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

Mary Sony Telugu(1002066593)

Nikhil Sai Chintagunti(1002071823)

Under the Guidance of

Md Hasanuzzaman Noor
(Professor)

&

Rutuja Ghadage (**GTA**)

Academic Session: Fall 22

The University of Texas, Arlington

Department of Computer Science and Engineering

Abstract

An unsorted list of size n can be effectively computed using a divide-and-conquer approach using median-finding algorithms (also known as linear-time selection algorithms), where i is an integer between 1 and n Selection algorithms are frequently used into other algorithms; for instance, they aid in choosing a pivot in quicksort and also assist identify the maximum, minimum, and median entries in a list are examples of i th Order statistics.

The median-of-medians algorithm is a deterministic linear-time selection algorithm. The algorithm works by dividing a list into sublists and then determines the approximate median in each of the sublists. Then, it takes those medians and puts them into a list and finds the median of that list. It uses that median value as a **pivot** and compares other elements of the list against the pivot. If an element is less than the pivot value, the element is placed to the left of the pivot, and if the element has a value greater than the pivot, it is placed to the right. The algorithm recurses on the list, honing in on the value it is looking for.

Randomized algorithm uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity from O(nlogn) to O(n) , with a worst case of O(n^2) .

In this project , We analyze four algorithms namely median of medians (median-of-3, median-of-5, and median-of-7) algorithm and the randomized median finding algorithm and compare their performances.

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):

The approximate median determined by the median-of-medians technique is a position that is assured to be between the 33.33th and 66.66th percentiles (in the middle 4 deciles). Thus, the search set has been reduced by at least 33.33%. The problem is reduced proportionately to 66.66% of its original size. The cost of repeating the same process until only one or two elements are left in the now-smaller collection is n / 1-0.9, or around 10.

Subroutine pivot is the actual median-of-medians procedure. The input is split into groups of three pieces initially, and a program then computes the median value for each of those groups. The real median of the n/3 medians found in the preceding step is then determined recursively.

Median of median(median-of-5):

The approximate median determined by the median-of-medians technique is a position that is assured to be between the 30th and 70th percentiles (in the middle 4 deciles). As a result, the search set has been reduced by at least 30%. The issue is reduced proportionately to 66.66% of its original size. n / 1-0.7, or roughly 3.33, is the cost of repeating the same method on the now-smaller collection until only one or two pieces are left.

Subroutine pivot is the actual median-of-medians procedure. The input is initially divided into groups of five pieces, and each group's median is then determined via a function. The real median of the n/5 medians found in the preceding step is then determined recursively.

Median of median(median-of-7):

Subroutine pivot is the actual median-of-medians procedure. The input is initially divided into groups of seven pieces, and each group's median is then determined via a function. The real median of the n/7 medians found in the preceding step is then determined recursively.

Half of the n/7 groups have at least 4 elements greater than the pivot. Omit the group containing the pivot and the group of n mod 7 elements. The number of elements greater than the pivot is at least: $4([1/2\,[n/7]]-2) \geq [(2n/7) - 8]$

Randomized median finding :

A natural approach is to select a pivot element, p, calculate its rank, and then divide the list into two sublists: one with entries smaller than p, the other with elements bigger than p. The probabilistic approach we now provide is simple. Simply select one entry from the list which was always chosen at random to be the pivot point. Applying this is quite simple. In the worst-case situation, we could do almost Cn2 comparisons since we had bad luck at every turn, but this is a highly unusual event. Randomized algorithm uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity from O(nlogn) to O(n) , with a worst case of O(n^2) .

Implementation:

- Python 3.x+

Libraries used:

- Python time library
- Python numpy library
- Python matplotlib.pyplot library
- Python pandas library

Code:

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

```
import time
import numpy as np
from numpy import random
```

```
def kthSmallest(arr, 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(arr, l + i * 3, n % 3))
            i += 1
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = kthSmallest(median, 0, i - 1, i // 2)
        pos = partition(arr, l, r, medOfMed)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return kthSmallest(arr, l, pos - 1, k)
        return kthSmallest(arr, pos + 1, r,
                           k - pos + l - 1)
    return 999999999999
```

```python
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp
def partition(arr, l, r, x):
    for i in range(l, r):
        if arr[i] == x:
            swap(arr, r, i)
            break
    x = arr[r]
    i = l
    for j in range(l, r):...
    swap(arr, i, r)
    return i
def findMedian(arr, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()
    return lis[n // 2]
```

```python
if __name__ == '__main__':
    start_time = time.time()
    # arr = random.randint(1000, size=(500))
    arr = [239, 197, 946, 232, 918, 744, 199, 51, 39, 408, 677, 975, 90, 130, 442, 254, 267, 359, 949, 98, 106, 304, 390, 7
    n = len(arr)
    print("array-", arr)
    k = 4
    print("Median element is", kthSmallest(arr, 0, n - 1, n // 2))
    print("--- %s seconds ---" % (time.time() - start_time))
```

Running time :

Best Case Time Complexity: O(n)

Worst Case Time Complexity: O(nlogn)

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

```
import time
import numpy as np
from numpy import random
```

```
def kthSmallest(arr, 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(arr, l + i * 5, 5))
            i += 1
        if (i * 5 < n):
            median.append(findMedian(arr, l + i * 5, n % 5))
            i += 1
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = kthSmallest(median, 0, i - 1, i // 2)
        pos = partition(arr, l, r, medOfMed)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return kthSmallest(arr, l, pos - 1, k)
        return kthSmallest(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(arr, l, r, x):
    for i in range(l, r):
        if arr[i] == x:
            swap(arr, r, i)
            break
    x = arr[r]
    i = l
    for j in range(l, r):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, r)
    return i
def findMedian(arr, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()
    return lis[n // 2]
```

```
if __name__ == '__main__':
    # arr = random.randint(100, size=(10))
    start_time = time.time()
    arr = [239, 197, 946, 232, 918, 744, 199, 51, 39, 408, 677, 975, 90, 130, 442, 254, 267, 35
    n = len(arr)
    print("array-", arr)
    print("Median element is", kthSmallest(arr, 0, n - 1, n // 2))
    print("--- %s seconds ---" % (time.time() - start_time))
```

Running time :O(n)

Best Case Time complexity :O(n)

Best Case Time complexity :O(n)

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

```
import time
import numpy as np
from numpy import random
import random
```

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

```python
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp
def partition(arr, l, r, x):
    for i in range(l, r):
        if arr[i] == x:
            swap(arr, r, i)
            break
    x = arr[r]
    i = l
    for j in range(l, r):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, r)
    return i
def findMedian(arr, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()
    return lis[n // 2]
```

```python
if __name__ == '__main__':
    start_time = time.time()
    # arr = random.randint(100, size=(10))
    arr = [239, 197, 946, 232, 918, 744, 199, 51, 39, 408, 677, 975, 90, 130, 442, 254, 267, 359, 94
    n = len(arr)
    print("array-",arr)
    print("Median element is", kthSmallest(arr, 0, n - 1, n // 2))
    print("--- %s seconds ---" % (time.time() - start_time))
```

Running time :O(n)

Best Case Time Complexity :O(n)

Worst Case Time Complexity : O(n)

Randomized median(RA) :

```
import time
import sys
import math
import random
```

```
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]
```

```
if __name__ == "__main__":
    start_time = time.time()
    arr = [239, 197, 946, 232, 918, 744, 199, 51, 39, 408, 677, 975, 90, 130, 442, 254, 267, 359, 949, 9
    random_input = random.sample(DOMAIN, NUM_ELEMENTS)
    print("Median:", find_median_randomized(arr))
    print("--- %s seconds ---" % (time.time() - start_time))
```

Running time :

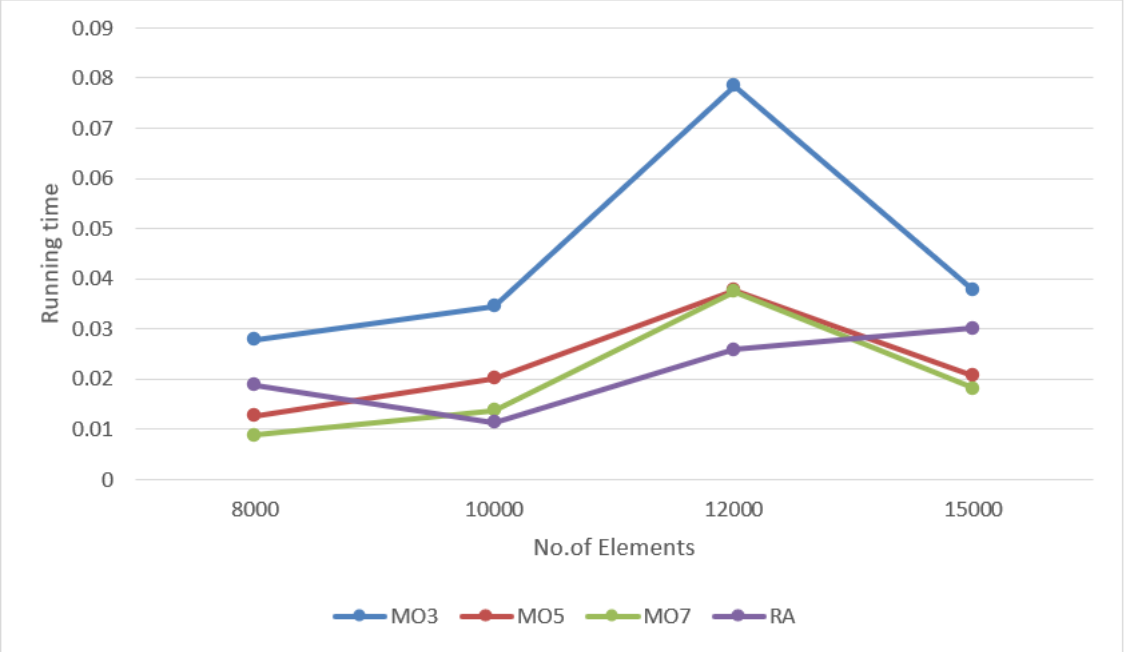
Best Case Time Complexity : O(n)

Worst Case Time Complexity : O(n^2)

Running Time as per code executed:

Number of Elements	MO3	MO5	MO7	RA
8000	0.027846813	0.01267075	0.00872898	0.01873493
10000	0.034427	0.0201146	0.0136857	0.01134
12000	0.07840466	0.037651300	0.037441968	0.0258765
15000	0.0377197	0.0206918	0.018127	0.0301003

Graphical representation of Time Complexity:



Time Complexity

Randomized Algorithm

Like quicksort, the randomized algorithm has good average performance, but is sensitive to the pivot that is chosen. If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, then the search set decreases in size exponentially and by induction (or summing the geometric series) one sees that performance is **linear**, as each step is linear and the overall time is a constant times this (depending on how quickly the search set reduces). However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case **performance is quadratic:** $O(n^2)$**.** This occurs for example in searching for the maximum element of a set, using the first element as the pivot, and having sorted data.

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)

This algorithm runs in ***O(n)*** linear time complexity, we traverse the list once to find medians in sublists and another time to find the true median to be used as a pivot. 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

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)

Conclusion:

From above experiment we can observe that Randomized algorithm has better performance compared to Median of Medians algorithm few times but not all the times.

Its because, Randomised Algorithm **has worst case time complexity O(n^2).**Where as Median of Medians algorithm runs in linear time both in worst and best cases.
Median of Medians Algorithm can guarantee worst case O(n). When using Randomized algorithm we can't guarantee worst case O(n), but the probability of the algorithm going to O(n^2) .

On the other hand, comparing median of medians (median-of-3, median-of-5, and median-of-7) algorithms it has to do with good split. MOM 3 has very bad performance. Its because when choosing 3-element blocks, at least half of the n/3 blocks have at least 2 elements >= median-of-medians, hence this gives a n/3 split, or 2n/3 in the worst case.
That gives T(n) = T(n/3) + T(2n/3) + O(n).
In this case, n/3 + 2n/3 = 1, so it **reduces to *O(n log n)* in the worst case.**

That's the reason it is always chose to divide blocks greater than or equal to 5.

Dividing into 5-element blocks assures a worst-case split of 70-30. The standard argument goes like this: of the n/5 blocks, at least half of the medians are >= the median-of-medians, hence at least half of the n/5 blocks have at least 3 elements (1/2 of 5) >= median-of-medians, and this gives a 3n/10 split, which means the other partition is 7n/10 in the worst case.That gives T(n) = T(n/5) + T(7n/10) + O(n).

Since n/5 + 7n/10 < 1**, the worst-case running time is *O(n)*.**

If we divide the input into 7 groups instead of 5, after partitioning the input array with the median-of-median, say $m*$, as the pivot element, we can get a lower bound on the number of elements that are greater than $m*$ and a lower bound on the number of elements smaller than $m*$ as follows.

For elements greater than $m*$, half of sublists consisting of 7 elements has at least 4 elements that are greater than $m*$. We here are indeed discounting the sublist containing $m*$ and the final sublist which has a size at most 7.

Thus number of elements greater than $m*$ is at least 4.([1/2[n/7]]-2)>=[(2n/7)-8].

Similarly, the number of elements that are smaller than $m*$ is also at least $(2n/7) - 8$.

Recurrence relation for the runtime becomes $T(n) \leq T(d\ n\ 7\ e) + T(\ 5n\ 7 + 8) + O(n)$.For this the **worst case running time is O(n)**

Therefor for the case where we divided the input into sublists of size 5 or 7, the total size of the problem reduces to less than n.

Learnings

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) .

Problems encountered

The creation of the dataset, comprehending the reasoning behind the randomized median, and the actual use of the median of median method were a few of the problems encountered.

Work Distribution

Nikhil Sai Chintagunti -Implemented Randomization Algorithm

Mary Sony Telugu - Implemented Median of Medians Algorithm(Grouping of 3,5,7)

Together we plotted the running times for different sizes of inputs and analyzed the time complexity. Studied approaches and understood how these algorithms work.

References:

- Introduction to Algorithms, 3rd Edition (The MIT Press) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.
- https://graphics.stanford.edu/courses/cs161-18-winter/Recitations/section3win2018_withsols.pdf
- https://www.cs.jhu.edu/~xfliu/600.363_F03/hw_solution/solution3.pdf
- https://en.wikipedia.org/wiki/Quickselect
- https://brilliant.org/wiki/median-finding-algorithm/
- https://iq.opengenus.org/median-of-medians/
- https://nh2.me/recent/Quickselect-with-median-of-medians.pdf
- https://en.wikipedia.org/wiki/Median_of_medians