A New $O(\sqrt{n})$ Method for the Range Mode Query in Arrays

Younan Gao/B00791197

1 Introduction

Given an array A of n elements, the statistic mode is the most frequent element, which occurs at least as frequently as any other elements in the array A. Range mode query seeks to compute the mode on the subarray $A[i, j]$. An array could contain multiple modes, but their occurrence is unique. Along with the mean and median, the mode is another fundamental data statistic in data analysis as expressed by Chan et al. [1]. The solutions covered in this paper are to construct a static data structure which supports efficient range mode query. Based on the work by Chan et al. [1], I present two new methods (in Sections 4 and 5) to solve the range mode query problem using a linear space data structure in $O(m)$ and $O(\sqrt{n})$ query time respectively. Furthermore, I implemented two $O(\sqrt{n})$ methods (one of them is from Chan et al. [1]) and make performance comparison on space and query efficiency between the two methods in practice.

2 Related Work

The range mode query problem was started by Krizanc et al. [5]. They provided an useful observation and gave a solution with fast queries, achieving linear space and $O(\sqrt{n}\log n)$ query time. Afterwards, based on that useful observation, a number of efficient query solutions using linear space are proposed by researchers. The first solution achieving $O(n)$ space and $O(\sqrt{n})$ query time is found by Durocher et al. [2]. Chan et al. [1] improved the query time to $O(\sqrt{\frac{n}{\log n}})$ with the knowledge from the world of succinct data structure. Meanwhile, Chan et al. [1] gave a strong evidence to indicate that a query time significantly below $\sqrt{n}$ can not be reached by purely combinatorial techniques by reducing the multiplication of two $\sqrt{n} \times \sqrt{n}$ boolean matrices to the mode range query problem using linear space. The lower bound of the range mode query problem was proved by Greve et al. [4] in

the cell probe model, which is for any data structure that uses S memory cells of w-bit words, it takes $\Omega(\frac{\log n}{\log(\frac{S \times w}{n})})$ query time to answer the range mode query problem. For linear space data structures in the RAM model, it corresponds to a lower bound of $\Omega(\frac{\log n}{\log \log n})$ query time, according to Durocher et al. [2]. However, with current knowledge, there is no solution achieving the lower bound. My work so far mainly focus on linear space solutions to the static range mode query problem under the RAM model. See the Figure 1 for the information of the other solutions found so far.

Query Time	Update Time	Space	Remark	Reference
$O(n^{2/3})$	$O(n^{2/3})$	$O(n)$	Dynamic	[3]
$O(n^{3/4}lglgn)$	$O(n^{3/4}lglgn)$	$O(n)$	Dynamic	[1]
$O(n^{2/3}\frac{lgn}{lglgn})$	$O(n^{2/3}\frac{lgn}{lglgn})$	$O(n^{4/3})$	Dynamic	[1]
$O(\sqrt{\frac{n}{lgn}})$	-	$O(n)$	-	[1]
$O(n^t)$	-	$O(n^{2-2t})$	$0 < t \leq \frac{1}{2}$	[2]
$O(\Delta)$	-	$O(n)$	-	[2]
$O(m)$	-	$O(n)$	-	[2]
$O(\|j-i\|)$	-	$O(1)$	-	[2]
$O(1)$	-	$O(n^2\frac{lglgn}{(lgn)^2})$	-	[6]
$O(n^t)$	-	$O(n^{2-2t})$	$0 \leq t < \frac{1}{2}$	[6]
$O(1)$	-	$O(\frac{n^2}{lgn})$	-	[6]
$O(n^tlgn)$	-	$O(n^{2-2t})$	$0 < t \leq \frac{1}{2}$	[5]
$O(1)$	-	$O(n^2\frac{lglgn}{lgn})$	-	[5]

Figure 1: The upper bound of the known solutions for both the static and dynamic range mode query problem

3 Preliminaries

Before discussing the methods to solve the range mode problem, I introduce the notations used in the following sections. Throughout the paper, let m denote the maximum frequency of the overall array, let s denote the number

of blocks, let t denote the number of elements in each block, let n denote the length of the overall array, and let Δ denote the total number of distinct elements. Both m and Δ are not larger than n.

Another concept used here is rank reduction. Rank reduction transfers an input array Arr[1:n] to A[1:n], where the element A[i] stores the rank of the element Arr[i]. Given an array $\{10, 20, 20, 10, 30, 30, 10, 40, 40\}$, after rank reduction it turns to be $\{1, 2, 2, 1, 3, 3, 1, 4, 4\}$. Rank Reduction on an array with length n takes $O(n \log \Delta)$ time using any balancing binary search tree data structure. I first apply rank reduction on the input array before querying the mode. It is obvious to tell that the mode of the array after rank reduction corresponds to the mode of the original array in the same range. Throughout the paper, let array $A[1\colon n]$ denote the input array after rank reduction.

4 First Method: $O(m)$ Query Time and $O(n)$ Space

I present a new method using linear-space data structure with $O(m)$ query time. The observation provided by Krizanc et al. [5] is useful.

Lemma 1 *Let A1 and A2 be any multisets. If c is a mode of A1 $\cup$ A2 and $c \notin$ A1, then c is a mode of A2*

How do we use this lemma to solve the range mode query problem? Suppose that the target range is separated into two parts, which are A1, and A2. The mode of the range A1 $\cup$ A2 is either the mode of the range A2 or an element in the range A1.

4.1 Data Structure Precomputation

Following Krizanc et al. [5], we partition array A into $\lceil s = n/m \rceil$ blocks of size $t = m$. Then the target range $[i, j]$ is partitioned into the either one of the following two scenarios: when $i \bmod t$ equals 0, we keep the original target range; Otherwise, it is separated into two parts, which are the prefix $[i,\ \lceil i/t \rceil \times t - 1]$, and the suffix $[\lceil i/t \rceil \times t,\ j]$. We precompute tables $Array\_F[0\colon s-1, 1\colon m]$ and $Array\_mode[0\colon s-1, 1\colon m]$ as following: for each $b_i \in \{0, \cdots, s-1\}$ and each $b_j \in \{1, \cdots, m\}$, $Array\_F[b_i][b_j]$ stores the smallest index $x \leq n$ such that the mode of the range $[b_i \times m, x]$ has frequency at most b_j. If the frequency of the mode could not reach b_j, then x is set to n. And $Array\_mode[b_i][b_j]$ stores the corresponding mode. See Figure 2 for an example.

To check whether each element in the prefix is a candidate mode, I use the same method introduced by Chan et al. [1]. Therefore, besides the

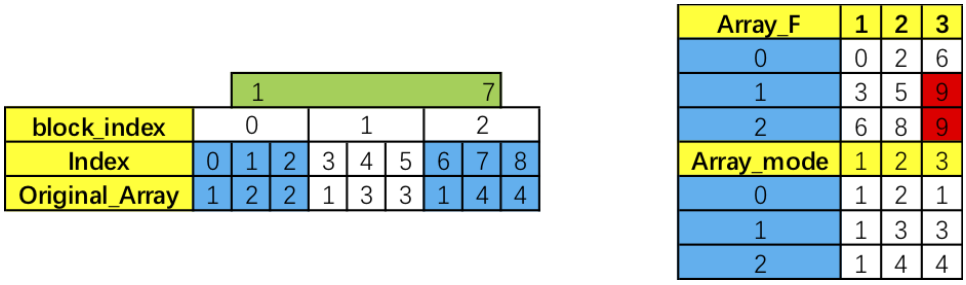

Figure 2: The input array A has $n = 9$ elements and $\Delta = 4$ distinct elements. The array A is partitioned into $s = (n/m) = 3$ blocks and each block has $t = m = 3$ elements. The target range is $A[1\colon 7]$, for which the mode is any of 1, 2, 3, occurring twice. The query range is separated into the prefix $A[1, 2]$ and the suffix $A[3, 7]$. The predecessor of 7 in the sub $Array\_F[\lfloor 3/t \rfloor]$ is 5 and its corresponding column index is $b_j = 2$. Then the mode of the suffix $A[3, 7]$ is stored at $Array\_mode[1][b_j = 2]$ (which is 3), occurring with frequency $b_j = 2$.

two tables $Array\_F$ and $Array\_mode$, another $\Delta + 1$ arrays are necessary. For each $a \in \{1, \cdots, \Delta\}$, let array $Q[a]$ store the set of indices b such that $A[b] = a$. The last one is array $A'[1\colon n]$, such that for each b, let $A'[b]$ store the index of b in array $Q[A[b]]$.

All of the data structure mentioned above are using O(n) space. The algorithm to construct $Array\_F$ and $Array\_mode$ can be summarized as follows:

Algorithm 1 Precompute Array_F and Array_mode

```
1: procedure PRECOMPUTE()
2:     Initialize Array_F[s_num_block][m_frequency + 1]
3:     Initialize Array_mode[s_num_block][m_frequency + 1]
4:     for s ← 0 to s_num_block − 1 do
5:         Initialize the array count_item[Δ]
6:         array_F[s][1] ← t_of_block × s
7:         array_mode[s][1] ← original_arr[t_of_block × s]
8:         index ← 2
9:         for i ← (t_of_block × s) to (original_arr.length − 1) do
10:            count_item[original_arr[i]] + +
11:            if count_item[original_arr[i]] = index then
12:                array_F[s][index] ← i
13:                array_mode[s][index] ← original_arr[i]
14:                index + +
15:        for j ← index to m_frequency do
16:            array_F[s][j] ← original_arr.length
17:            array_mode[s][j] ← array_mode[s][j − 1]
```

It is obvious to see that $Algorithm$ 1 takes $O(s \times m)$ time to precompute

$Array\_F$ and $Array\_mode$.

4.2 Query Algorithm

Suppose that the target range has been partitioned into the two parts prefix $[i, b_j - 1]$, and the suffix $[b_j, j]$. For the suffix part, we can find the predecessor of j in the sub array $Array\_F[\lfloor b_j/t \rfloor]$. Here, the predecessor means the leftmost maximum element in array $Array\_F[\lfloor b_j/t \rfloor]$ which is not larger than j. Let f denote the index of the predecessor in array $Array\_F[\lfloor b_j/t \rfloor]$. Then f represents the frequency of the mode in the suffix range and $Array\_mode[\lfloor b_j/t \rfloor][f]$ stores the corresponding mode. To find the predecessor, there are three ways. Linear scan takes $O(m)$ time in the worst case, regarding there are m elements in the array $Array\_F[\lfloor b_j/t \rfloor]$. Considering the array $Array\_F[\lfloor b_j/t \rfloor]$ is a monotonically increasing sequence, binary search also applies, which takes $O(\log m)$ time. If using advanced data structure "Van Emde Boas tree", it takes $O(\log \log n)$ time, because the universe of the array $Array\_F[\lfloor b_j/t \rfloor]$ is $\{1, \cdots, n\}$. For simplicity, I choose the linear scan method to find the predecessor. For the prefix part, I use the same method introduced by Chan et al.[1] to find the candidate mode in the prefix, which frequency in the range $[i, j]$ is larger than the frequency f of the mode in the suffix range. The query cost is bounded by $O(t = m)$ [1]. Putting the prefix and suffix together, the overall cost to find the mode in the range $[i, j]$ is $O(m)$.

The idea of the $O(m)$ query time method derives from the $O(m)$ query time method proposed by by Chan et al.[1]. However, the latter method partitions the target range into 3 parts, the prefix, the span, and the suffix. And the latter method needs to check both the prefix and suffix parts to find the candidate modes. Furthermore, the method introduced in this paper cuts off an array with size $s \times m$, which is used in the method from by Chan et al.[1] to store the corresponding frequency for each mode kept in the array $Array\_mode$.

5 Second Method: $O(\sqrt{n})$ Query Time and $O(n)$ Space

The idea of the second method is to combine the first method (in section 4) and the $O(\Delta)$ query method [1] together. Firstly, I define a threshold $\sqrt{n}$ to separate the input array $A[0, n-1]$ into two arrays $B0[0, n_0 - 1]$ and

[1]The $O(\Delta)$ query method is borrowed from Chan et al. [1]. It uses the trick of the prefix sum to implement range mode query in $O(\Delta)$ time using a linear data structure. Another advantage of this method is that it takes O(n) time to precompute the static data structure.

$B1[0, n_1 - 1]$ (Here, $n_0 + n_1 = n$). Let array $B0[0, n_0 - 1]$ store the elements which frequency is not larger than $\sqrt{n}$. And let array $B1[0, n_1 - 1]$ store the remain elements with frequency larger than $\sqrt{n}$. It is necessary to remind that immediately after partition array A into arrays $B0$ and $B1$, we have to do one more round of rank reduction on both array $B0$ and $B1$. Regarding there are no common elements between the arrays $B0$ and $B1$, we can apply mode query in the corresponding ranges on arrays $B0$ and $B1$ respectively. The one with the larger frequency is the final mode of the range $[i, j]$.

5.1 The methods applied on array $B0$ and $B1$

Based on the different data patterns of the elements stored between arrays $B0$ and $B1$, we can apply different methods to achieve high performance. See Figure 3 for the detail of the strategies.

Condition	B0		B1
B1.length ==0	O(|j-i|)	O(m)	O(△)
	X	√	X
B0.length ==0	O(|j-i|)	O(m)	O(△)
	X	X	√
B0.length <=s && B1.length > 0	O(|j-i|)	O(m)	O(△)
	√	X	√
Otherwise	O(|j-i|)	O(m)	O(△)
	X	√	√

Figure 3: The different strategies applied on array $B0$ and $B1$. The $O(|j - i|)$ method is a linear scan method to find the mode without precomputation introduced by Durocher et al. [2]. And the query cost is $O(|j - i|)$.

.

To analyze the query time of the third method, we will use the following lemma.

Lemma 2 *Suppose an array A with length n is partitioned into s blocks with size t in each block, if the frequency of each distinct element is more than s, then the number of distinct elements Δ in array A is less than t.*

Because of the pre-defined threshold $\sqrt{n}$, m in the array $B0$ is less than or equal to $\sqrt{n}$. In the array $B1$, the frequency of each distinct element is more than $\sqrt{n}$. According to the Lemma 2, Δ in array $B1$ must be less than n/s, which is $\sqrt{n}$. Therefore, the overall query cost is bounded by $O(\sqrt{n})$.

5.2 How to locate the corresponding ranges in $B0$ and $B1$

Once array A partitioned into arrays $B0$ and $B1$, the target range on array A is not appropriate on arrays $B0$ and $B1$ any more. We have to find the corresponding ranges on the new arrays. Chan et al. [1]. provides a linear space data structure and supports to find the corresponding ranges on arrays $B0$ and $B1$ in constant time. Here, I totally borrow the solution from Chan et al. [1]. But I will give more details to illustrate this solution. See Figure 4 for an example of how to use the precompute arrays I_0, J_0, I_1, and J_1 to find the corresponding ranges in arrays $B0$ and $B1$ respectively.

	0	1	2	3	4	5	6	7	8
I_0	0	1	1	2	2	3	4	4	-1
J_0	0	0	1	1	2	3	3	4	4
I_1	0	0	1	1	2	2	2	3	3
J_1	-1	0	0	1	1	1	2	2	3
A	2	7	2	7	1	5	7	2	7
B0	2		2		1	5		2	
B1		7		7			7		7

Figure 4: The original array A is $\{2, 7, 2, 7, 1, 5, 7, 2, 7\}$ without rank reduction. Then let array $B0$ be $\{2, 2, 1, 5, 2\}$ storing sparse elements and let array $B1$ be $\{7, 7, 7, 7\}$ storing the dense elements. The target range is [3, 8]. Its corresponding ranges are $[2, 4]$ in array $B0$ and $[1, 3]$ in array $B1$, respectively.

.

To partition array A into arrays $B0$ and $B1$ is trivial. I will focus on introducing how to construct the precompute arrays I_0, J_0, I_1, and J_1, among which, the arrays I_0, and J_0 are used to find the corresponding range in array $B0$ and the arrays I_1, and J_1 are used to find the corresponding range in array $B1$. At the same time when partitioning array A, it is necessary to construct array $Array\_P[0\colon 1][0\colon n-1]$, such that for each $i \in \{0, 1\}$ and each $j \in \{0, n-1\}$, the entry $Array\_P[i][j]$ stores the index of the element $A[j]$ in array Bi. Let the counter entry $Array\_P[1-i][j]$ of the entry $Array\_P[i][j]$ store -1. The algorithm to construct arrays I_0, and J_0 can be summarized as follows. We can use the same method to construct arrays I_1, and J_1 .

Algorithm 2 Precompute Array_I_0 and Array_J_0

```
1: procedure PRE_PARTITION_ARR(B[0:1][0:n−1], Array_P[0:1][0:n−1])
2:     for i ← 0 to A.length − 1 do
3:         if array_P[0][i] != −1 then
4:             array_I[0][i] ← array_P[0][i]
5:         else
6:             tmp_index ← −1
7:             for end_index ← i + 1 to A.length − 1 do
8:                 if array_P[0][end_index] != −1 then
9:                     tmp_index ← array_P[0][end_index]
10:                    end_index + +
11:                    break
12:            for j ← i to end_index − 1 do
13:                array_I[0][j] ← tmp_index
14:            i ← end_index − 1
15:
16:    for i ← 0 to A.length − 1 do
17:        if array_P[0][i] != −1 then
18:            array_J[0][i] ← array_P[0][i]
19:        else
20:            if i = 0 then
21:                array_J[0][i] ← −1
22:            else
23:                array_J[0][i] ← array_J[0][i − 1]
```

For the first *for loop* from line 2 to line 14, it takes $O(n)$ time, because each entry $array\_I[0][i]$ is set only once. For the second *for loop* from line 16 to line 23, it is obvious to see that the cost is also bounded by $O(n)$. Therefore, the overall cost of constructing the arrays I_0, and J_0 takes $O(n)$ time.

6 Performance Comparison Between Two $O(\sqrt{n})$ Methods

I implemented both the $O(\sqrt{n})$ methods introduced by Chan et al. [1] and the new $O(\sqrt{n})$ method using Java language, and compared the query and space performance between these two methods. The programs are run on the laptop with the processor $Intel\ Core\ i5@2.90GHz$ and $8GB$ memory. See Figure 8 for the detail of the test data.

From the test results, we can tell the overall average query time of the new method is slightly better than the $O(\sqrt{n})$ method from Chan et al. [1]. When coming across the input array with special data patterns such as the frequency of the mode of the overall array is far less than $\sqrt{n}$, the new method performs much better. The outstanding advantage of the new method is that it decreases the pre-processing time to construct precomputed data structure

by more than 80%, when the size of array exceeds 1 million. One of the two reason is that the pre-processing time of the method applying on the array $B1$ is bounded by $O(n)$. On the other hand, the pre-processing time of the method by Chan et al. [1] is bounded by $O(s \times n + s \times s)$. The other reason is that it takes $O(s \times n + s \times m)$ time to pre-process the data structure for array $B0$. And m could be far less than s.

However, the disadvantage of the new method is the size of its precomputed data structure could be as twice large as the $O(\sqrt{n})$ method from Chan et al. [1]. See Figures 5, 6 and 7 for the query and space performance of the two methods.

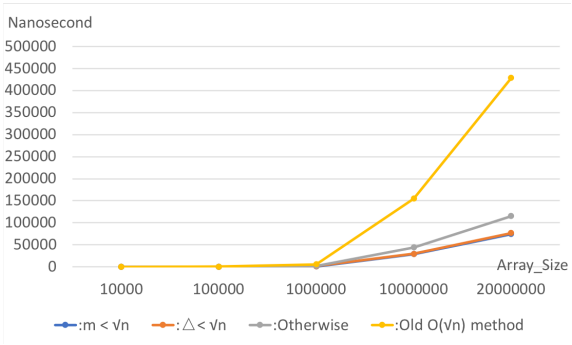

Figure 5: The yellow line shows the precomputing time of the $O(\sqrt{n})$ method from Chan et al. [1]. The other three lines represents the precomuting time of the new $O(\sqrt{n})$ method, among which, the blue line shows the case when m is less than $\sqrt{n}$, the red line represents the case when Δ is less than $\sqrt{n}$, and the grey line shows the other general cases.

.

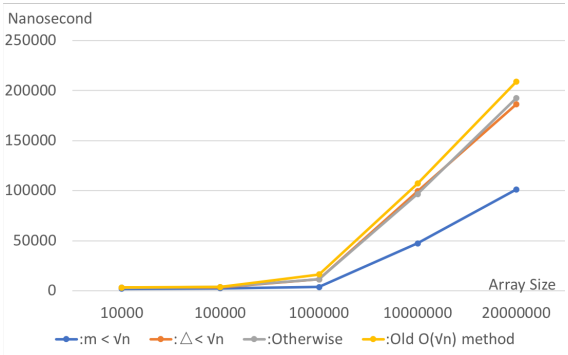

Figure 6: The average query time comparison between the two $O(\sqrt{n})$ methods

.

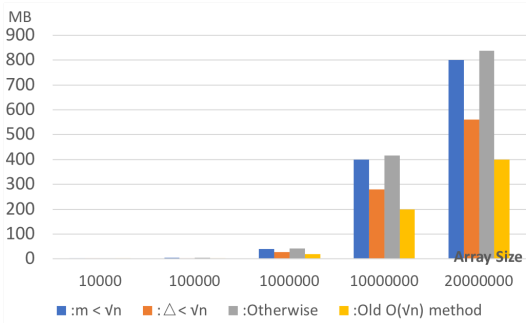

Figure 7: The comparison of the space of the precomputed data structures between the two $O(\sqrt{n})$ methods

.

n	m < √n		△ < √n		Otherwise		Test Cases for Query
20000000	m	△	m	△	m	△	500000
	4224	5000	6969	3000	4500	4800	
	B0.length	B1.length	B0.length	B1.length	B0.length	B1.length	
	20000000	0	0	20000000	10100000	9900000	
10000000	m	△	m	△	m	△	500000
	2672	4000	5275	2000	10400	3229	
	B0.length	B1.length	B0.length	B1.length	B0.length	B1.length	
	10000000	0	0	10000000	5100000	4889600	
1000000	m	△	m	△	m	△	1000000
	572	2000	1382	800	1100	1200	
	B0.length	B1.length	B0.length	B1.length	B0.length	B1.length	
	1000000	0	0	1000000	461000	539000	
100000	m	△	m	△	m	△	100000
	296	400	454	250	350	340	
	B0.length	B1.length	B0.length	B1.length	B0.length	B1.length	
	100000	0	0	100000	65000	35000	
10000	m	△	m	△	m	△	10000
	69	200	163	70	120	160	
	B0.length	B1.length	B0.length	B1.length	B0.length	B1.length	
	10000	0	0	10000	5200	4800	

Figure 8: There are 5 groups of test data. The sizes of the arrays range from 10,000 to 20 million. In each group, it contains 3 kinds of data patterns to make sure different strategies tested. The different strategies include that only $O(m)$ method applies, only $O(\Delta)$ method applies, and both $O(m)$ and $O(\Delta)$ methods apply. To compare the query efficiency among different data patterns, I used the indicator average query time based on different kinds of ranges $[i, j]$. The numbers of different pairs of ranges ([i, j]) are listed in the last column.

.

References

[1] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. *Linear-Space Data Structures for Range Mode Query in Arrays.* Theory Comput. Syst., 2014.

[2] Stephane Durocher and Jason Morrison. *Linear-Space Data Structures for Range Mode Query in Arrays*. CoRR abs, 2011.

[3] Hicham El-Zein, Meng He, J. Ian Munro, and Bryce Sandlund. *Improved Time and Space Bounds for Dynamic Range Mode*. ESA, 2018.

[4] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. *Cell Probe Lower Bounds and Approximations for Range Mode*. ICALP, 2010.

[5] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. *Range Mode and Range Median Queries on Lists and Trees*. Nord. J. Comput., 2005.

[6] Holger Petersen and Szymon Grabowski. *Range mode and range median queries in constant time and sub-quadratic space*. Inf. Process. Lett., 2009.