# Effective Mode Range Query in Arrays

Younan Gao

# Introduction

- Mode
  - The most frequent element in the array is called mode. Mode is not necessarily unique.
- Range Query
  - Seeks to compute the corresponding statistic on the subarray A[l, r]
- Motivations:
  - Mode is a fundamental statistic in data analysis
  - My project tries to break the bound of the query time O($\sqrt{n}$) **under certain data patterns** in practice

# Related Work

| Reference | Author | Year | Query Time | Update Time | Space | Worst Case | Lower bound | Remark |
|---|---|---|---|---|---|---|---|---|
| [1] | Hicham El-Zein | 2018 | O(n^(2/3)) | O(n 2/3 ) | O(n) | | | Dynamic |
| [2] | Timothy M. Chan | 2014 | O(n^(3/4) log log n) | O(n^(3/4) log log n) | O(n) | O(n^(3/4) log n/ log log n) | | Dynamic |
| [2] | Timothy M. Chan | 2014 | O(n^(2/3) lg n/ lg lg n) | O(n^(2/3) lg n/ lg lg n) | O(n^(4/3)) | | | Dynamic |
| [2] | Timothy M. Chan | 2014 | O( √ n/ log n) | | O(n) | | | Static |
| [3] | S. Durocher | 2011 | O(n^t); O(k); O(m); O(\|j-i\|) | | O(n^(2-2t)) | | | 0<t<=1/2 |
| [4] | Mark Greve | 2010 | | | S memory cells of w bits | | Ω(log n/(log(Sw/n))) | Cell Probe Model |
| [5] | Holger Petersen | 2008 | O(1) | | O(n^2( loglog n)/ (log n)^2) | | | |
| [5] | Holger Petersen | 2008 | O(n^ε ) | | O(n^ (2 − 2ε )) | | | 0 ≤ ε< 1/2 |
| [5] | Holger Petersen | 2008 | O(1) | | O(n^2/logn) | | | |
| [6] | Krizanc | 2005 | O(n^t log n) | | O(n^(2-2t)) | | | 0<t<=1/2 |
| [6] | Krizanc | 2005 | O(1) | | O(n^2( loglog n)/ (log n)) | | | |

- What is the difference between dynamic and static
  - Dynamic means the precomputed data structures supports data update
  - In contrast, static means, once data is update, all the precomputed data structures have to been constructed from the scratch.

# Why breaks the bound O($\sqrt{n}$) is hard

- Lower bound of MRQ from Greve et al.
  - Ω(log n / log (s * w / n))) uses s memory cells of w bits
  - Which is Ω(log n / loglog n) query time using O(n) space under RAM model
- Reduce Boolean Matrix Multiplication to MRQ by Chan et al.
  - "A query time significantly below $\sqrt{n}$ cannot be achieved by purely combinatorial techniques"
- Preprocess an O(n)-sized data structure and answer RMQ cannot be done better in O($n^{ω/2}$) time from He et al.
  - ω is the constant in the exponent of the running time of matrix multiplication, which is 2.3727 with current knowledge

# Why breaks the bound O($\sqrt{n}$) is hard(cont.)

- Reduce Boolean Matrix Multiplication to MRQ by Chan et al.
  - An example of $\sqrt{n}$ * $\sqrt{n}$ Boolean Matrix Multiplication

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | | 0 | 1 | 1 | | 1 | 1 | 1 |
| 1 | 1 | 0 | * | 0 | 0 | 1 | = | 0 | 1 | 1 |
| 1 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 |

A[3][3] * B[3][3] = C[3][3]

  - How to calculate the multiplication with MRQ
    - Preprocess

| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Item | 2 | 1 | 3 | 3 | 1 | 2 | 1 | 2 | 3 | 3 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 3 |

| | 1 | 2 | 3 |
|---|---|---|---|
| i | 2 | 5 | 7 |
| j | 10 | 13 | 16 |

    - Calculate

| | | |
|---|---|---|
| 4 | 5 | 4 |
| 2 | 4 | 4 |
| 2 | 2 | 4 |

$\xrightarrow{\text{Max Freq – (\#Complete blocks) - 1}}$

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

# Preliminaries(Notation)

| Notation | Meaning |
| --- | --- |
| m | The highest frequency of the whole array, which is unique |
| s | The block size |
| t | The size of one block |
| n | The length of the array |
| $\triangle$ | The total number of distinct items |

# Preliminaries(cont.)

- Lemma
  - (Krizanc et al.) Let A1 and A2 be any multisets. If c is a mode of A1 ∪ A2 and c ∉ A1, then c is a mode of A2.
- Convention
  - Preprocess the input array by constructing a data structure to speed up the query time
  - However, all methods here are using O(n) space
  - The project focuses on static range query

# Rank Reduction

- Example
  - {10, 20, 20, 10, 30, 30, 10, 40, 40}
  - After reduction, {1, 2, 2, 1, 3, 3, 1, 4, 4}
- Functions
  - Rank Reduction transfer the data set from universe to $\{1..\Delta\}$
  - The mode of the rank reduction array corresponds to the mode of the original array
  - Operations on rank reduction arrays improves query and space efficiency
  - By using TreeMap(AVL Tree), Rank Reduction could be implemented within $O(\log \Delta)$ time

# O(|j - i|) Method

- The most obvious method
  - Direct search without preprocess
  - Takes O(n) space and O(|j-i|) time

```
public Result query_algorithm(int index_i, int index_j) {

    Map<Integer, Integer> map_count_item = new HashMap<Integer, Integer>();
    Result result = new Result(-1, 0);
    int tmp;

    for (int i = index_i; i <= index_j; i++) {
        if (map_count_item.containsKey(this.original_arr[i])) {
            tmp = 1 + map_count_item.get(this.original_arr[i]);
            map_count_item.put(this.original_arr[i], tmp);
        } else {
            tmp = 1;
            map_count_item.put(this.original_arr[i], tmp);
        }

        if (tmp > result.getFrequency()) {
            result.setFrequency(tmp);
            result.setMode(this.original_arr[i]);
        }
    }

    return result;
}
```

# O(△) Method

- Preprocess
  - Make t = △, and s = n / △.
  - Precompute an Array C[△][s], for each C[i, j] stores the frequency of item i in the range from 0 to (j*△-1)
  - An example as shown below
    - Input array(Rank Reduction beforehand):

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Item | 1 | 2 | 3 | 4 | 1 | 2 | 2 | 4 | 1 | 1 | 4 | 2 |

    - Array C[△][s]

| | 0 | 1 | 2 |
|---|---|---|---|
| 1 | 1 | 2 | 4 |
| 2 | 1 | 3 | 4 |
| 3 | 1 | 1 | 1 |
| 4 | 1 | 2 | 2 |

    - With array C, we can get the frequency of any item between any span of blocks in O(1) time
  - Preprocessing Operation takes O(n) space and O(n) time

# O($\triangle$) Method(cont.)

- Query Algorithm
  - Compute the respective frequencies of all distinct items in the target range [i, j]
    - Compute the respective frequencies of all distinct items in the target range [0, j] and store the frequencies in array C1[$\triangle$]
    - Compute the respective frequencies of all distinct items in the target range [0, i-1] and store the frequencies in array C2[$\triangle$]
    - The frequencies in the range [i-j] could be computed by C2 – C1
  - Pick the maximum frequency among the array if C2 - C1
  - Overall, it takes O($\triangle$)

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Item | 1 | 2 | 3 | 4 | 1 | 2 | 2 | 4 | 1 | 1 | 4 | 2 |

# O($\sqrt{n}$) Method

- Preprocess
  - Make s = t = $\sqrt{n}$
  - There are totally four arrays needed to be precomputed. Each one takes O(n) space.
    - Q[1..△][0..m-1]: Each entry Q[i][j] stores the index of item i in the original array.
    - Array_Prime[0..n-1](denoted by P): Each entry P[i] stores the index of the item Original_Array[i] in the array Q[Original_Array[i]]
    - Array_Freq[0..s-1][0..s-1]: Each entry Array_Freq[i][j] stores the maximum frequency in the range from (i*t) to (j*t-1)
    - Array_Mode [0..s-1][0..s-1]: Each entry Array_Mode[i][j] stores the mode in the range from (i*t) to (j*t-1)
  - The respective time cost of each array is as shown below

| | **Array_Prime** | **Q** | **Array_Freq** | **Array_Mode** |
|---|---|---|---|---|
| **Time cost** | O(n) | O(n) | O(s*n) | O(s*n) |

  - An example for illustration

| **block_index** | 0 | | | 1 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **Original_Array** | 1 | 2 | 2 | 1 | 3 | 3 | 1 | 4 | 4 |
| **Array_Prime** | 0 | 0 | 1 | 1 | 0 | 1 | 2 | 0 | 1 |

| **Q** | **0** | **1** | **2** |
|---|---|---|---|
| 1 | 0 | 3 | 6 |
| 2 | 1 | 2 | |
| 3 | 4 | 5 | |
| 4 | 7 | 8 | |

| Mode | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 1 |
| 1 | | 3 | 3 |
| 2 | | | 4 |

| Freq | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 2 | 2 | 3 |
| 1 | | 2 | 2 |
| 2 | | | 2 |

# O($\sqrt{n}$) Method(cont.)

- Query Algorithm includes 3 parts
  - Compute the mode and its frequency (denoted by fc) in the span, which covers the complete blocks within the target range
  - Compute the mode and its frequency in the prefix and suffix

| block_index | 0 | | | 1 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Original_Array | 1 | 2 | 2 | 1 | 3 | 3 | 1 | 4 | 4 |

Prefix Span Suffix

```
for each item in the prefix
    /**
     * get the precessor index of the "[original_arr_prime[x]"th item
     * [original_arr[x]] in array Q[original_arr[x]]
     */
    prev_in_q = array_Q[original_arr[x]][original_arr_prime[x] - 1];
    if(prev_in_q >= the start of the target range)
        //This element original_arr[x] has already been counted
        continue;
    else if((original_arr_prime[x] + fc - 1) < array_Q[original_arr[x]].length &&
            array_Q[original_arr[x]][original_arr_prime[x] + fc - 1] <= end_j)

        scan array_Q[original_arr[x]] starts
                    from original_arr_prime[x] + fc - 1 to the end of the target range
        update the maximum frequency and the corresponding mode so far
    else
        //The frequency of original_arr[x] is less than fc
        continue
```

- The overall query time is O(t), which is O($\sqrt{n}$)

# O(m) Method

- Preprocess
  - This is a **new method** and slightly more efficient on space and query time
  - Make t = m and s = n / t
  - There are totally four arrays to be precomputed. Each one takes O(n) space.
    - Q[1..△][0..m-1]: Each entry Q[i][j] stores the index of item i in the original array.
    - Array_Prime[0..n-1] : Each entry Array_Prime[i] stores the corresponding index of the item Original_Array[i] in the array Q[i]
    - F[0..s-1][1..m]: Each entry F[i][j] stores the smallest index(denoted by v), such that the mode of the range [i*m, v] has frequency **at most j**;
    - Arr_Mode[0..s-1][1..m]: Each entry Arr_Mode[i][j] stores the corresponding mode of the entry F[i][j]
  - The respective time cost to precompute each array is as shown below

| | Array_Prime | Q | Array_F | Array_Mode |
|---|---|---|---|---|
| Time Cost | O(n) | O(n) | O(s*n) | O(s*n) |

  - An example

| block_index | 0 | | | 1 | | | 2 | | |
|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Original_Array | 1 | 2 | 2 | 1 | 3 | 3 | 1 | 4 | 4 |
| Array_Prime | 0 | 0 | 1 | 1 | 0 | 1 | 2 | 0 | 1 |

| Q | 0 | 1 | 2 |
|---|---|---|---|
| 1 | 0 | 3 | 6 |
| 2 | 1 | 2 | |
| 3 | 4 | 5 | |
| 4 | 7 | 8 | |

| Array_F | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 2 | 6 |
| 1 | 3 | 5 | 9 |
| 2 | 6 | 8 | 9 |
| Arr_Mode | 1 | 2 | 3 |
| 0 | 1 | 2 | 1 |
| 1 | 1 | 3 | 3 |
| 2 | 1 | 4 | 4 |

# O(m) Method(cont.)

- Query Algorithm
  - A mode of the span **plus suffix** and its frequency (denoted by fc) can be computed by finding the predecessor of index_j in corresponding array Array_F[t] (t represents the block index where the left side of the span locate);
    - The index of the found predecessor equals to the frequency fc
    - Arr_mode[t][fc] represents the mode of the span plus suffix. (t represents the block index where the left side of the span locate)
  - Secondly, we only need to scan the prefix items to identify the candidate mode which frequency is more than fc using the same way to the previous method.
  - There is no need to scan the suffix items, Proof:
    - If the suffix shares some items with the prefix, then when scanning the items in prefix, these common items can be covered.
    - For the items only contained in the suffix, their frequency could not be more than fc.
  - 3 ways to find the index of the predecessor in array_F

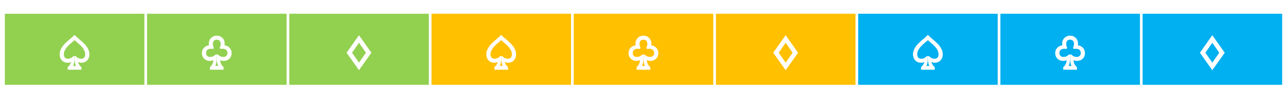
| | vEB | Binary Search | Linear Scan |
|---|---|---|---|
| Time cost | O(lglgn) | O(lgm) | O(m) |
| | | O(lgn) | O(sqrt(n)) |

# Another O($\sqrt{n}$) Method

- Idea in Pseudocode:

```
Method_Four(int []p, int n):
    s = sqrt(n);
    #B0 stores the items with freq <= s
    #B1 stores the items with freq > s
    [B0, B1] = array_partition(p);
    result_1 <- Apply "O(m)" method on B0
    result_2 <- Apply "O(△)" method on B1
    final_result <- compare(result_1, result_2)
    return final_result
```

- Why is it O($\sqrt{n}$)
  - Get mode from array B0 takes O(m), which is O($\sqrt{n}$)
  - Get mode from array B1 takes O(△) time. When the frequencies of each item is **more than s**, △ must be **less than t**. (Here s = t = $\sqrt{n}$)
  - Proof:



# Another O($\sqrt{n}$) Method (cont.)

- It still uses O(n) space
  - B0 takes O(s*m) space, which is O(n)
  - B1 needs O(s*△) space, which is also O(n)
- The value of the new O($\sqrt{n}$) Method
  - Original method: the worst case of the mode range query for any array takes O($\sqrt{n}$)
  - New method: there exists some arrays satisfying certain data pattern, on which the worst case of query time could be much less than O($\sqrt{n}$).
  - E.g. {1, 2, 3, 4, 5, 6, 7, 8, 9}

# Implementation

- Programming Language: Java
- Data Structure from Java utility package
  - HashMap
  - Array
  - TreeMap
- Implements on five methods
- Machine: 2.9GHz/8GB

# Data Structure from Java

- Array
  - Implemented by self-adjusting list
  - However, it is used as a fixed size list during implementation
- HashMap:
  - Put/Get: O(1) time
  - Avoid iterating HashMap by Rank Reduction beforehand
- TreeMap
  - Red Black Tree
  - Put/Get/ContainsKey: O(lgn) time
  - Iteration: O(nlgn) time

| Methos | Array | HashMap | TreeMap |
|---|---|---|---|
| \|j - i\| | √ | √ | X |
| O(m) | √ | √ | X |
| O(△) | √ | X | X |
| O($\sqrt{n}$) | √ | X | √ |
| Rank Reduction | √ | √ | √ |
| Array_Partition | √ | √ | X |

# Implementation Support

- Array Partition
- Rank Reduction on Array
- Sample Data Generator
- Serialization by Java
  - Serialization is the process of turning an object in memory into a stream of bytes.

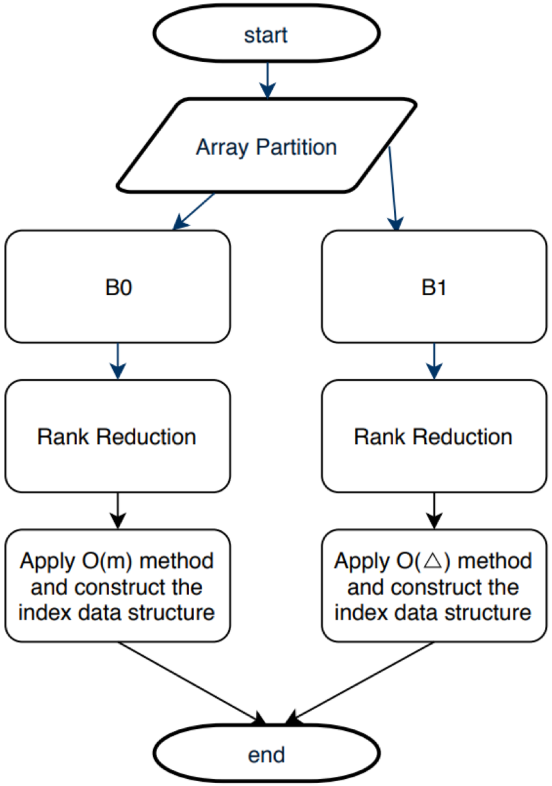
# An Example on Array Partition

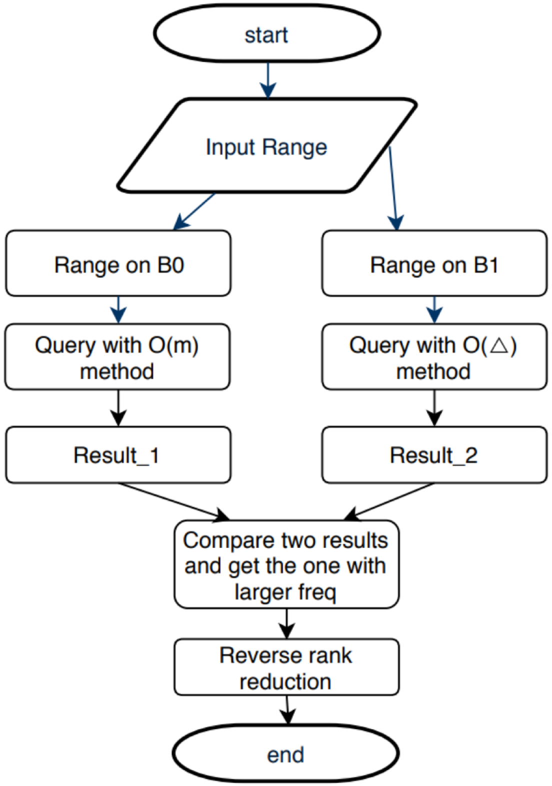
- Original Array
  - {2, 7, 2, 7, 1, 5, 7, 2, 7}
- Array Partitions
  - B0: {2, 2, 1, 5, 2}
  - B1: {7, 7, 7, 7}
- Precompute Arrays take O(n) time and O(n) space

| | 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 |

# Implementation of the new O($\sqrt{n}$) Method

- One of the four strategies as shown below:




Precompute

Range Query

# Implementation of the new O($\sqrt{n}$) Method

- Based on different data patterns, apply different strategies

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

# Performance Comparison between two $O(\sqrt{n})$ methods

- Performance Comparison

| Performance Comparison | | n | m | △ | n | m | △ | n | m | △ |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 10000000 | 2659 | 4000 | 10000000 | 10370 | 1000 | 10000000 | 10400 | 3229 |
| O(√n)Method | | Original | New | | Original | New | | Original | New | |
| | | | B0.length | B1.length | | B0.length | B1.length | | B0.length | B1.length |
| | | | 10000000 | 0 | | 0 | 10000000 | | 5100000 | 4889600 |
| | | | O(m) method | | | O(△) method | | | O(m) method | O(△) method |
| Precompute | Total Size of Precompute Data Structure(MB) | 200.1 | 400.2 | | 200.1 | 280.1 | | 200.1 | 416.5 | |
| | Time Cost(Millisecond) | 63432 | 29546 | | 62855 | 710 | | 70246 | 14731 | |
| Range Query (Microsecond) | Prefix+Span | 693 | 223 | | 371 | 451 | | 575 | 913 | |
| | Span+Suffix | 626 | 2 | | 677 | 182 | | 564 | 209 | |
| | Less than one block | 224 | 2220 | | 243 | 170 | | 262 | 2302 | |
| | Prefix+Span+Suffix | 690 | 407 | | 133 | 179 | | 982 | 560 | |
| | Span | 372 | 2 | | 325 | 150 | | 354 | 154 | |

- Analysis
  - Space for precompute data structure

| Space | | | | |
|---|---|---|---|---|
| Original | | New | | |
| Array | Size | Array | | Size |
| original_arr | n | original_arr | | n |
| original_arr_prime | n | Array_Patition | array_B | n |
| array_Q | n | | array_I | 2*n |
| array_s | n=s*s | | array_J | 2*n |
| array_s_prime | n=s*s | O(m) Method | original_arr_prime | n |
| | | | array_Q | n |
| | | | array_F | s*m |
| | | | array_mode | s*m |
| | | O(△) Method | array_C | △*s |
| Total | 5n | Total | | range from 6n to 11n |

# Performance Comparison between two $O(\sqrt{n})$ methods (cont.)

- Analysis
  - Precompute time cost

| Precompute Time | | | | |
|---|---|---|---|---|
| Original | | New | | |
| Array | Time | Array | | Time |
| original_arr | O(1) | original_arr | | O(1) |
| original_arr_prime | O(n) | Array_Patition | array_B | O(n) |
| array_Q | O(n) | | array_I | O(n) |
| array_s | O(n*s + s*s) | | array_J | O(n) |
| array_s_prime | O(n*s + s*s) | O(m) Method | original_arr_prime | O(n) |
| | | | array_Q | O(n) |
| | | | array_F | O(s*n+s*m) |
| | | | array_mode | O(s*n+s*m) |
| | | O(△) Method | array_C | O(n) |
| **Total** | O(n*s + s*s) | **Total** | | range from O(n) to O(s*n+s*m) |

  - Query time

| Note: $t == \sqrt{n}$ $m <= \sqrt{n}$ $\triangle <= \sqrt{n}$ | Query Time | | |
|---|---|---|---|
| | Original | New | |
| | | O(m) method | O(△)method |
| **Prefix+Span** | O(t)+O(1) | O(t)+O(m) | O(△) |
| **Span+Suffix** | O(t)+O(1) | O(m) | O(△) |
| **Less than one block** | O(\|j-i\|)∈O(t) | O(\|j-i\|)∈O(m) | O(\|j-i\|)∈O(△) |
| **Prefix+Span+Suffix** | O(t)+O(1)+O(t) | O(t)+O(m) | O(△) |
| **Span** | O(1) | O(m) | O(△) |

# Future Work

- **Target:** Identify other data patterns satisfying fast mode range query which breaks the bound O($\sqrt{n}$).
- One Trivial Finding
  - Query the frequency of any number in range blocks with constant time.
- Example: {10, 20, 20, <u>10, 30, 30</u>, 10, 40, 40}

| | 1 | | 2 | | 3 | |
|---|---|---|---|---|---|---|
| 10 | 0 | 1 | 0 | 1 | 0 | 1 |
| 20 | 0 | 0 | 1 | 1 | 1 | |
| 30 | 1 | 0 | 0 | 1 | 1 | |
| 40 | 1 | 1 | 0 | 0 | 1 | |

Find the freq of 10 between block 2 and 3

| | |
|---|---|
| $Select_1(3) = 6$ | $Select_1(2-1) = 2$ |
| $Rank_1(6) = 3$ | $Rank_1(2) = 1$ |

The freq of 10 = 3 - 1

- Analysis:
  - Space: O( (n + ▵ * m) / word size)
  - Running time: O(▵)
  - Limitation: The unit of query range is block

# Future Work ( cont.)

- Extension:
  - O(1) time to get the frequency of any number in any range
- Example

| | |
|---|---|
| 10 | 011101110111 |
| 20 | 10101111111 |
| 30 | 11110101111 |
| 40 | 11111110101 |

- Analysis:
  - Space: O(($\triangle$+1)*n / log n) words, regarding log n is O(word size)
  - Query time: O(1)
- Apply this method on mode range query:
  - Space: the same
  - Query time: O($\triangle$)
  - When $\triangle$ is O(log n), the space is O(n) and the query time is O(log n). **This is another way to break the $O(\sqrt{n})$ bound for arrays with specific data pattern.**

# Reference


1. Hicham El-Zein, Meng He, J. Ian Munro, Bryce Sandlund: Improved Time and Space Bounds for Dynamic Range Mode. ESA 2018: 25:1-25:13
2. Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, Bryan T. Wilkinson: Linear-Space Data Structures for Range Mode Query in Arrays. Theory Comput. Syst. 55(4): 719-741(2014)
3. Stephane Durocher, Hicham El-Zein, J. Ian Munro, Sharma V. Thankachan:Low space data structures for geometric range mode query. Theor. Comput. Sci. 581: 97-101 (2015)
4. Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, Jakob Truelsen: Cell Probe Lower Bounds and Approximations for Range Mode. ICALP (1) 2010: 605-616
5. Holger Petersen, Szymon Grabowski:Range mode and range median queries in constant time and sub-quadratic space. Inf. Process. Lett. 109(4): 225-228(2009)
6. Danny Krizanc, Pat Morin, Michiel H. M. Smid: Range Mode and Range Median Queries on Lists and Trees. Nord. J. Comput. 12(1): 1-17 (2005)

# Thanks!

Any question?