ 3] + 4) + O(n)$
Best case: O(n)
Worst case: O(nlogn)

Median of Median (grouping of 5 elements)
In order to obtain the true median, which will be used as a pivot, we scan the list twice: once to discover the medians in the sublists. This technique has an **O(n) linear time complexity**. A pivot element that is bigger than and less than at least 30% of all the elements in the entire list will be returned by the median of medians.
Best case: O(n)
Worst case: O(n)

Median of Median (grouping of 7 elements)
The key to the success of this algorithm is that every step discards a constant fraction of the elements. The median of (2k+1) has k elements on either sides. Then the median of medians of (2k+1)(2l+1) elements has certainly (k+1)(l+1)-1 elements on either sides (every median is no smaller than k+1 elements and there are l+1 medians, all no smaller than the median of medians). The fraction is ((k+1)(l+1)-1)/(2k+1)(2l+1). In the case of k=2, we have (3(l+1)-1)/5(2l+1) ~ 3l/10.
Best case: O(n)
Worst case: O(n)

Graphical representation of Time Complexity:



Conclusion:

From the aforementioned experiment, we may deduce that the Randomized algorithm occasionally, but not always, performs better than the Median of Medians approach.
The reason is that the worst-case time complexity of the random algorithm is O(n^2).
The Median of Medians algorithm, however, operates in linear time in both the worst and best scenarios. Median's Algorithm can ensure the worst-case O (n). When employing a random approach, we cannot guarantee that the worst-case result will be O(n), but we can predict the likelihood that it will be O(n^2).
Comparing median of median methods (median-of-3, median-of-5, and median-of-7) has to do with a good split, though. MOM 3 performs incredibly poorly. This is due to the fact that, when choosing 3-element blocks, at least half of the n/3 blocks have at least 2 elements that are greater than the median of all the components. In the worst scenario, this results in a 2n/3 split.
the formula for
T(n) i= T(n/3) + T(2n/3) + O(n).
In this instance, n/3 + 2n/3 = 1, hence in the **worst case it decreases to O(n log n).**
For this reason, blocks greater than or equal to 5 are always divided.
By dividing into 5-element blocks, a worst-case 70-30 split is guaranteed. According to the standard argument, if at least half of the medians in the n/5 blocks are greater than the median-of-medians, then at least half of the n/5 blocks have at least three elements (1/2 of 5) that are greater than the median-of-medians. This results in a 3n/10 split, meaning that in the worst case, the other partition is a 7n/10 split.
T(n) = T(n/5) + T(7n/10) + O is **the result has the worst case complexity of O(n).**
After splitting the input array with the median-of-median, say m, as the pivot element, we may obtain the following lower bounds on the number of elements bigger than m and the number of elements smaller than m if we divide the input into 7 groups rather than 5.
For components bigger than m, half of sublists with 7 elements or more have at least 4 of those bigger than m. We are, in fact, discounting the sublist that contains m and the last sublist, which has a maximum size of 7.

So there are at least 4 elements that are greater than m.
([1/2[n/7]]-2)>=[(2n/7)-8].
Similarly, there exist at least (2n /7) - 8 items that are smaller than m.
The runtime's recurrence relation is T(n)<= T(d n 7 e) + T(5n 7 + 8) + O. (n).
The worst-case running time for this is O(n)
Therefore, the overall size of the problem decreases to less than n for the case where the input was divided into sublists of size 5 or 7.

Division of labor:

The source code and experimentation part was accomplished by Aman Bheemisetty. The report and run time analysis was performed by Rudranshi Dave.

Learning and Issues:

Median of Medians Algorithm can guarantee worst case O(n). But specifically in Median of Median with division of blocks with 3 elements reduces to O(n log n) in the worst case. It is recommended to divide the blocks equal or greater than 5. When using Randomized algorithm we can't guarantee worst case O(n), but the probability of the algorithm goes to O(n^2) .Few of the issues encountered was generation of dataset ,understanding the logic behind randomized median and practical application of median of median algorithm.

References:

- Introduction to Algorithms, 3rd Edition (The MIT Press) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.
- https://iq.opengenus.org/median-of-medians/
- https://nh2.me/recent/Quickselect-with-median-of-medians.pdf
- https://math.mit.edu/~goemans/18310S15/rand_median_quicksort-notes.pdf