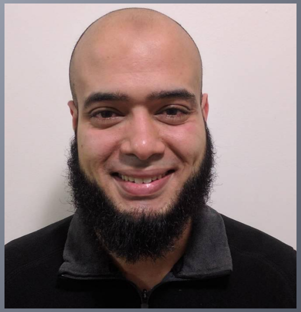
THINK FAST

Competitive Programming

From Problem 2 Solution in O(1)

Cumulative (Prefix) Sum

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University



Range Query on arrays

- Given array of N Numbers and Q queries [Start-end], find in the range/interval:
  - range **sum**/max/min/average/median/lcm/gcd/xor
  - number of elements repeated K times (k = 1 = distinct)
  - **position** of 1st index with **accumulation >=** C
  - the **smallest** number < S (or [their count]())
  - Value repeats **exactly once** or **most frequent**
  - Find the [kth elemnt]() in the sorted distinct list of range
- Brute force is O(NQ), can we do better?
  - Preprocessing algorithms / Data Structures

Range Sum

- Range sum problem
  - The easiest range query problem is the range sum
  - E.g. A[] = {2, 1, 4, 5, 3, 7}
  - Given 10^6 query: What is sum of range:
  - (0, 5) = 2 + 1 + 4 + 5 + 3 + 7 = 22
  - (1, 5) = 1 + 4 + 5 + 3 + 7 = 20
  - (2, 4) = 4 + 5 + 3 = 12
  - We can just loop and sum the array! But this is O(NQ)
  - Or try to **pre-process the array**, such that we can answer the queries much faster!
  - Let’s pre-process in O(N) and answer all queries in O(Q)

Cumulative Sum

- Cumulative (or prefix sum) **array**
  - Let’s create a new array such that S[i] = A[0]+A[1]..A[i]
  - Let’s rewrite that: S[i] = A[0]+A[1]..+A[i-1]+A[i]
  - But, S[i-1] = A[0]+A[1]..+A[i-1]
  - Then, S[i] = S[i-1] + A[i]
  - Intuitively, S[i], the sum of previous numbers + current
  - A[] = {2, 1, 4, 5, 3, 7}
  - S[] = {2, 3, 7, 12, 15, 22}
  - Notice, S[5] = Sum(0, 5) = 22
  - This means, S[] can answer any query of format Sum(0, E) in O(1) as S[E], which is great
  - Can we extend to Sum(S, E)

Cumulative Sum

- Cumulative (or prefix sum)
  - A[] = {2, 1, 4, 5, 3, 7}
  - S[] = {2, 3, 7, 12, 15, 22}
  - Can we express (2, 4) as cumulative **sums**? Yes
  - Range(2, 4) = Range(0, 4) - Range(0, 1)
  - 4, 5, 3 = {2, 1, 4, 5, 3} - {2, 1}
  - Then Sum(2, 4) = S[4] - S[2-1]
  - Again, we can answer such a query in O(1)

Cumulative Sum - code 1

```
int sum_range1(int S, int E, vector<int> & cum_sum) {
  if(S == 0)
    return cum_sum[E];

  return cum_sum[E] - cum_sum[S-1];
}

void zero_based() {
  vector<int> A = { 2, 1, 4, 5, 3, 7 };
  vector<int> S(A.size(), 0);

  //pre-processing: Compute cumulative sum array
  for (int i = 0; i < (int) A.size(); i++)
    S[i] += (i == 0) ? A[i] : A[i] + S[i - 1];

  cout<<sum_range1(0, 5, S)<<"\n";
  cout<<sum_range1(1, 5, S)<<"\n";
  cout<<sum_range1(2, 4, S)<<"\n";
}
```

Cumulative Sum - code 1

- In our code, we naturally did the usual 0-based indexing, however
  - While building the array, we care of S[0] case
  - In queries, we handle S(0, E) different from S(S, E)
  - However, pushing everything to 1 based makes things easier
  - Should we always do it? It is up to you. 0-based is more sync with your remaining coding usually

Cumulative Sum - code 2

```
int sum_range2(int S, int E, vector<int> & cum_sum) {
  return cum_sum[E] - cum_sum[S-1];
}

void one_based() {
  vector<int> A = {0, 2, 1, 4, 5, 3, 7 }; // let A[0] = 0
  vector<int> S(A.size(), 0);

  //pre-processing: Compute cumulative sum array: Start from 1
  for (int i = 1; i < (int) A.size(); i++)
    S[i] += A[i] + S[i - 1];

  // 1-based queries
  cout<<sum_range1(1, 6, S)<<"\n";
  cout<<sum_range1(2, 6, S)<<"\n";
  cout<<sum_range1(3, 5, S)<<"\n";
}
```

Range Max Query

- RMQ
  - Let’s change the problem
  - Instead of finding the range sum, find the max in a range
  - E.g. A[] = {2, 1, 4, 5, 3, 7}
  - (2, 4) = max(4, 5, 3) = 5
  - Can we do something similar?
  - More complex data structures (e.g. Segment Tree) or Algorithms (such as sparse table DP) are needed

2D Sum Queries

- From 1D to 2D
  - What if instead 1D array, we have 2D one
  - Query is to find a rectangle sum
  - E.g. Sum((2, 4), (5, 7)) where (2, 4) is the top left corner of a sub-matrix?
  - The idea is as following:
  - Create new Array S.
  - For each row in A, create its cumulative sum in S
  - In S, in-place, create cumulative sum for each column
  - Now S(i, j) = Sum((0, 0), (i, j))

2D Sum Queries

Accumulate each row

1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

1	3	5	9	10
3	7	8	13	15
2	5	8	10	14
4	5	10	14	20
6	9	11	12	15

8 = row 3 +4 + 1

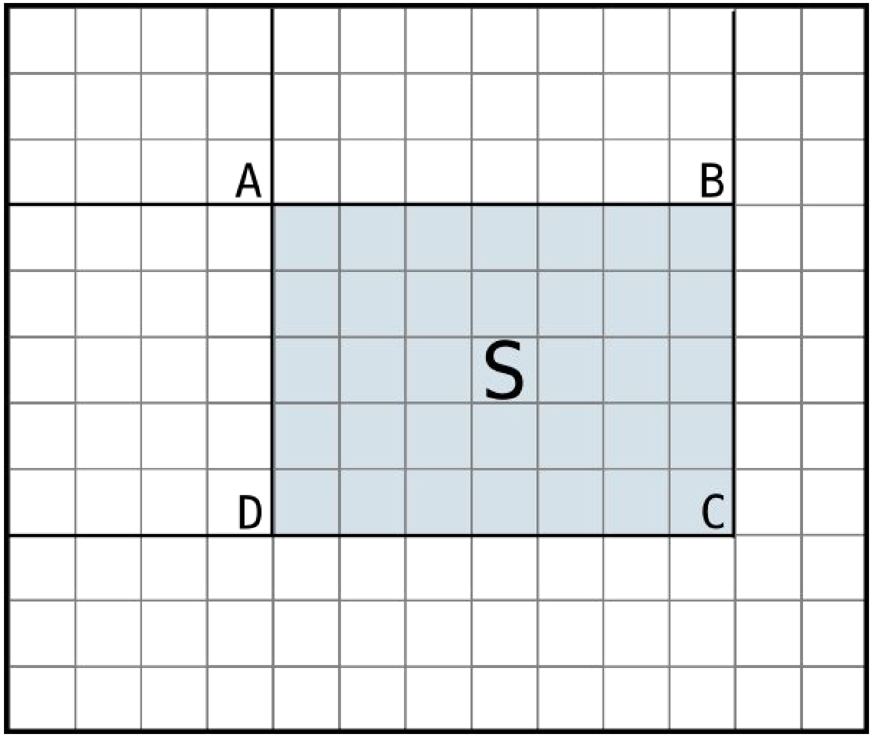
Accumulate each column

1	3	5	9	10
4	10	13	22	25
6	15	21	32	39
10	20	31	46	59
16	29	42	58	74

32 = col 9 +13 + 10
32 = sum((0, 0), (2, 3))

1	2	2	4
3	4	1	5
2	3	3	2

2D Sum Queries

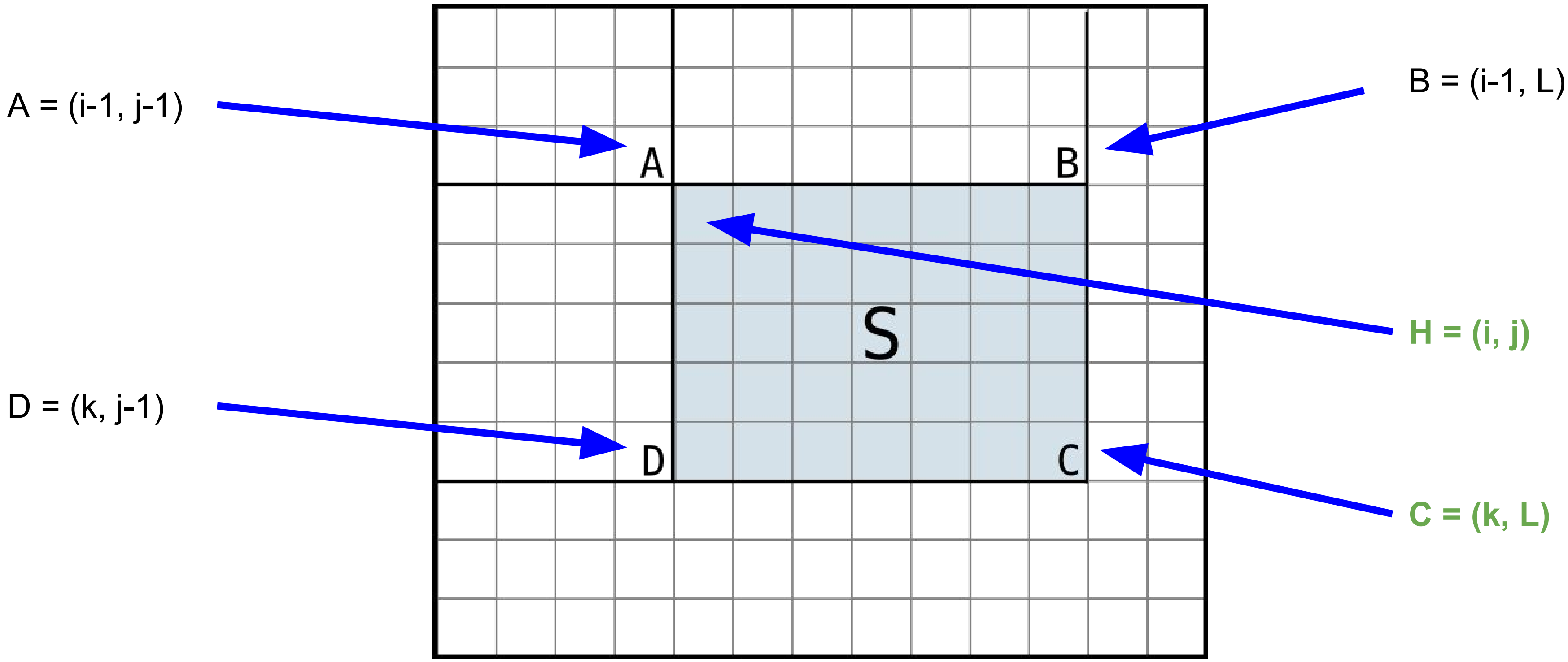


Src: http://www.nongnu.org/rapp/doc/rapp/integral.png

What about computing submatrix S?

- Assume S has bottom right position C
- sum of C is a bigger rectangle
- Let's remove B
- Let's remove D
- Now, A is removed **twice**
- Let's Add A
- So Sum(S) = C - B - D + A
- So 4 values from the 2D accum matrix are enough to compute the sub-matrix sum
- Computing indices of A, B, D is easy

2D Sum Queries



S = C - B - D + A

Src: http://www.nongnu.org/rapp/doc/rapp/integral.png

2D Sum Queries: 1-based code

```
// sum((i, j) (k, l)) where (k, l) is the bottom right
int sum_range(int i, int j, int k, int l, vector<vector<int>> & S) {
  return S[k][l] - S[k][j-1] - S[i-1][l] + S[i-1][j-1];
}

void accumSum2D() {
  // 1-based matrix
  // Append extra top row and col with zero
  vector<vector<int>> A =
      { { 0, 0, 0, 0, 0, 0 },
        { 0, 1, 2, 2, 4, 1 },
        { 0, 3, 4, 1, 5, 2 },
        { 0, 2, 3, 3, 2, 4 },
        { 0, 4, 1, 5, 4, 6 },
        { 0, 6, 3, 2, 1, 3 }, };

  // Accumulate each row
  for (int i = 1; i < (int) A.size(); i++)
    for (int j = 1; j < (int) A[0].size(); j++)
      A[i][j] += A[i][j-1];

  // Accumulate each col
  for (int j = 1; j < (int) A[0].size(); j++)
    for (int i = 1; i < (int) A.size(); i++)
        A[i][j] += A[i-1][j];

  // 1, 5, 2
  // 3, 2, 4
  cout << sum_range(2, 3, 3, 5, A) << "\n";
}
```

3D Sum Queries

- Same logic
  - Say array is [i][j][k]
  - Accumulate over i
  - Accumulate over j
  - Accumulate over k
  - Now, we S[i][j][j] is cube sum (0, 0, 0) to (i, j, k)
  - Your turn, write function to compute the cub sum

Accumulate sum Apps

- 1D
  - Comes regularly as a small functionality in many problems
  - Practice: CF433-D2-B Kuriyama Mirai's Stones
- 2D
  - Popular way in computing the largest submatrix sum in 2D matrix
  - Used in computer vision field
- 3D
  - Hmm...Rarely

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً