On Range-1-Query (R1Q) Problem

1 Introduction

The Range-1-Query (R1Q) problem is defined as follows. Given a bit vector $A[1 \dots n]$ and indices i and j such that $1 \leq i \leq j \leq n$, find efficiently if there exists a 1 in the subarray $A[i, j]$. We usually represent it as $R1Q(i, j)$. In this paper, we present an efficient approximation algorithm to answer the range queries in constant time with very less extra memory. We also extend the algorithm to solve the problem in higher dimensions.

The R1Q problem is closely related to Range Minimum Query (RMQ) problem. Given an array $A[1 \dots n]$ and indices i and j, the range minimum query denoted as $RMQ(i, j)$ asks us to find the position of the minimum element in the subarray $A[i, j]$.

As no good algorithms have been developed for R1Q, we focus our attention to RMQ and survey a few existing algorithms for the same. A good survey of RMQ algorithms is given in [8]. For each of the algorithms we give the complexity of the algorithm as a 3-tuple in the form (ExtraSpaceInBits, PreprocessTime, QueryTime).

1.1 Scan Algorithm

We simply scan the array A from index i to j and find the minimum element. Complexity = $(\mathcal{O}(1), \mathcal{O}(1), \mathcal{O}(n))$

Algorithm 1 : RMQ-SCAN(*start*, *end*)

```
1: min ← start
2: for i ← start + 1 to end do
3:     if A[i] < A[min] then
4:         min ← i
5: return min
```

1.2 Query Table (QT)

The array A is preprocessed and a table called *Query Table*, M, is created which stores the range minimum for all $\mathcal{O}(n^2)$ possible range queries. Each element of the query table, $M[i][j]$, stores the position of the minimum element in the subarray $A[i \ldots j]$ which is found by scanning over the range. For any given query, we just find the value at $M[i][j]$.

Complexity $= (\mathcal{O}(n^2 \log(n)), \mathcal{O}(n^3), \mathcal{O}(1))$

Algorithm 2 : RMQ-QT-PREPROCESS()

```
1: for i ← 1 to n do
2:     for j ← i to n do
3:         M[i][j] ← RMQ-SCAN(i, j)
4: return M
```

Algorithm 3 : RMQ-QT(*start*, *end*)

```
1: return M[start][end]
```

1.3 Query Table (QT): Dynamic Programming (DP)

The algorithm is almost same as the Query Table algorithm except while finding the range minimum during preprocessing, we use dynamic programming. That is, we find $M[i][j]$ using $M[i][j-1]$ and $a[j]$ as shown below. Using DP, we can reduce the preprocessing time by a factor of n.

Complexity $= (\mathcal{O}(n^2 \log(n)), \mathcal{O}(n^2), \mathcal{O}(1))$

Algorithm 4 : RMQ-QTDP-PREPROCESS()

```
1: for i ← 1 to n do
2:     M[i][i] ← i
3: for i ← 1 to n − 1 do
4:     for j ← i + 1 to n do
5:         if A[j] < A[M[i][j − 1]] then
6:             M[i][j] ← j
7:         else
8:             M[i][j] ← M[i][j − 1]
9: return M
```

Algorithm 5 : RMQ-QTDP($start$, end)

1: **return** $M[start][end]$

1.4 Chunks Algorithm

The array A is divided into $\sqrt{n}$ chunks, each of which will be of size $\sqrt{n}$. For simplicity, we can assume the n is a perfect square. Maintain an array M of size $\sqrt{n}$, where each element $M[k]$ stores the position of the minimum element in the k^{th} chunk. To find the position of the minimum element in subarray $A[i \dots j]$, we have to make use of at most three subranges:

- The partial chunk, from i to the start of the first innermost chunk
- All complete chunks in the range from i to j
- The partial chunk, from the end of the last innermost chunk to j

Complexity = $(\mathcal{O}(\sqrt{n}\log(n)), \mathcal{O}(n), \mathcal{O}(\sqrt{n}))$

Algorithm 6 : RMQ-Chunks-Preprocess()

1: **for** $i \leftarrow 1$ to $\sqrt{n}$ **do**
2: $\quad min \leftarrow \sqrt{n}(i-1)+1$
3: $\quad$ **for** $j \leftarrow \sqrt{n}(i-1)+1$ to $\sqrt{n} \cdot i - 1$ **do**
4: $\quad\quad$ **if** $A[j] \leq A[min]$ **then**
5: $\quad\quad\quad min \leftarrow j$
6: $\quad M[i] \leftarrow min$
7: **return** M

1.5 Sparse Table (ST)

The algorithm is given in [4]. The array A is preprocessed and a table called *Sparse Table*, M, of size $n\log(n)$ is created. The element $M[i][j]$ stores the position of the minimum element in the subarray $A[i \dots (i+2^j-1)]$. To answer the range query, divide the range $[i, j]$ into two subranges which entirely cover the range and find the range minimum from the two. Both the subranges will be of size $2^{\lfloor \log(j-i+1) \rfloor}$, but, one subrange starts at i and the other ends at j.

Complexity = $(\mathcal{O}(n\log^2(n)), \mathcal{O}(n\log(n)), \mathcal{O}(1))$

Algorithm 7 : RMQ-Chunks($start$, end)

```
min ← start
chunkstart ← ⌈start/√n⌉
chunkend ← ⌊end/√n⌋
for i ← start + 1 to √n · chunkstart do
    if A[i] < A[min] then
        min ← i
for i ← chunkstart to chunkend do
    if M[i] < A[min] then
        min ← M[i]
for i ← √n · chunkend to end do
    if A[i] < A[min] then
        min ← i
return min
```

Algorithm 8 : RMQ-ST-Preprocess()

```
for i ← 1 to n do
    M[i][0] ← i
for j ← 1 to ⌊log(n)⌋ do
    for i ← 1 to n − 2^j + 1 do
        if A[M[i][j − 1]] < A[M[i + 2^(j−1) − 1][j − 1]] then
            M[i][j] ← M[i][j − 1]
        else
            M[i][j] ← M[i + 2^(j−1) − 1][j − 1]
return M
```

Algorithm 9 : RMQ-ST($start$, end)

```
k ← ⌊log(end − start + 1)⌋
min ← 0
if A[M[start][k]] ≤ A[M[end − 2^k + 1][k]] then
    min ← M[start][k]
else
    min ← M[end − 2^k + 1][k]
return min
```

1.6 Segment Trees

The array A is preprocessed and a segment tree is built for it. A segment tree is a heap like data structure where each node represents an interval or a segment.

The segment tree for the interval or range $[l, r]$ is recursively defined as:

1. The node for $[l, r]$ holds the information for the interval. In RMQ problem, the node stores the position of the minimum element in the interval.
2. If $l < r$, then the node $[l, r]$ will have two children
   (a) Left child with interval $[l, \lfloor (l + r)/2 \rfloor]$
   (b) Right child with interval $[\lfloor (l + r)/2 \rfloor + 1, r]$

The height of the segment tree is $\lfloor \log n \rfloor + 1$. Store the information of each of the nodes of the segment tree in an array M, of size $2^{\lfloor \log n \rfloor + 2}$, where each element $M[i]$ stores the position of the minimum element for the i^{th} node. To answer $RMQ(i, j)$ we split $[i, j]$ as

$$[i, j] = [i, p_1] + [p_1 + 1, p_2] + [p_2 + 1, p_3] + \cdots + [p_m + 1, j]$$

such that the intervals $[i, p_1], [p_1+1, p_2], \ldots, [p_m+1, j]$ are present in the segment tree and the position of the minimum element for all these intervals are stored in M. The query can be answered by finding the minimum element among all these intervals and taking their minimum. Any interval can be split down into $\mathcal{O}(\log(n))$ segment tree intervals. Hence, answering the query takes $\mathcal{O}(\log(n))$ time.

Complexity = $(\mathcal{O}(n \log(n)), \mathcal{O}(n), \mathcal{O}(\log(n)))$

Algorithm 10 : RMQ-SegT-Preprocess()

```
1: for i ← 0 to n − 1 do
2:     M[i] ← −1
3: RMQ-SegT-Initialize(1, 0, n − 1)
4: return M
```

1.7 Berkman Vishkin Algorithm

The core ideas of the algorithm are from previous works [11] and [10]. The presentation combines the ideas given in [5] and [10]. The presentation is closely followed from [3] and [9]. A cartesian tree C, of an array A is a binary tree. The root of the tree is the minimum element of the array and it stores the position of the minimum element. The left and right children are recursively constructed. The tree can be built in $\mathcal{O}(n)$ time.

Algorithm 11 : RMQ-SegT-Initialize$(i, low, high)$

1: **if** $low = high$ **then**
2: $M[i] \leftarrow low$
3: **return**
4: RMQ-SegT-Initialize$(2i, low, \left(\frac{low+high}{2}\right))$
5: RMQ-SegT-Initialize$(2i+1, \left(\frac{low+high}{2}+1\right), high)$
6: **if** $A[M[2i]] \leq A[M[2i+1]]$ **then**
7: $M[i] = M[2i]$
8: **else**
9: $M[i] = M[2i+1]$
10: **return**

Algorithm 12 : RMQ-SegT$(i, low, high, start, end)$

1: **Function call:** RMQ-SegT$(1, 0, n-1, start, end)$
2: **if** $start > high$ or $end < low$ **then**
3: **return** -1
4: **if** $low \geq start$ and $high \leq end$ **then**
5: **return** $M[i]$
6: $lmin \leftarrow$ RMQ-SegT$(2i, low, \left(\frac{low+high}{2}\right), start, end)$
7: $rmin \leftarrow$ RMQ-SegT$(2i+1, \left(\frac{low+high}{2}+1\right), high, start, end)$
8: **if** $lmin = -1$ **then**
9: $min \leftarrow rmin$
10: **else if** $rmin = -1$ **then**
11: $min \leftarrow lmin$
12: **else if** $A[lmin] \leq A[rmin]$ **then**
13: $min \leftarrow lmin$
14: **else**
15: $min \leftarrow rmin$
16: **return** min

The cartesian tree for the range $A[l, r]$ is recursively defined as:

1. The root is the minimum element in the range $[l, r]$ and labeled with the position of the minimum element in the range.
2. If $l < r$, then the node $[l, r]$ will have two children
   (a) Left child with interval $[l, \lfloor (l + r)/2 \rfloor]$
   (b) Right child with interval $[\lfloor (l + r)/2 \rfloor + 1, r]$

Three arrays E, H, R are created to store the information of the cartesian tree.

1. $E[1 \dots 2n - 1]$ stores the labels of visited nodes in the Eulerian tour
2. $H[1 \dots 2n - 1]$ stores the heights of the corresponding visited nodes
3. $R[1 \dots n]$ stores the first occurrence of $A[i]$ in the Euler tour

We now answer the range query using the formula
$RMQ_A(i, j) = E[\pm RMQ_H(R[i], R[j])]$

We solve $\pm RMQ_H(R[i], R[j])$ as follows. Let $N = 2n - 1$. The array $H[1 \dots N]$ is partitioned into blocks of size $\frac{\log N}{2}$. Define two arrays A' and B such that

1. $A'[1 \dots \frac{2N}{\log N}]$ stores the minimum element in different blocks
2. $B[1 \dots \frac{2N}{\log N}]$ stores the position of the minimum element in different blocks

The array A' is preprocessed using ST algorithm. Answering range queries in A' has the complexity in extra space and query time as $(\mathcal{O}(N), \mathcal{O}(1))$. The array A' should also be preprocessed to answer queries in a single block. There are $\frac{2N}{\log N}$ blocks each of size $\frac{\log N}{2}$. Also, the array H has the special property of adjacent elements differing by 1. In that case, all the $\frac{2N}{\log N}$ blocks can be mapped to $\mathcal{O}(\sqrt{N})$ normalized blocks. For each normalized block, all possible RMQs are precomputed and stored in a table. An array $R[1 \dots \frac{2N}{\log N}]$ is used to map each of the blocks to a normalized block. To answer $\pm RMQ(i, j)$, we compute:

1. The minimum from i to the end of its block
2. The minimum of inner complete blocks between i and j
3. The minimum from starting of j's block to j

The position of the minimum of the three values is returned as the answer. All the three minimums can be found in constant time. The second minimum is found through the preprocessing done to array A'. The first and the third minimum can be found by identifying the normalized blocks they map to and then finding the RMQ for the particular ranges in those normalized blocks.

Complexity $= (\mathcal{O}(n), \mathcal{O}(n), \mathcal{O}(1))$

2 Proposed Algorithm: Exact

In this section, we propose a sublinear space, constant time algorithm to solve R1Q problem. The algorithm works for higher dimensions as well.

2.1 One Dimensional R1Q

For 1-D R1Q, the input is a linear array $A[1 \dots n]$. The query R1Q(i, j) should answer the question as to whether there exists a 1 in the subarray $A[i \dots j]$.

2.1.1 Preprocessing

Partition the array $A[1 \dots n]$ into $\frac{n}{2^p}$ blocks, each of size 2^p where p varies from $\log l$ to $\log n$. For simplicity, set $l = \frac{\log n}{2}$. For each value of p, we define two arrays

- $L[1 \dots \frac{n}{2^p}]$, where $L[i]$ holds the distance of the position where the first 1 appears from the start of the i^{th} block when the block size is 2^p
- $R[1 \dots \frac{n}{2^p}]$, where $R[i]$ holds the distance of the position where the first 1 appears from the end of the i^{th} block when the block size is 2^p

The arrays L and R will be used to answer inter-block queries. A data structure to answer the intra-block/in-block queries is still needed. We use the Four Russians trick to compute the intra-block queries. That is, for each of the blocks we maintain a query table where the answers for all possible in-block range queries would be stored.

For a block size of $l = \frac{\log n}{2}$, there are $2^l = 2^{\frac{\log n}{2}} = \sqrt{n}$ unique blocks. These are called normalized blocks. For more details on normalized blocks, refer [3]. For each normalized block i, we find the answers for all possible range queries and store in a table $T[i]$.

2.1.2 Query Execution

To answer the query R1Q(i, j), we consider two cases as given in Algorithm 14:

1. If the query (i, j) falls in the same block, we answer the query by a lookup for the particular block. We find the index of the normalized block for the range and using the Four Russians trick, we answer the query
2. If the query (i, j) falls in different blocks, we find two overlapping blocks that completely cover the range and then compute the answer from the two queries

Algorithm 13 : R1Q-Preprocess()

1: $l \leftarrow \frac{\log n}{2}$
2: // For intra-block queries
3: **for** $i \leftarrow 1$ to $\frac{n}{2^l}$ **do**
4: $\quad T[i] \leftarrow$ query table for i^{th} block
5: // For inter-block queries
6: **for** $p \leftarrow \log l$ to $\log n$ **do**
7: $\quad$ **for** $i \leftarrow 1$ to $\frac{n}{2^p}$ **do**
8: $\quad\quad L[p][i] \leftarrow$ distance where first 1 occurs from i^{th} block's start
9: $\quad\quad R[p][i] \leftarrow$ distance where first 1 occurs from i^{th} block's end
10: **return** T, L, R

When the query (i, j) are in different blocks, the range query is answered as follows. Let 2^p be the largest power of 2 present in the range. Then $p = \lfloor \log(j - i + 1) \rfloor$. At this point, we find whether 1 exists in any of the two blocks: $A[i \dots i + 2^p - 1]$ and $A[j - 2^p + 1 \dots j]$. These range queries can be answered using L and R tables corresponding to value p. Here we show how to answer the range query R1Q$(i, i + 2^p - 1)$. Same idea can be used to answer R1Q$(j - 2^p + 1, j)$.

The range $(i, i + 2^p - 1)$ occupies at most two blocks of size 2^p. Let the block containing i be ib. If the range is present entirely in a block, we find whether a 1 exists in that block by checking whether $L[ib] \leq 2^p$ is true. If the range is split across two consecutive blocks at a divider point div, we find whether a 1 exists in both the split blocks by checking whether $R[ib] \leq div - i$ or $L[ib + 1] \leq i + 2^p - 1 - div$ is true and then report the answer.

Algorithm 14 : R1Q($start$, end)

1: $l \leftarrow \frac{\log n}{2}$
2: $startblock \leftarrow \frac{start}{2^l}$
3: $endblock \leftarrow \frac{end}{2^l}$
4: // Intra-block queries
5: **if** $startblock = endblock$ **then**
6: $\quad$ **return** $T[startblock][start, end]$
7: **else**
8: $\quad$ // Inter-block queries
9: $\quad p \leftarrow \lfloor \log(end - start + 1) \rfloor$
10: $\quad$ **return** R1Q-Evaluate(p, $start$) or R1Q-Evaluate(p, $end - 2^p + 1$)

Algorithm 15 : R1Q-EVALUATE(p, $start$)

```
1: end ← start + 2^p − 1
2: startblock ← ⌈start / 2^l⌉
3: divider ← startblock · 2^p
4: if start%2^p = 1 then
5:    if L[startblock] ≤ 2^p then
6:       return true
7: else
8:    if R[startblock] ≤ divider − start or
      L[startblock + 1] ≤ end − divider then
9:       return true
10: return false
```

2.2 Higher Dimensional R1Q

Lets extend the technique to 2 dimensions. The problem statement can be redefined as: Given a 2-dimensional bit array A of order $m \times n$ and indices (i_1, i_2), (j_1, j_2), find efficiently if there exists a 1 in the subregion $A[i_1, j_1][i_2, j_2]$.

Let m and n be powers of 2. Partition the array A in different ways in blocks of size $2^i \times 2^j$ where $0 \leq i \leq \log m$, $0 \leq j \leq \log n$. For each of these blocks thus created, maintain arrays TL, TR, BL, BR of size 2^i from the four corners top left, top right, bottom left and bottom right respectively. Consider the array TL of a top left corner for a particular block. The element $TL[k] = c$, where $1 \leq k \leq 2^i$ and $1 \leq c \leq 2^j$ represents the largest rectangle from top corner point of the block till (k, c) which does not have a 1. The other arrays TR, BL, BR are defined similarly.

Now when a query rectangle say $A[i_1, j_1][i_2, j_2]$ is given, we construct four rectangles from each of the corners of the query rectangle. These four rectangles have sides that are the largest powers of two in the query range in respective dimensions i.e $2^p \times 2^q$ where $p = \lceil \log(j_1 - i_1 + 1) \rceil$ and $q = \lceil \log(j_2 - i_2 + 1) \rceil$. The answers to these four rectangles which we call as *power rectangles* are combined to get the answer for the query rectangle. We consider the partition of the array A in blocks of size $2^p \times 2^q$. It is easy to see that each of the four power rectangles of size $2^p \times 2^q$ will be present in at most four blocks of the same size which are called *split rectangles*. Now it is straightforward to find whether there is a 1 in a power rectangle using all its split rectangles and the arrays RL, TR, BL, BR.

3 Algorithm Analysis

3.1 Space Complexity

For 1D R1Q, we use Four Russians trick to solve the intra-block queries and the left and right arrays for inter-block queries. Let $l = \frac{\log n}{2}$. For intra-block queries,

$$\text{Space } = \mathcal{O}\left(\sqrt{n} \cdot \log^2 n\right) \tag{1}$$

For inter-block queries,

$$\text{Space } = \sum_{i=\log l}^{\log n} \frac{n}{2^i} \cdot 2i = \mathcal{O}\left(\frac{n \log l}{\log n}\right) \tag{2}$$

$$= \mathcal{O}\left(\frac{n \log \log n}{\log n}\right) \tag{3}$$

Hence, total space complexity will be $\mathcal{O}\left(\frac{n \log \log n}{\log n}\right)$.

For 2D R1Q, we divide the array A into rectangles of size $2^i \times 2^j$. For simplicity, let the order of A be $n \times n$ where $n = 2^k$. Let $S(i, j)$ denote the space required for a rectangle

$$S(i, j) = \frac{mn}{2^i 2^j} \cdot 4 \cdot 2^{min(i,j)} \cdot (max(i, j) + 1) \tag{4}$$

Total space required for all rectangles is given by

$$\begin{aligned}
\text{Space} &= \sum_{0\le i,j\le k} S(i,j) && (5)\\
&= \sum_{0\le i<j\le k} S(i,j) + \sum_{0\le j<i\le k} S(i,j) + \sum_{0\le i=j\le k} S(i,j) && (6)\\
&= 4n^2 \sum_{0\le i<j\le k} \frac{j+1}{2^j} + 4n^2 \sum_{0\le j<i\le k} \frac{i+1}{2^i} + 4n^2 \sum_{0\le i=j\le k} \frac{j+1}{2^j} && (7)\\
&= 8n^2 \sum_{0\le i<j\le k} \frac{j+1}{2^j} + 4n^2 \sum_{0\le j\le k} \frac{j+1}{2^j} && (8)\\
&= 8n^2 \left(8 - 6(k+1)\left(\frac{1}{2}\right)^{k+1} - 8\left(\frac{1}{2}\right)^{k+1} - 2(k+1)^2\left(\frac{1}{2}\right)^{k+1}\right) && (9)\\
&+ 4n^2 \left(4 - 4\left(\frac{1}{2}\right)^{k+1} - 2(k+1)\left(\frac{1}{2}\right)^{k+1}\right) && (10)\\
&= 8n^2 \left(8 - 6\cdot\frac{\log n+1}{2n} - 8\cdot\frac{1}{2n} - 2\cdot\frac{(\log n+1)^2}{2n}\right) && (11)\\
&+ 4n^2 \left(4 - 4\cdot\frac{1}{2n} - 2\cdot\frac{\log n+1}{2n}\right) && (12)\\
&< 64n^2 + 16n^2 && (13)\\
&= 80n^2 && (14)\\
&= \mathcal{O}(n^2) && (15)
\end{aligned}$$

Hence, the total space complexity will be $\mathcal{O}(n^2)$ which is linear.

3.2 Time Complexity

In 1D, we find the largest power of two in the query range and then use the left right tables to answer the range query. The time taken to answer the query is constant.

In 2D, for a query rectangle, we find the power rectangles and then the split rectangles for those. The range queries for a maximum of 16 split rectangles have to be answered. Answering the range query for a split rectangle is constant time. Hence the overall algorithm takes constant time.

4 Proposed Algorithm 2: Exact

We propose another algorithm to answer the queries which are of size at least m.

4.1 Preprocessing

Initially, the array A of size n is divided into two parts by a level 1 divider in the middle at $n/2$. This L1 divider holds the distance where the first 1 appears in both the left and right halves of the array. The two halves of the arrays are again subdivided into two equal parts by level 2 dividers. The level 2 dividers will be at locations $n/4$ and $3n/4$. Again, the dividers hold the distances of the first 1 which appear at left and right of the dividers. The process continues until the size of the smallest block becomes m.

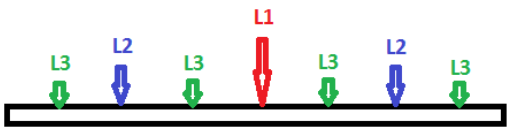

Figure 1: Different levels of dividers

4.2 Query Execution

Given a query range which is of size at least m, we can find whether a 1 exists in the range in constant time.

AM NOT ABLE TO RECALL HOW

4.3 Space Complexity

The array is so partitioned that the size of the blocks become m. Let $n = 2^k$ and $m = 2^r$. The L_1 divider takes $2(k-1)$ bits to store the positions of the first 1 in left and right halves. The two L_2 dividers each take $2(k-2)$ bits to store the positions of the first 1 in their left and right halves. This continues for L_{k-r} dividers. Then the space complexity is given as follows.

$$
\begin{aligned}
\text{Space} &= 2(k-1) + 2^2(k-2) + \cdots + 2^{k-r}(k-(k-r)) && (16)\\
&= k(2^1 + 2^2 + \cdots + 2^{k-r}) && (17)\\
&\quad - \left(2 \cdot 1 + 2^2 \cdot 2 + \cdots + 2^{k-r} \cdot (k-r)\right) && (18)\\
&= 2k(2^{k-r} - 1) - (k-r)2^{k-r+1} - 2 - 2^2(2^{k-r-1} - 1) && (19)\\
&= (r-1)2^{k-r+1} - 2k + 2 && (20)\\
&= 2(r-1)2^{k-r} - 2k + 2 && (21)\\
&= \mathcal{O}\left(\frac{n}{m} \log m\right) && (22)
\end{aligned}
$$

4.4 Time Complexity

The preprocessing time to compute the positions of the first 1 on either sides of a divider is linear. This can be achieved by computing all the L_{k-r} points first, then L_{k-r-1} points then continuing through L_3, L_2 and L_1. The process uses Dynamic Programming and the time taken is $\mathcal{O}(n)$.

4.5 Error Complexity

From the above idea, we can answer the inter-block queries but we cannot answer intra-block queries perfectly. Hence, if we consider all intra-block queries as errors, then the error ratio is calculated as follows.

$$\text{Error} = \frac{1/2 \cdot n/m \cdot {}^{m}C_2}{{}^{n}C_2} \tag{23}$$

$$= \frac{m-1}{2(n-1)} \approx \frac{m}{2n} \tag{24}$$

$$= \frac{1}{2^{k-r+1}} \tag{25}$$

To answer intra-block queries we can make use of $\log m$ bits for each block. We divide each block of size m into $\log m$ intervals and for each interval we assign a bit. This bit if set to 1 represents the range queries of more than 50% of all queries in that interval is 1. Else, the bit is set to 0. We can answer the intra-interval query using that bit. But to answer inter-interval queries we may use of a simple rule. If at least one of the bits in the intervals present in the query range is set to 1, we can say 1. Else, we say 0.

5 Proposed Algorithm: Approximate

The proposed algorithm works for both 1 and higher dimensional R1Qs. The algorithm uses the modified version of Count-Min Sketch data structure proposed in [7]. The Count-Min data structure was developed as an alternative to Count Sketch [6] and AMS Sketch [1]. The preprocessing and query execution stages of the proposed algorithm are explained in detail as follows:

5.1 One Dimensional R1Q

For 1-D R1Q, the input is a linear array $A[1 \dots n]$. The query R1Q(i, j) should answer the question as to whether there exists a 1 in the subarray $A[i \dots j]$.

5.1.1 Preprocessing

For the array $A[1 \dots n]$, we create $\log n$ Count-Min (CM) tables each of size kb, where

1. k is the number of completely independent hash functions
2. b is the number of buckets the hash functions map to

$$h_1 \dots h_k : \{1 \dots n\} \Longrightarrow \{1 \dots b\}$$

Initially, all the CM tables are initialized to 0. In the zeroth stage, we check the values of $A[1], A[2], \dots, A[n]$. If they are 1, their positions are hashed in CM table 0, onto b buckets using k different hash functions and those hashed positions are set to 1. In the first stage, we check the values of $A[1,2], A[2,3], \dots, A[n-1,n]$. If these subarrays have a 1, the starting positions of those subarrays are hashed in CM table 1, following the aforementioned rule. Similarly, in stage 2 we consider all subarrays of size 4 and hash the required values in CM table 2. The above process is continued till stage $\lfloor \log n \rfloor$. At every stage p, the information of all ranges of the form $A[i \dots i + 2^p - 1]$ are preprocessed and if they have a 1, their starting positions are hashed onto the p^{th} CM table using k different hash functions.

Algorithm 16 and 17 gives the method of preprocessing the array A to build CM tables.

Algorithm 16 : R1Q-PREPROCESS()

```
1: CM ← {0}
2: for p ← 0 to ⌊log n⌋ do
3:     for i ← 1 to n − 2^p + 1 do
4:         if A[i ... i + 2^p − 1] contains a 1 then
5:             HASH(p, i)
6: return CM
```

Algorithm 17 : R1Q-HASH(p, key)

```
1: for i ← 1 to k do
2:     CM[p][j][h_j(key)] ← 1
3: return true
```

5.1.2 Query Execution

To answer the query R1Q(i, j) we use algorithm 18. As per the technique, we find two overlapping blocks that completely cover the range. Then we find the largest power of of 2 present in the range. Let 2^l be the largest power of 2 present in the range. Then $l = \lfloor \log(j - i + 1) \rfloor$. At this point, we find whether 1 exists in any of the two blocks: $A[i \dots i + 2^l - 1]$ and $A[j - 2^l + 1 \dots j]$. As the answers to these ranges are already precomputed in the form of hashes, the query can be answered approximately in constant time.

In the preprocessing stage, we said that if a subarray $A[i \dots i + 2^p - 1]$ has a 1, the first position of the subarray i.e. i would be hashed onto the p^{th} CM table. So, the two blocks $A[i \dots i + 2^l - 1]$ and $A[j - 2^l + 1 \dots j]$ should have been preprocessed and the starting positions of these subarrays: i and $j - 2^l + 1$ should have been hashed onto the l^{th} CM table. During query execution, we check whether we have 1 present in all hashed positions of i and $j - 2^l + 1$ in the l^{th} CM table. If true, then the range $A[i \dots j]$ has a 1, with some error probability. Else, it does not. The algorithms are given in 18 and 19.

An important point to note is that we do not have to use any algorithm to answer a range query inside a word. We just use bit shifts to answer these queries. If the range (i, j) spans over different words, we compute the range queries for the three regions:

1. From i to the end of its word, using bit shifts
2. The inner words
3. From j's word to j, using bit shifts

Therefore, we just need to take care of answering the range queries for the words and not individual bits. We proceed assuming that we are trying to answer the range queries for words of the array.

Algorithm 18 : R1Q($start$, end)

```
1: p ← ⌊log(end − start + 1)⌋
2: return R1Q-IsHashed(p, start) or R1Q-IsHashed(p, end − 2^p + 1)
```

Algorithm 19 : R1Q-IsHashed(p, key)

```
1: for i ← 1 to k do
2:     if CM[p][i][h_i(key)] = 0 then
3:         return false
4: return true
```

5.2 Higher Dimensional R1Q

The problem statement can be redefined as: Given a d-dimensional bit array A of order $n_1 \cdot n_2 \cdots n_d$ and indices $(i_1, i_2, \ldots, i_d)$, $(j_1, j_2, \ldots, j_d)$, find efficiently if there exists a 1 in the subregion $A[i_1, j_1][i_2, j_2] \ldots [i_d, j_d]$.

The first linear space preprocessing algorithm for higher dimensional RMQ was given in [2]. The proposed algorithm detailed above can be extended to higher dimensions as well.

An orthotope is a generalization of a rectangle and cuboid in higher dimensions. In one dimensional R1Q, we considered two subarrays whose size is the largest power of 2 and which completely covers the range. In d dimensional R1Q, as there are 2^d corners/vertices for the range, we consider 2^d identical orthotopes whose edges are the largest powers of 2 in the range values in respective dimensions and which completely covers the range.

The preprocessing and the query execution procedures for higher dimensional R1Q are given in algorithms 20 and 21. We make use of LINEARIZE function just for simplicity. It converts a d-tuple to a number. The notation '|' is used for *or*. In the query execution algorithm, there find 2^d different points/keys as there are 2^d corners for an orthotope.

Algorithm 20 : dR1Q-PREPROCESS()

```
1:  CM ← {0}
2:  for i_1 ← 1 to n_1 do
3:      ⋱
4:    for i_d ← 1 to n_d do
5:      for p_1 ← 1 to ⌊log(n_1 − i_1 + 1)⌋ do
6:          ⋱
7:        for p_d ← 1 to ⌊log(n_d − i_d + 1)⌋ do
8:          if A[i_1, i_1 + 2^{p_1} − 1] ... [i_d, i_d + 2^{p_d} − 1] contains a 1 then
9:            p ← LINEARIZE(p_1, p_2, ..., p_d)
10:           key ← LINEARIZE(i_1, i_2, ..., i_d)
11:           HASH(p, key)
12: return CM
```

6 Algorithm Analysis

In this section we present the space and time complexity of the proposed algorithm. We show that the algorithm takes very less space and time but comes

Algorithm 21 : dR1Q($(i_1, i_2, \ldots, i_d)$, $(j_1, j_2, \ldots, j_d)$)

1: **for** $k \leftarrow 1$ to d **do**
2: $\quad p_k \leftarrow \lfloor \log(j_k - i_k + 1) \rfloor$
3: $p \leftarrow$ Linearize$(p_1, p_2, \ldots, p_d)$
4: $[key_1, \ldots, key_{2^d}] \leftarrow$ Linearize$(i_1|(j_1 - 2^{p_1} + 1), \ldots, i_d|(j_d - 2^{p_d} + 1))$
5: **for** $k \leftarrow 1$ to 2^d **do**
6: $\quad$ **if** R1Q-IsHashed(p, key_k) **then**
7: $\quad\quad$ **return** true
8: **return** false

with an expense of errors. Finally, we discuss the limitations and possible use-cases of the method.

6.1 Space Complexity

The common algorithms for R1Q take linear space. We shall derive the space complexity of the proposed algorithm. If we have use k hash functions and b buckets for the CM table, the size of a single CM table will be kb. For one dimensional R1Q for an array of size n, we will have $\log n$ tables. Hence,

$$\text{Space for 1-D R1Q} = \mathcal{O}(kb \log n)$$

For d dimensional R1Q for an array of size $n_1 \cdot n_2 \cdots n_d$, the size of each table will be kb only, but the number of tables now will be $\log n_1 \cdot \log n_2 \cdots \log n_d$. Therefore,

$$\text{Space for } d\text{-D R1Q} = \mathcal{O}(kb \log n_1 \log n_2 \cdots \log n_d)$$

6.2 Time Complexity

In preprocessing, from each of the points in the d dimensional array, we find all possible orthotopes and check whether it has 1. As there are $n_1 \cdot n_2 \cdots n_d$ points and there are $\mathcal{O}(\log n_1 \cdot \log n_2 \cdots \log n_d)$ orthotopes from each point,

$$\text{Preprocessing time for } d\text{-D R1Q} = \mathcal{O}(n_1 \cdots n_d \log n_1 \cdots \log n_d)$$

For 1-D R1Q, we want to find if there is 1 in two overlapping blocks. This can be done in constant time. As the starting positions of the blocks are already hashed onto the CM tables if they contain a 1, we just need to hash and check they indeed contain a 1. For d-D R1Q, during query execution we would be checking 2^d orthotopes by hashing. Hence,

$$\text{Execution time for } d\text{-D R1Q} = \mathcal{O}(2^d)$$

6.3 Limitations

The proposed algorithm is a randomized algorithm. The accuracy of the method is inversely proportional to the randomness of the input array. If the input array has a particular pattern, say, alternate blocks of ones and zeros, it would generate more errors compared to if we had used a randomized array. The results are presented in the implementation section. In summary, the method is best suited for randomized arrays.

One important limitation of the method is false positives. If a range has a 1, then the answer to the range query will always be true. But, if the range does not have a 1, the answer can be either false or true. So, depending on the usecases, this property may be either an advantage or a disadvantage.

6.4 Error Bounds

We follow a similar line of reasoning for giving errors as presented in [7]. We could use the Count Min Sketches to store the values for both the algorithms. Then for a range query, the estimated value for the i^{th} given by the sketch say $\hat{a}[i]$ will always be greater than or equal to the real value $a[i]$.

Given ϵ' and δ which means the error in answering a query is within a factor of ϵ' with probability $1-\delta$. That is with probability $1-\delta$, $a[i] \leq \hat{a}[i] \leq a[i]+\epsilon'|A|$, where $A = \sum a_i$. In this problem, we consider $|A|$ to be $|L|$ which is the sum of all left offsets in all blocks of size 2^p. Similar argument holds good for right offsets. Mathematically, the approximate value got from count min sketch is given by

$$\hat{a}[i] \leq a[i] + \epsilon'|L| \tag{26}$$

$$\leq a[i] + \left(\epsilon' \frac{|L|}{2^p}\right) 2^p \tag{27}$$

$$\leq a[i] + \epsilon 2^p \quad \left(\text{where } \epsilon = \frac{\epsilon'|L|}{2^p}\right) \tag{28}$$

Now, the bucket size b increases to

$$b = \frac{e}{\epsilon'} \tag{29}$$

$$= \frac{e}{\epsilon} \frac{|L|}{2^p} \tag{30}$$

Here, 2^p represents the size of a query range which can be answered using this count min sketch table. Consider all queries Q of size 2^p. Consider two adjacent blocks A and B of size 2^p. How many queries of length 2^p will cross the divider of A and B? It is 2^p queries. And how many of them can be answered incorrectly? At most 2ϵ fraction of 2^p queries can be answered incorrectly. How?

A query Q can be entirely in block A or span across both the blocks or entirely in B. To answer the query Q, we find the approximate right offset in A and approximate left offset in B. At most ϵ of all 2^p queries can be wrong because of left offset of A. Similarly, at most ϵ of all 2^p queries can be wrong because of right offset of B. So, at most 2ϵ fraction of total queries can be answered incorrectly and the fraction is independent of the value of 2^p. If we consider a general query which is a collection of two 2^p sized queries, then the total error is at most 4ϵ fraction of the number of queries. Hence, the error of the algorithm is at most 4ϵ.

For 2D, we consider the rectangles are of size $2^p \times 2^q$. Assuming $p < q$, we have

$$\hat{a}[i] \leq a[i] + \epsilon' |L| \tag{31}$$

$$\leq a[i] + \left(\epsilon' \frac{|L|}{2^q}\right) 2^q \tag{32}$$

$$\leq a[i] + \epsilon 2^q \quad \left(\text{where } \epsilon = \frac{\epsilon' |L|}{2^q}\right) \tag{33}$$

Now, the bucket size b increases to

$$b = \frac{e}{\epsilon'} \tag{34}$$

$$= \frac{e}{\epsilon} \frac{|L|}{2^q} \tag{35}$$

Note that $|L|$ can be at most n^2.

7 Other Shapes

Till now, we have focused our attention to answering range queries in an array, a rectangle, a cuboid and in general, an orthotope. We can also use the above method to answer range queries for other shapes like right angled triangle, isosceles triangle and extending further to regular polygons and in general, regular polytopes. Similarly, the range queries of many more shapes could be answered.

7.1 Right Angled Triangle

Consider the right angled triangle ABC as shown in Figure 2. We have to find whether there exists a 1 in a right angled triangle from a 2-dimensional bit array given the slope of the hypotenuse slope. We preprocess the input array and store the information of right angled triangles from every point with height which is a power of 2. We should also have the preprocessed CM tables for rectangular queries that we have already seen. During preprocessing, all those right angled

triangles which have a vertical height of 2^p and with a 1 inside them will be hashed onto the p^{th} CM table. The right angled triangle queries can now be answered with the preprocessed data.

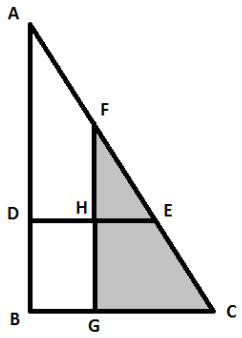

Figure 2: Range query in a right angled triangle

To answer the range query of a right triangle as in ABC, we do the following. Divide the triangle ABC into the following three regions and compute the range queries for them.

1. Triangle ADE, where AD is the highest power of 2 in AB
2. Triangle FGC, where $AD = FG$
3. Rectangle $DHGB$

The range queries of the two triangles would already be precomputed in the preprocessing stage. The range query of rectangle $DHGB$ can be answered as discussed in Section 5.2.

Right angled triangles with variable hypotenuse slope can also be answered. Consider the triangle ABC where AB is a power of 2. If D and E are points on BC such that if

- $BE < BD$, then ABE does not contain a 1
- $BE \geq BD$, then ABE contains a 1

then we store the index D for point A for the size AB. In this way using these index points we can answer range queries for right triangles with horizontal and vertical sides and a hypotenuse of any slope. We have to construct separate tables if we want to answer triangles that are vertical reflections of triangle ABC.

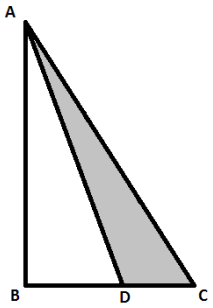

Figure 3: Range query in a right angled triangle with variable slope

7.2 Isosceles Triangle

Consider the isosceles triangle ABC as shown in Figure 4. The triangle can be split into four right triangles AFL, ALG, DBH and EIC and a rectange $JKIH$, where $AL = DH = EI = 2^p$.

We could answer an isosceles triangle with a fixed slope with a combination of preprocessed right triangles and rectangle. The range queries of isosceles triangles with variable slope can be answered if we use the right angled triangles with variables slopes as explained in the previous section.

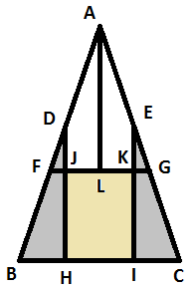

Figure 4: Range query for an isosceles triangle

7.3 Regular Polygons

The range queries could be answered even for regular polygons. It is easy to see that any regular polygon with number $n \geq 5$ sides can be covered with right triangles of four different types and rectangles. So, it is easy to answer queries with right triangles and rectangles.

However, we want to reduce the number of queries performed if we use preprocessed polygons of same shape. We represent a straight orientation regular polygon by (n, s), where n is the number of sides and s is the side length. From all points, preprocess all the (n, l) polygons where $l = 2^p$. Now, to answer a query of a regular polygon (n, s), from each of the vertices we construct a polygon (n, l) where $l = 2^p$ and $2^p \leq s < 2^{p+1}$.

The two cases to consider for polygons is:

1. When n is even, the smaller polygons (n, l) cover the entire range of the bigger polygon (n, s)
2. When n is odd, the smaller polygons (n, l) cover the entire range of the bigger polygon (n, s) iff $l \geq s/\left(1 + \sin\left(\frac{n-2}{n} \cdot 90^\circ\right)\right)$

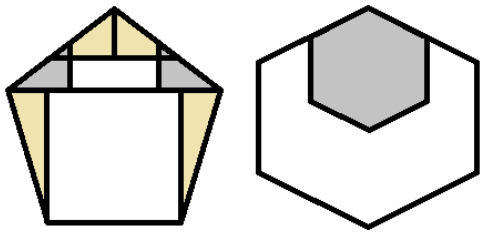

Figure 5: Sample polygons with odd and even number of vertices

As we have a limitation for polygons (n, s) where n is odd, we can cover these polygons with right angled triangles and rectangles and answer the range queries as shown in Figure 5. In fact, some irregular convex polygon queries can also be answered if it meets the following criteria.

7.4 Regular Polytopes

The arguments given for polygons can be extended to regular polytopes as well. Let a polytope be represented as (v, s), where v is the number of vertices and s is the edge/side length. From all points, preprocess all the polytopes (n, l) of same shape where $l = 2^p$. Now, to answer a query of a regular polytope (n, s), from each of the vertices we construct a polytope where $l = 2^p$ and $2^p \leq s < 2^{p+1}$.

When n is even, the smaller polytopes (n, l) cover the entire range of the bigger polytope (n, s). When n is odd, hopefully some methods similar to the ones we discussed for polygons with odd number of vertices may exist.

8 Triangular Queries

Here, we describe a method to answer the range query for any triangle. Consider the diagram as in Figure 6. Assume that we can answer the number of 1's present in a rectangle or any right angled triangle. Then the R1Q for any triangle EFC can be answered from the formula

Number of 1's in triange EFC = Number of 1's in rectangle $ABCD$ - (Number of 1's in right triangles AEC, EBF and CDF combined)

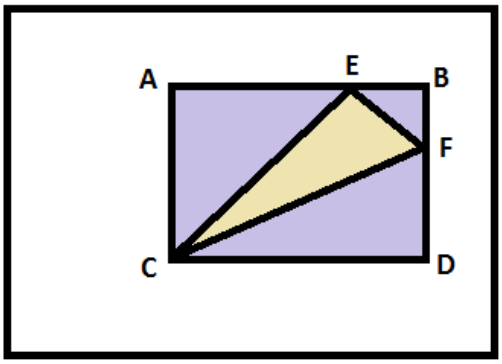

Figure 6: Rotated Triangles

8.1 Exact Algorithm

Here, we describe an exact algorithm to answer the range query of any triangle. It is enough if we can find the number of 1's in a rectangle and number of 1's in a right triangle. Then using the above relation, we can find the number of 1's in any triangle.

8.1.1 Number of 1's in a Rectangle

From hereon, we assume the input matrix is of size $N = n \times n$. Every point P stores the number of 1's present in the rectangle with opposite vertices of top left corner of the input matrix and P as shown in the diagram. Now, to answer the number of 1's in a rectangle with opposite vertices P and Q, we just need to subtract P from Q.

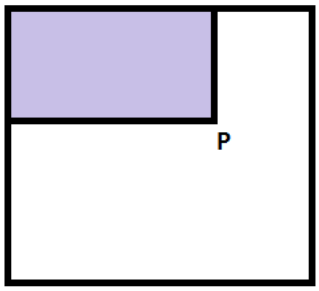

Figure 7: Rectangles

8.1.2 Number of 1's in a Right Triangle

We preprocess the right triangles for one orientation as follows. Similar thing can be done to other orientations as well. From every point, we consider the right triangles which have a height, a power of two. For each of these right trianges, for different lengths of the base, we store the number of 1's. This means, we have the answers to all possible right triangles with a height a power of two.

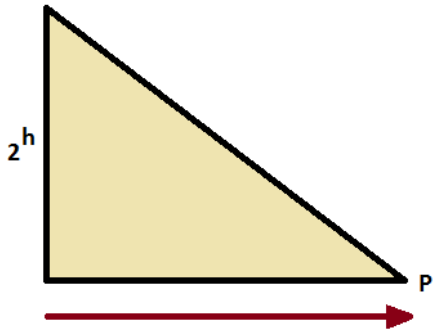

Figure 8: Triangles

To answer the number of 1's in a general right triangle, we divide the general right triangle into rectangles and at most $\mathcal{O}(\log N)$ right triangles with height a power of two as shown in the diagram.

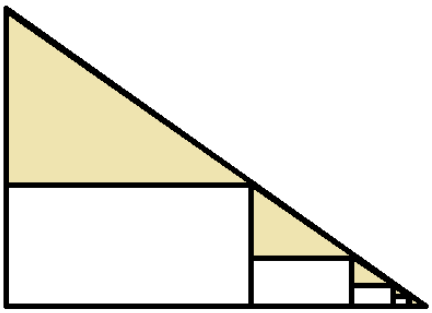

Figure 9: Triangles

8.1.3 Algorithm Complexity

There are N points. The different heights which are powers of two from a particular point are $\mathcal{O}(\log N)$. Each of these triangles can have at most $\mathcal{O}(\sqrt{N})$ base length. Each of these triangles with different bases have to store the count of 1's in $\mathcal{O}(\log N)$.

$$\begin{aligned}\text{Space complexity} &= \mathcal{O}(N \cdot \log N \cdot \sqrt{N} \cdot \log N) \\ &= \mathcal{O}(N^{3/2} \log^2 N)\end{aligned}$$

The time complexity of answering a general triangle is same as the time complexity of answering a general right angled triangle which is $\mathcal{O}(\log N)$

$$\text{Time complexity} = \mathcal{O}(\log N)$$

8.2 Approximation Algorithm

Here, we retain the mechanism to answer a query rectangle, but to answer the right angled triangles approximately. Hence, a general rectangle is answered approximately.

The idea is as follows. Consider a right triangle whose height is a power of two. As in the exact algorithm, we do not store the number of ones for all bases. Let C denote the number of 1's in a particular polyong. We store the number of points for points $P_0, P_1, \ldots P_k$ such that

$$\begin{aligned} C(ABP_0) &< (1+\epsilon)^0 \\ (1+\epsilon)^0 \leq C(AP_0P_1) &< (1+\epsilon)^1 \\ &\cdots \\ (1+\epsilon)^k &\leq C(AP_kC) \end{aligned}$$

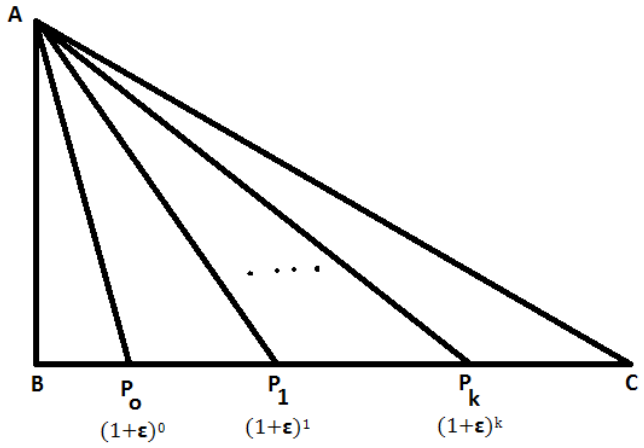

Figure 10: Triangles Approx

8.2.1 Algorithm Complexity

In this algorithm we are only storing the points at which the number of points will be greater than or equal to a power of $1 + \epsilon$. Hence, the number of points for the base of a right triangle can be $\mathcal{O}(\log_{1+\epsilon} N)$.

$$\begin{aligned}\text{Space complexity} &= \mathcal{O}(N \cdot \log N \cdot \log_{1+\epsilon} N \cdot \log N) \\ &= \mathcal{O}(N \log^2 N \log_{1+\epsilon} N)\end{aligned}$$

To answer a right triangle query ABP, through a binary search we have to find P such that $P_i < P \leq P_{i+1}$. This requires $\mathcal{O}(\log_{1+\epsilon} N)$ time. And there can be $\mathcal{O}(\log N)$ right triangles in a general triangle. Hence

$$\text{Time complexity} = \mathcal{O}(\log N \log_{1+\epsilon} N)$$

Now, to answer the number of 1's in any triangle ABP, let the approximate answer be denoted as $\hat{t}$ and the exact answer be denoted as t, then

$$\frac{t}{(1+\epsilon)} \leq \hat{t} \leq t(1+\epsilon)$$

9 Implementation & Results

We implemented the proposed algorithm for both 1-D and 2-D R1Q in C++. Our tests were performed using Intel Core i5-2410M processor, 2.30 GHz speed, 6GB RAM on Windows 7 OS. Five hash functions were used throughout our experiments i.e $k = 5$. We have used 64 bits as the word size.

9.1 One Dimensional R1Q

We used $n = 6 \cdot 10^6$. Two sets of patterns of input data were tested. We have compared our algorithm with three other algorithms: a) scan algorithm b) binary search algorithm, where we store the positions of ones and binary search using the range c) sparse table, similar to Algorithm 9 but stores the presence or absence of 1 instead of the position of the minimum element. Our algorithm is depicted in blue, scan algorithm in red, binary search algorithm in green and sparse stable algorithm in yellow. The number of buckets is set to 8 words.

9.1.1 Alternate 1-0 Segments

In this pattern, the input array consists of segments of size *segsize*. In each segment, the first *segones* bits are set to 1 and the remaining 0. Figures 11 and 12 give the execution time and error percentage of our algorithm for six different *segsize* values of 10^1 to 10^6. For each of these *segsize* the *segones* take the values of 10% to 90% of the *segsize*. We have used 10^7 random queries.

The scan algorithm was taking a lot of time. Hence it was considered as 0. Our algorithm ran faster than the binary search and sparse table algorithm. Also, the error percentage was found to be indirectly proportional to *segsize* and directly proportional to *segones*.

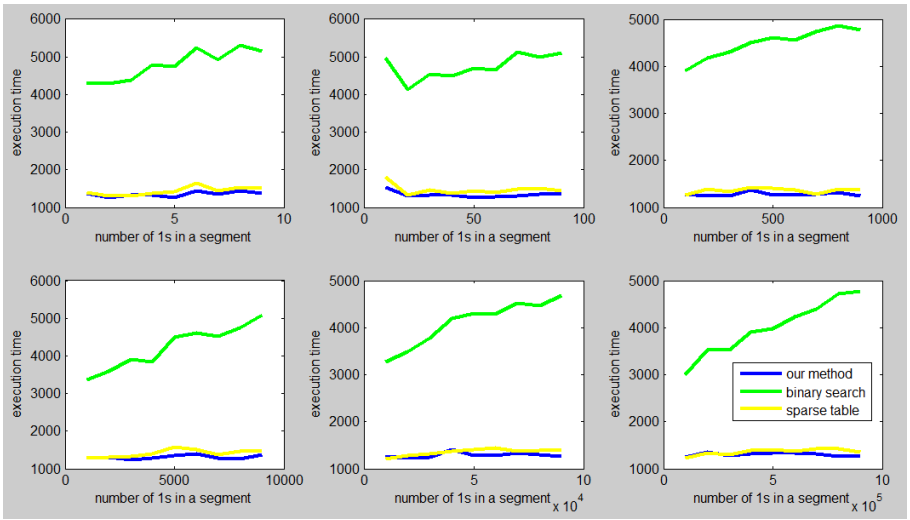

Figure 11: 1-D: Alternate 1-0: Execution time vs Number of 1s in a segment. For 6 segments of length 10^1 to 10^6

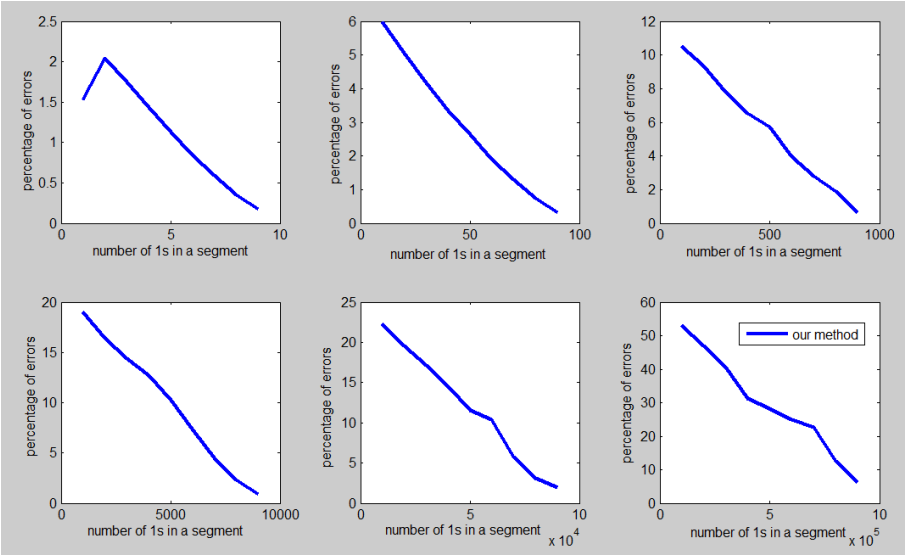

Figure 12: 1-D: Alternate 1-0: Error percentage vs Number of 1s in a segment. For 6 segments of length 10^1 to 10^6

9.1.2 Random Segments

In this pattern, the input array consists of segments of size *segsize*. Each segment was randomly assigned zeros or ones. The *segsize* values was varied from 10^0 to 10^5. Figures 13 and 14 give the execution time and error percentage of our algorithm for small (< 1000 size) and large queries (> 1000 size).

We see from the plots that the execution time of our algorithm is better than that of binary search or sparse table algorithm. Also that the error percentage is very less for smaller segments and the error percent almost doubles when *segsize* increases by a factor of 10.

9.2 Two Dimensional R1Q

We used a two dimensional input array of size $n_1 = 1000$ (rows) and n_2 (columns). The execution time is compared with the naive scan algorithm which is shown in red and our algorithm is shown in blue.

9.2.1 Random words set to all ones

In this pattern, some random words are set to all ones. We used $n_2 = 25600$. Figure 15 gives the execution time and error percentage of our algorithm. The

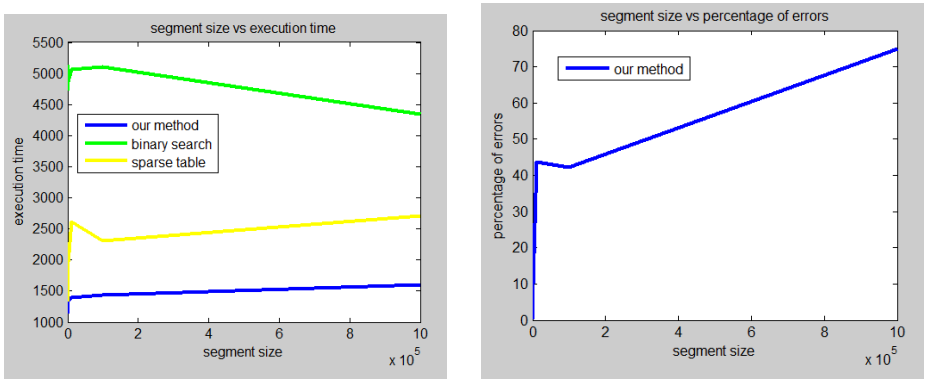

Figure 13: 1-D: Random segments: Small queries: Exec time and Error percent

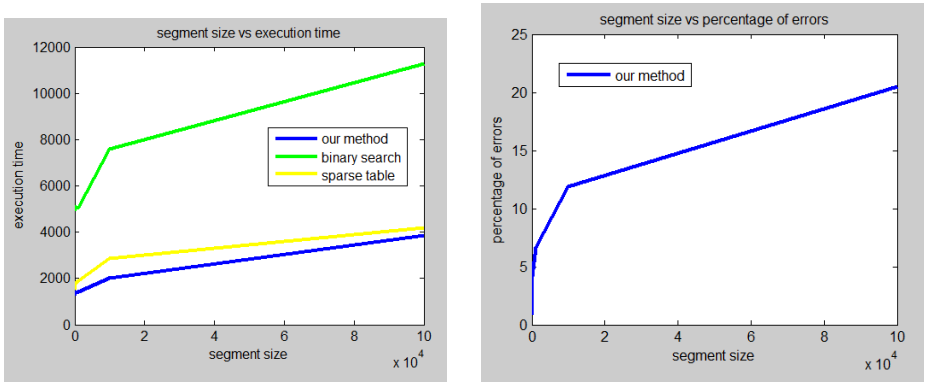

Figure 14: 1-D: Random segments: Large queries: Exec time and Error percent

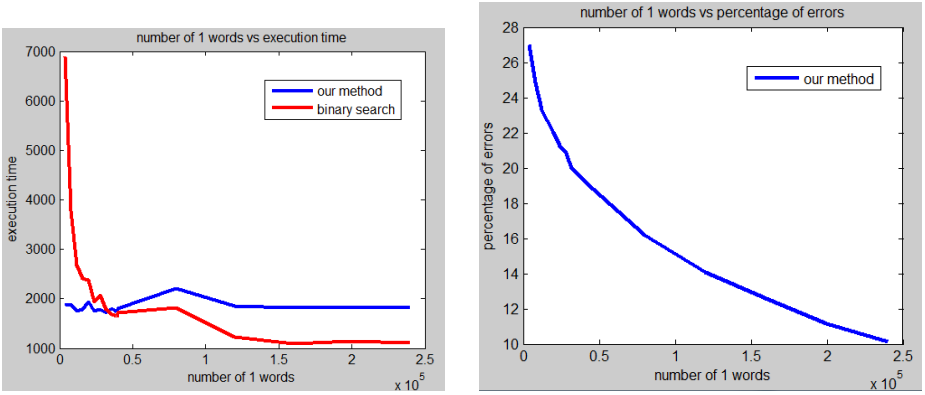

Figure 15: 2-D: Random words set to ones: Exec time and Error percent

number of random words set to all ones are varied from 4K to 240K in increments of 4K.

The scan algorithm ran faster than our algorithm after a certain point. We see that the error percentage is inversely proportional to the number of words set to all ones.

9.2.2 All words set to random numbers

In this pattern, all the words are set to random numbers. We vary n_2 from 128 to 32768 in factors of 2. Figure 16 gives the execution time and error percentage of our algorithm.

The scan algorithm ran faster than our algorithm. We see that the error percentage is inversely proportional to the number of columns. Also that error percentage is less because of more randomness in the input array.

10 Conclusion

We surveyed some existing algorithms to solve RMQ problem. Then we proposed an approximation algorithm to solve the 1-D R1Q problem with sublinear space and constant time. The algorithm was later extended to be used in higher dimensions with sublinear space and time as a function of dimensions.

The proposed algorithm was implemented in C++ and the results were plotted. The graphs showed that the algorithm works well with big data and with longer queries. Also, that the algorithm works very well with randomized data set. As

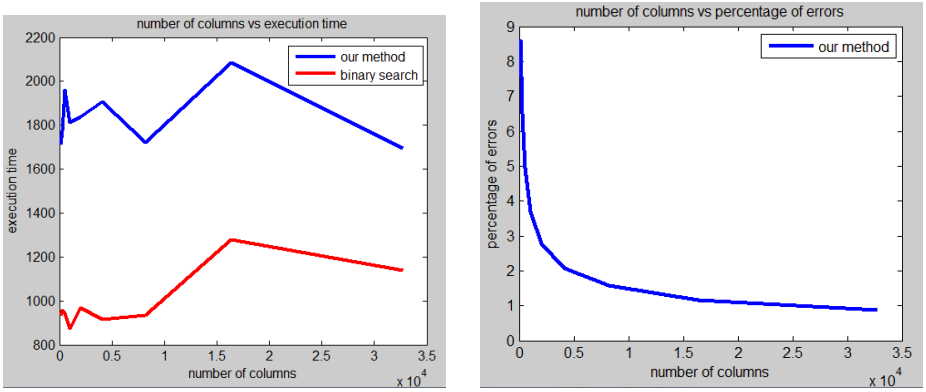

Figure 16: 2-D: All words set to random numbers: Exec time & Error percent

the error percentage is less and the space required is very less, the algorithm can be used in different applications.

References

[1] Matias Y. Szegedy-M. Alon, N., *The space complexity of approximating the frequency moments*, Proceedings of the ACM Symposium on Theory of Computing (1996), 20–29.

[2] Yuan H. Atallah, M. J., *Data structures for range minimum queries in multidimensional arrays*, Proceedings of the ACM Symposium on Discrete Algorithms (2010), 150–160.

[3] Farach-Colton M. Bender, M.A., *The lca problem revisited*, Proc. LATIN, LNCS.

[4] Farach-Colton M.-Pemmasani G. Skiena S. Sumazin P. Bender, M.A., *Lowest common ancestors in trees and directed acyclic graphs*, J. Algorithms **57** (2005), 75–94.

[5] Vishkin-U. Berkman, O., *Recursive star-tree parallel data structure*, SIAM J. Comput. **22** (1993).

[6] Chen-K. Farach-Colton M. Charikar, M., *Finding frequent items in data streams*, Procedings of the International Colloquium on Automata, Languages and Programming (ICALP) (2002), 693–703.

[7] Muthukrishnan-S. Cormode, G., *An improved data stream summary: The count-min sketch and its applications*, J. Algorithms **55** (2005), 58–75.

[8] P. Daniel, *Range minimum query and lowest common ancestor.*

[9] Heun-V. Fischer, J., *Theoretical and practical improvements on the rmq-problem, with applications to lca and lce*, Proc. CPM. **4009** (2006), 36–48.

[10] Bentley-J.L. Tarjan-R.E. Gabow, H.N., *Scaling and related techniques for geometry problems*, Proc. of the ACM STOC (1984), 135–143.

[11] Tarjan-R.E. Gabow, H.N., *A linear time algorithm for a special case of disjoint set union*, Proceedings of the 15th Annual ACM Symposiumon Theory of Computing (1983), 246–251.