CS648A - Randomized algorithms

Finding Median of 10^{10} numbers in practice

Authors

Anuj - 210166
Goutam Das - 210394

March 31, 2024

Contents

1 Abstract

The problem that we are aiming to solve is to find median of a large dataset of numbers (around 10^{10} numbers) such that the data cannot fit inside the RAM completely and thus we need to make passes over the data to find the "exact" median.

We design two 2-pass randomized-algorithms for this problem (later referred as Algorithm 1 and Algorithm 2) which are both Monte-Carlo and compute the exact median with high probability (inverse polynomial in input size) and use only $O(n^{2/3} \log n)$ space. The idea behind both the algorithms was similar, however, one uses complex data structures (Algorithm 2) whereas the other (Algorithm 1) just uses an array, this changes their running time in practice and we compare the results obtained from them.

Also, we designed another algorithm (Algorithm 3), which assumes that the input is uniformly randomly ordered, in which case, it finds the median in a single pass with high probability and using only $O(\sqrt{n} \log n)$ space.

Both the 2-pass algorithms can easily be converted to Las Vegas algorithms by repetition as at the end of each algorithm, we also get to know whether we have found the median or not.

Through this report, we present the designed algorithms along with their analysis as well as the empirical results we have obtained and their comparison. Also, we compare the algorithms' performance for smaller datasets as well and compare it with the well known quick median algorithm's implementation given in the Standard Template Library for a comparative analysis.

2 Algorithms

2.1 Algorithm 1

A 2-pass randomized algorithm using $n^{2/3} \log n$ space.
The main idea behind this algorithm is that, in one pass, we can get a uniformly random sample of the inputs of a smaller size that can fit inside RAM, then we can find two elements from this sample (one slightly to the left and one to the right of sample's median) such that two conditions hold

1. With high probability, the number of elements in the whole array that fall between the two chosen elements can fit inside RAM
2. With high probability, the median of original array lies between the two chosen elements

Once we get the two elements, we perform a second pass to store all the numbers between the 2 chosen "pivot" elements and simultaneously compute the rank of the left pivot. Then at the end of this pass, with high probability, we will have the median of the numbers stored and we can find it using the rank of left pivot.

2.1.1 Algorithm

Algorithm 1 Find Median in Two Passes

```
1: procedure ALGORITHM_1(file, n)
2:     sample_size ← n^{2/3} × log(n)
3:     sample ← GetSample(sample_size, file)
4:     offset ← n^{1/3} × log(n)
5:     left_pivot_index ← sample_size/2 − offset
6:     right_pivot_index ← sample_size/2 + offset
7:     left_pivot ← element at left_pivot_index in sample // compute using Quick-select algorithm
8:     right_pivot ← element at right_pivot_index in sample
9:     middle_elements ← Array[sample_size]
10:    count ← 0
11:    rank_left_pivot ← 0
12:    while Performing second pass through file do
13:        num ← ReadNextNumber(file)
14:        if num ≥ left_pivot and num ≤ right_pivot then
15:            middle_elements[count] ← num
16:            count ← count + 1
17:            if count == sample_size then
18:                Print "too many elements between pivots, algorithm failed"
19:                return -1
20:            end if
21:        end if
22:        if num < left_pivot then
23:            rank_left_pivot ← rank_left_pivot + 1
24:        end if
25:    end while
26:    if n/2 − rank_left_pivot < 0 or n/2 − rank_left_pivot ≥ count then
27:        Print "Median does not lie in the range, algorithm failed"
28:        return -1
29:    end if
30:    median ← element at (n/2 − rank_left_pivot) in middle_elements // using Quick-select algorithm
31:    return median
32: end procedure
```

2.1.2 Analysis

Now we will present a formal analysis on the error probability of algorithm 1.

- **Error bound on number of elements between left and right pivot**
  We are choosing the elements in sample with sampling probability $p = n^{-1/3} \log n$
  We know that there are $2n^{1/3} \log n$ elements between left and right pivot in sample.
  Let us first define random variables X to be number of elements between left and right pivot in original array and Y to be number of elements between pivots in sample.
  Then we can show

$$P(X \geq x_0 | Y = y_o) = P(Y \leq y_0 | X = x_0)$$

The idea here is quite similar to the proof of quick sort we did in class, we can again construct a new experiment in which the two events are same.
Choosing $x_0 = n^{2/3}\log n$ and $y_0 = 2n^{1/3}\log n$ (based on our sample)

$$P(X \geq n^{2/3}\log n) \leq P(\text{less than } 2n^{1/3}\log n \text{ sampled from } n^{2/3}\log n \text{ elements})$$

The expected number of elements sampled from $n^{2/3}\log n$ elements is, using linearity of expectation

$$\mu = p \times n^{2/3}\log n = n^{1/3}\log^2 n$$

$$P(Y \leq 2n^{1/3}\log n) = P\left(Y \leq \frac{2\mu}{\log n}\right) = P\left(Y \leq \mu\left(1 - \left(1 - \frac{2}{\log n}\right)\right)\right)$$

Using Chernoff bound, with $\delta = \left(1 - \frac{2}{\log n}\right)$, we know each element is sampled independently.

$$\begin{aligned} P(Y \leq 2n^{1/3}\log n) &\leq \exp\left(-\frac{1}{2}\mu\delta^2\right) \\ &\leq \exp\left(-\frac{1}{2}n^{1/3}\log^2 n\left(1 - \frac{2}{\log n}\right)^2\right) \end{aligned}$$

Using $\log n > 2$ for large n,

$$\begin{aligned} &\leq \exp\left(-\frac{1}{2}n^{1/3}\log^2 n\left(\frac{1}{\log n}\right)^2\right) \\ &\leq \exp\left(-\frac{1}{2}n^{1/3}\right) \\ &\leq e^{-n^{1/3}/2} \\ P(X \geq n^{2/3}\log n) &\leq e^{-n^{1/3}/2} \end{aligned}$$

Thus we have shown that this probability is actually inverse exponential in n

- **Bound on probability that true median does not lie between left and right pivot**
Let E be event that true median doesn't lie between left and right pivot
Then either rank of left pivot is greater than $n/2$ or rank of right pivot is less than $n/2$, by symmetry we can write

$$P(E) \leq 2P(\text{rank of left pivot is greater than } n/2)$$

We know that there are $(n^{2/3}\log n)/2 - n^{1/3}\log n)$ elements smaller than left pivot in sample, thus

$$P(E) \leq 2P\left(\text{less than } \left(\frac{n^{2/3}\log n}{2} - n^{1/3}\log n\right) \text{ sampled from } \frac{n}{2} \text{ elements}\right)$$

Similar to above part, the expected number of elements sampled are

$$\mu = p \times \frac{n}{2} = \frac{n^{2/3}\log n}{2}$$

Let Y again be the number of elements sampled from first $n/2$ elements

$$\begin{aligned}
P(E) &\leq 2P\left(Y \leq \left(\frac{n^{2/3}\log n}{2} - n^{1/3}\log n\right)\right) \\
&\leq 2P\left(Y \leq \mu\left(1 - 2n^{-1/3}\right)\right) \\
\text{Again with } \delta &= 2n^{-1/3}\log n \text{ , Applying Chernoff Bound} \\
P(E) &\leq 2\exp\left(-\frac{1}{2}\mu\delta^2\right) \\
&\leq 2\exp\left(-\frac{1}{2}\frac{n^{2/3}\log n}{2}\left(2n^{-1/3}\right)^2\right) \\
&\leq 2\exp\left(-\log n\right) \\
&\leq \frac{2}{n}
\end{aligned}$$

Thus, the probability here is also inverse polynomial in n, and it can be further boosted to n^{-c} by changing sampling probability by a constant factor

2.2 Algorithm 2

Another 2-pass randomized algorithm using $n^{2/3}\log n$ space
This algorithm is slightly different from Alggorithm 1 as here, rather than choosing 2 pivots, in the second pass we choose the elements that are adjacent to the sample median. Formally, we store the smallest $n^{2/3}\log n$ elements that are greater than the sample median and similarly store the largest $n^{2/3}\log n$ elements that are smaller than sample median and simultaneously compute the rank of sample median. Then, we can claim, as we prove formally in analysis, that the actual median will lie in either of the 2 sets with high probability.

2.2.1 Algorithm

Algorithm 2 Find Median using Multisets

```
1: procedure ALGORITHM_2(file, n)
2:     sample_size ← n^{2/3} × log(n)
3:     range_size ← n^{2/3} × log(n)
4:     sample ← GetSample(sample_size, file)
5:     sample_median ← median(sample) // using quick-median
6:     small_nums ← new multiset()
7:     large_nums ← new multiset()
8:     rank_median ← 0
9:     while second pass through file do
10:        num ← ReadNextNumber(file)
11:        if num < med_num and num > min(small_nums) then
12:            rank_median ← rank_median + 1
13:        end if
14:        if num < med_num and num > min(small_nums) then
15:            insert num to small_nums
16:            if size(small_nums) > range_size then
17:                delete min(small_nums) from small_nums
18:            end if
19:        end if
20:        if num >= med_num and num < max(large_nums) then
21:            insert num to large_nums
22:            if size(large_nums) > range_size then
23:                delete max(large_nums) from large_nums
24:            end if
25:        end if
26:    end while
27:    if rank_median <= n/2 and rank_median + size(large_nums) > n/2 then
28:        median ← (n/2 − rank_median)^{th} smallest element from large_nums
29:    else if rank_median > n/2 and rank_median − size(small_nums) <= n/2 then
30:        median ← (rank_median − n/2)^{th} largest element from small_nums
31:    else
32:        Print "Median does not lie in range of sample median, algorithm failed"
33:        return -1
34:    end if
35:    return median
36: end procedure
```

2.2.2 Analysis

Now we will present a formal analysis on the error probability of algorithm 2
Here we only need to show **bound on probability that median does not lie in both the small and large sets** , let this be Event E For median to not lie in either sets, the rank of sample median should be either greater than $n/2 + set\_size$ or less than $n/2 - set\_size$

Again we will use symmetry and say that,

$$P(E) \leq 2P(\text{sample median has rank greater than } n/2 + set\_size)$$

i.e less than $(n^{2/3}\log n/2)$ elements sampled from first $(n/2 + n^{2/3}\log n)$ elements
Again using analysis similar to that of Algorithm 1
Let Y again be the number of elements sampled from first $n/2 + n^{2/3}\log n$ elements
The expected number of elements sampled are

$$\mu = p \times (n/2 + n^{2/3}\log n) = \frac{n^{2/3}\log n + 2n^{1/3}\log^2 n}{2}$$

$$\begin{aligned}
P(E) &\leq 2P\left(Y \leq \left(\frac{1}{2}n^{2/3}\log n\right)\right) \\
&\leq 2P\left(Y \leq \mu\left(1 - \frac{2\log n}{n^{1/3} + 2\log n}\right)\right) \\
\text{Again with } \delta &= \frac{2\log n}{n^{1/3} + 2\log n} \text{ , Applying Chernoff Bound} \\
P(E) &\leq 2\exp\left(-\frac{1}{2}\mu\delta^2\right) \\
&\leq 2\exp\left(-\frac{1}{2}\left(\frac{n^{2/3}\log n + n^{1/3}\log^2 n}{2}\right)\left(\frac{2\log n}{n^{1/3} + 2\log n}\right)^2\right) \\
\text{for sufficiently large } n & \\
&\leq 2\exp\left(-\frac{1}{2}\left(\frac{n^{2/3}\log n}{2}\right)\left(\frac{2}{n^{1/3}}\right)^2\right) \\
&\leq \frac{2}{n}
\end{aligned}$$

Thus, the error probability is inverse polynomial in n

2.3 Algorithm 3

A 1-pass algorithm for uniformly randomly ordered input, using $O(\sqrt{n}\log(n))$ space.
The idea here is to maintain a window of middle elements, by keeping 2 counters $left$ and $right$, and a set of the middle elements of size $window\_size(= \sqrt{n}\log(n))$, then during the first pass, we first fill the set with the initial elements, then once the set is filled, for each remaining element, 3 cases can arise

1. it is smaller than minimum of window, then we increment the left counter
2. it is bigger than maximum of window, then we increment the right counter
3. it lies between the min and max of window, in this case, we get further two cases
   - 3.1 left counter is less than right counter, then we remove min element from window and insert the current num and increment left counter
   - 3.2 otherwise, then we remove max element from window and insert the current num and increment right counter

Finally if the left and right counter are such that median lies between them, then we can find the median, and we show in analysis that this happens with high probability for a random input.
The formal pseudocode of the algorithm is given in next section.

2.3.1 Algorithm

Algorithm 3 Median Computation in 1 Pass

```
1: procedure ALGORITHM_3(file, n)
2:     window ← new multiset()                    ▷ Maintain the set of elements in the middle
3:     left ← 0, right ← 0
4:     window_size ← √n × log(n)
5:     for i ← 0 to window_size do                ▷ Read the first window_size elements
6:         num ← ReadNextNumber(file)
7:         window.insert(num)
8:     end for
9:     for i ← window_size to n do
10:        num ← ReadNextNumber(file)
11:        mn ← min(window)
12:        mx ← max(window)
13:        if curr ≤ mn then
14:            left ← left + 1
15:        else if curr ≥ mx then
16:            right ← right + 1
17:        else
18:            if left < right then
19:                delete mn from window          ▷ Remove the smallest element
20:                insert num to window
21:                left ← left + 1
22:            else
23:                delete mx from window          ▷ Remove the largest element
24:                insert num to window
25:                right ← right + 1
26:            end if
27:        end if
28:    end for
29:    index_needed ← n/2 − left
30:    if index_needed > window_size then
31:        print "One-pass Algorithm Failed"
32:        return -1
33:    else
34:        median ← index_needed^th rank element from window
35:        return median
36:    end if
37: end procedure
```

2.3.2 Analysis

Now we will present a formal analysis on the error probability of algorithm 3.
Let random variables l_i and r_i denote the value of left and right counters at index i
Define a new random variable $d_i = r_i - l_i$ Note that our algorithm will fail only if at the end $|d_n|$ is greater than $window\_size(= \sqrt{n}\log n)$
Now we analyse the probability that $|d_i|$ increases at i_{th} index, i.e. $|d_{i+1}| = |d_i| + 1$
Taking case when $d_i > 0$ and $d_i < window\_size$ (Symmetric case for $d_i < 0$)

$$P(d_{i+1} = d_i + 1 | d_i > 0, d_i < window\_size) = P(r_{i+1} = r_i + 1 | r_i < l_i + window\_size)$$

Consider all the $(r_i + l_i + window\_size)$ elements that are seen till now in sorted order, since the stream is random order, the next element will lie between any 2 consecutive elements in these elements with equal probability
Further, r_i will only increase if the next element lies between $window\_size$ and any of the r_i elements, hence

$$P(r_{i+1} = r_i + 1) = \frac{r_i}{r_i + l_i + window\_size}$$

$$P(r_{i+1} = r_i + 1 | r_i < l_i + window\_size) = \frac{r_i}{r_i + l_i + window\_size} \leq \frac{1}{2}$$

Now, this is somewhat analogous to a random walk on a line problem, with probability $\leq 1/2$, the person goes away from origin, and otherwise moves closer to origin, hence

$$P(|d_i| > window\_size) \leq P(\text{person is at distance} > window\_size \text{ after } i \text{ steps in random walk})$$

This has already been shown in quiz but reiterating the proof for completeness,
Let X_i's be independent 0-1 random variables denoting if the person went right at ith step
Distance of man from origin is then given by

$$D_i = \left| 2 \times \left(\sum_{j=0}^{i} X_j \right) - i \right|$$

$$\begin{aligned} P(D_i > c\sqrt{n}\log n) &\leq 2P\left(2 \times \left(\sum_{j=0}^{i} X_j \right) - i > c\sqrt{n}\log n \right) \\ &\leq 2P\left(\sum_{j=0}^{i} X_j > \frac{i + c\sqrt{n}\log n}{2} \right) \end{aligned}$$

Now probability that algorithm fails is

$$P(Error) \leq \bigcup_{i=1}^{n} P(D_i > c\sqrt{n}\log n)$$

Using Union Bound

$$\leq \sum_{i=1}^{n} P(D_i > c\sqrt{n}\log n)$$
$$\leq n \times P(D_n > c\sqrt{n}\log n)$$
$$\leq 2n \times P\left(\sum_{i=0}^{n} X_i > \frac{n + c\sqrt{n}\log n}{2}\right)$$

Using Chernoff bound

$$\mu = \frac{n}{2}$$
$$\delta = \frac{c\log n}{\sqrt{n}}$$
$$P(Error) \leq 2n\exp\left(-\frac{1}{4}\mu\delta^2\right)$$
$$\leq 2n\exp\left(-\frac{n}{8}\left(\frac{c\log n}{\sqrt{n}}\right)^2\right)$$
$$\leq 2n\exp\left(-\frac{(\log n)^2}{8}\right)$$

For large n, log n > 16

$$P(Error) \leq 2n\exp(-2\log n)$$
$$\leq \frac{2}{n}$$

So, the error probability is atleast inverse polynomial in n

3 Implementation Details

3.1 Code Organization

The code of project is structured as follows:

```
project
|-util.hpp, util.cpp (header files for helper functions, used in implementation)
|-algo1.cpp, algo2.cpp, algo3.cpp (implementation of algorithms described above)
|-algo4.cpp (quick-median algorithm from Standard template library)
|-data.cpp (code to generate datasets)
|-makefile (script to make executables from the above files)
|-test.sh (script to test the algorithms and output observations.csv)
```

The project's code can also be found in the git repository: https://github.com/AnujSinghal21/CS648-project
To run the code locally, clone this git repository, or extract files from the submitted zip file, and run `make all` on any UNIX based system.

3.2 Key Features in Implementation

3.2.1 Generating and reading data

- The data is written in a file named `data.bin`, which is a binary file for optimized storage (if we wrote the numbers as text, converting numbers to text and then scanning was really slow), writing in binary got us a speed up of **around 20x**
- We have implemented a "reader" class in the `util.cpp` file that allows sequential reading of data from a file, it implements a function `next()` that returns the next number in file
- This class also implements a read buffer that reads data from file in chunks to optimize reading time, this gives additional speed up of **around 5-7x**

3.2.2 Generating random numbers

- We started with using the C language's `rand()` function to generate random numbers, but the algorithms were getting huge error
- It was really confusing for us to find where the bug lies and it turns out that the `rand()` function is not reliably uniformly random for generating stream of random numbers
- After some research, we found that the right way to do it something known as the Mersanne Twister pseudo-random number generator, implemented in `C++`'s `random` library as `mt19937_64` class

3.2.3 Algorithms

- The data structures and algorithms used in implementing the above algorithms have all been taken from Standard Template Library for best optimized implementation and easy to understand code
- We tried to do some testing by implementing the data structures ourselves but the library's inbuilt functions were highly optimized and worked around 1.5x faster than even our best implementations

3.2.4 The bottleneck of memory

- Though our device's RAM was capable of running the algorithms even for $n = 10^{11}$, the window's anti-malware system used to kill our program whenever it tried to take around 1GB RAM, for this, we had to shift to virtual box and linux environment, though our familiarity with OS and linux was really helpful.
- The only thing that stopped us from reaching the earlier decided 10^{11} numbers objective was our device's hard disk!, as that would have required 800 GB's of storage, which none of our systems had, though we ran the algorithm on what was best feasible on our systems at $n = 5 \times 10^{10}$.

4 Experimental results

4.1 Running time of algorithms on 10^{10} numbers

When running such large processes, the time can be measured in 2 ways, the CPU time that measures how much time of computation the process actually took, this is the more reliable metric for running time as the operating system keeps switching between the processes and doesn't run a single process continuously till completion.
The other way to compute time which we are calling the apparent running time is how much time the program

actually took to end from the time it is started, this can very a lot based on how many other programs are open on system, what is the battery level, etc., however this is more practical to look at from a common person's perspective.

Table 1: Execution Time of Algorithms

Algorithm	CPU running time	Apparent Running time
Algorithm 1	141.8 seconds	16 minutes 18 seconds
Algorithm 2	4486.7 seconds	3 hour 36 minutes
Algorithm 3	119.4 seconds	14 minutes 17 seconds

4.2 Analysing performance on smaller input size

We took readings of average running time of algorithms and compare it with the quick median algorithm, here we chose datasets of sizes 1000, 10000, 100000, 1000000, 10000000 and 100000000, since quick median can only work till a size of 10^8 (RAM limitation) and computing time for sizes below 1000 was very unreliable since it is very negligible.
The results are tabulated as follows

Table 2: Average Execution Time of Algorithms (in ms)

Size	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
1000	0.17	0.34	0.18	0.12
10000	0.45	2.25	0.55	0.26
100000	2.52	20.19	2.87	1.47
1000000	17.24	246.72	17.99	10.91
10000000	149.51	3850.67	139.02	107.09
100000000	1443.16	42885.56	1469.84	1131.67

* Note that the time shown above is the CPU time, which is the time for which CPU dedicatedly ran the process, the actual running time was around 2 minutes for algorithm 1 on size 10^8
We also plot the results for better visualization

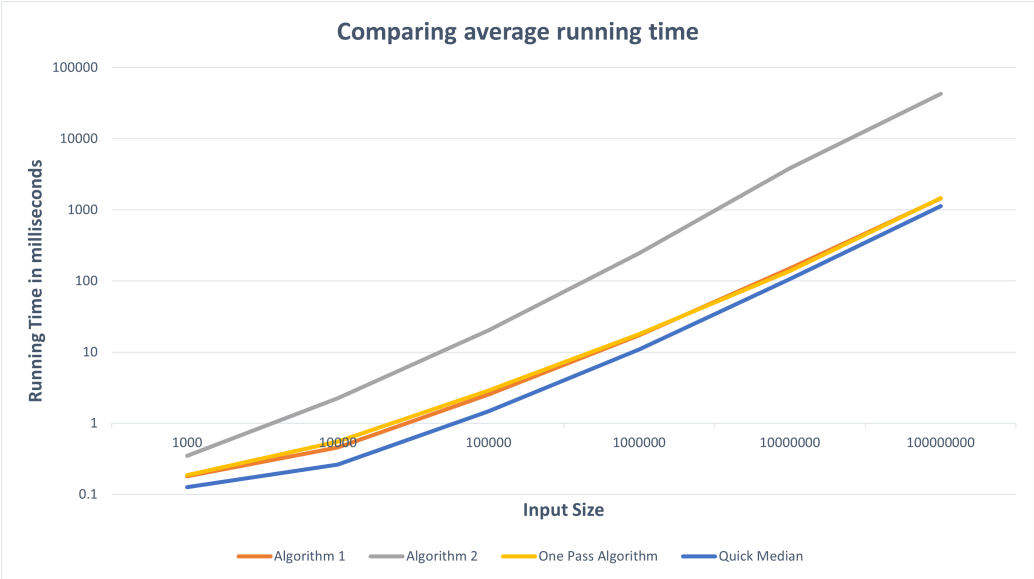

Figure 1: Average Execution Time of Algorithms

4.2.1 Inferences

- First of all we see that the running time of algorithm 1 and algorithm 3 is very much comparable to that of quick median.
- Also, we see the impact of the complexity data structures used very nicely here, the only reason that algorithm 2 is around 25 times slower is because it requires a multiset which utilizes binary search tree in its implementation.
- Algorithm 3 also has the same problem that even though it takes only one pass, its running time is close to algorithm 1 since it also uses multiset, whereas algorithm 1 deals with just an array.
- This shows that simplicity in algorithms can be so useful in practice.

4.3 Robustness of the algorithms

The robustness of the designed randomized algorithms has been demonstrated through extensive empirical testing. While the probability analysis suggests a negligible likelihood of errors for large input sizes, our empirical observations corroborate these findings. Throughout our testing on various datasets of considerable size, we have yet to encounter any instances of errors attributable to the algorithms' inherent randomness. This is supported by our analysis that the chances of getting an error for a dataset of even 10^5 size is 1 in 100000. Given the number of times we have tested, we can only say that we have not been that lucky (or rather unlucky) enough to witness such a rare event!

Moreover, even on smaller input sizes of around 1000 where the probability of error is slightly elevated, our experiments have revealed only isolated occurrences of deviations from the expected outcomes. These rare occurrences underscore the resilience and reliability of the algorithms, reaffirming their efficacy in practical scenarios.

5 Acknowledgement

We would like to express our sincere gratitude to our Professor for their invaluable guidance and insightful suggestions throughout the course of this project. Their expertise and constructive feedback have greatly contributed to the development and refinement of our work. The key idea for algorithm 1 which can be seen is working much faster than algorithm 2 was provided as hint by our professor. Furthermore, we affirm that no other external sources have been utilized in the creation of this project, except for some research on implementation topics. Overall this project was a great source of learning for us and we feel testing the algorithms gave us an actual feel of the challenges that researchers might face in developing new algorithms.