Cumulative Frequency Array: For Range Sum Queries

(Arup Guha, University of Central Florida)

Imagine the problem of having a list of numbers and needing to calculate the sum of any contiguous range of those numbers. For example, if the array stored:

Index	0	1	2	3	4	5	6	7	8
Value	3	12	6	9	17	4	3	2	19

and we were asked to find the sum of the values stored from index 2 to index 7, we could just add 6 + 9 + 17 + 4 + 3 + 2 = 41.

But…this is rather inefficient, especially for queries on large ranges!!!

Another way to store this same information is to store in a particular index the sum of the values up to that index in the original list. This information is typically known as cumulative frequency of the list. The corresponding cumulative frequency array for the list shown above is:

Index	0	1	2	3	4	5	6	7	8
Value	3	15	21	30	47	51	54	56	75

Now, if we want to know the sum of the values in the original array from index 2 to index 7, we take the value in index 7 of this array, 56, and subtract from it the value in index 1, 15, to get 56 – 15 = 41.

Basically, what we're doing is as follows (assume the original array is called a):

(a[0] + a[1] + a[2] + a[3] + a[4] + a[5] + a[6] + a[7]) – (a[0] + a[1]) =

a[2] + a[3] + a[4] + a[5] + a[6] + a[7]

In this manner, we can get the sum of any contiguous array by subtracting two values from our cumulative frequency array. Our special case is when the low bound is index 0, and then we don't subtract anything. Another way to handle this special case is to add an extra array index on the left:

Index	0	1	2	3	4	5	6	7	8	9
Value	0	3	15	21	30	47	51	54	56	75

In this situation, all of our array indexes are shifted over by 1, but an original range starting at index 0 can be handled without a special case. Either method can be used to handle ranges starting at the beginning.

Cumulative Frequencies: Two-dimensional Arrays

We can use cumulative frequencies in two dimensional arrays as well, to quickly get the sums of any contiguous rectangle.

First, let's go over a quick example of how to obtain cumulative frequencies for a two dimensional array. Let the following be our input array:

2	3	4	8
1	2	6	5
6	3	9	2

We first run our cumulative frequency code for each row to obtain:

2	5	9	17
1	3	9	14
6	9	18	20

Then, we redo the same code, but for each column, to get:

2	5	9	17
3	8	18	31
9	17	36	51

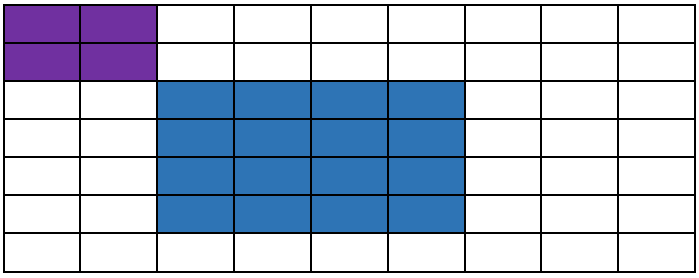
Now, in this new array, each entry represents the sum of the rectangle defined by the top left corner and that square. For example, the 18 in row 2, column 3 of the new array, highlighted in green represents the sum of the rectangle shown in yellow in the original array.

Consider the following picture. Let's say we want the sum of the values in blue in this original array. (Note: I haven't put in actual values since the key idea doesn't require actual values to be in the boxes.)

Once we have an auxiliary cumulative frequency array stored in the manner above, we can then calculate the sum of any contiguous rectangle of values in the input array.

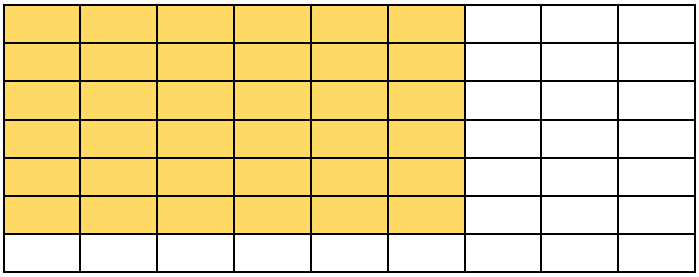
A visual example follows on the next page.

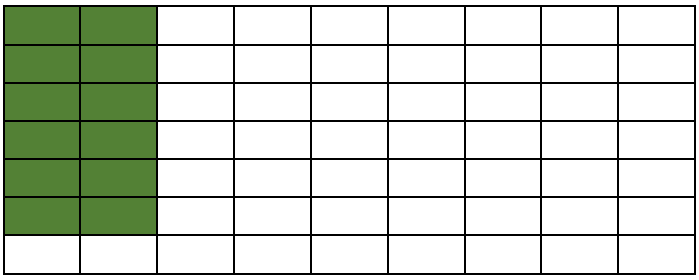
Original Array:

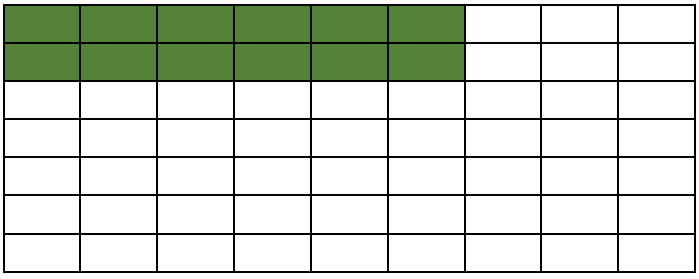


We can obtain the sum of the blue values as follows: Imagine getting the sum of the items in yellow (a single look up in the cumulative frequency array), subtracting out the sum of the items in green (two look ups in the cumulative frequency array), and then adding back the sum of the items in purple, shown above.

Total Sum (in Yellow) –







The green subtracts out everything from the yellow we don't want, but subtracts out the purple twice, so we want to add this back in so it gets subtracted out properly. You have to be careful with border conditions (just like the special case mentioned in the one dimensional example), but basically if we have a cumulative frequency array, then we can calculate the sum of the box from (lowx, lowy) to (highx, highy) inclusive, as follows:

```
cumfreq[highx][highy] - cumfreq[lowx-1][highy]
                      - cumfreq[highx][lowy-1]
                      + cumfreq[lowx-1][lowy-1]
```

*** Note for Jan 2022 Lecture:

We will only cover the preceding pages for the lecture on Binary Index Trees.

Problem #1: Maximal Contiguous Subsequent Sum Problem

Maximum Contiguous Subsequence Sum: given (a possibly negative) integers $A_1, A_2, \ldots, A_n$, find (and identify the sequence corresponding to) the maximum value of

$$\sum_{k=i}^{j} A_k$$

For the degenerate case when all of the integers are negative, the maximum contiguous subsequence sum is zero.

Examples:

If input is: {-2, 11, -4, 13, -5, 2}. Then the output is: 20.

If the input is {1, -3, 4, -2, -1, 6}. Then the output is 7.

In the degenerative case, since the sum is defined as zero, the subsequence is an empty string. An empty subsequence is contiguous and clearly, 0 > any negative number, so zero is the maximum contiguous subseqeunce sum.

The brute force solution tries all possible subsequences (there are $\frac{n(n-1)}{2}$ of these), and adds up the numbers in each of these sequences. Many of these sequences have *O(n)* values in them, so if we add up all the numbers in each contiguous subsequence, our algorithm would take $O(n^3)$ time. But, if we store a cumulative frequency array first, before processing the data, we can reduce the run time of this brute force solution to $O(n^2)$ time.

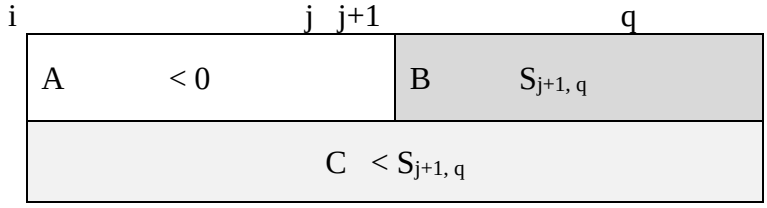
Alternatively, we can also achieve an $O(n^2)$ by realizing that if we have calculated the sum of the contiguous subsequence a[i..j] it only takes us one more step to calculate the sum of the contiguous subsequence a[i..j+1]. Here is the code corresponding to using this observation:

```
public static int MCSS(int [] a) {
     int max = 0, sum = 0, start = 0, end = 0;
     // Try all possible values of start and end indexes for the sum.
     for (i = 0; i < a.length; i++) {
          sum = 0;
          for (j = i; j < a.length; j++) {
               sum += a[j]; // No need to re-add all values.
               if (sum > max) {
                   max = sum;
                   start = i; // Although method doesn't return these
                   end = j;  // they can be computed.
               }
          }
     }
     return max;
}
```

MCSS Problem: O(n) Algorithm

To further streamline this algorithm from a quadratic one to a linear one will require the removal of yet another loop. Getting rid of another loop will not be as simple as was the first loop removal. The problem with the quadratic algorithm is that it is still an exhaustive search, we've simply reduced the cost of computing the last subsequence down to a constant time (O(1)) compared with the linear time (O(N)) for this calculation in the cubic algorithm. The only way to obtain a subquadratic bound for this algorithm is to narrow the search space by eliminating from consideration a large number of subsequences that cannot possibly affect the maximum value.

How to eliminate subsequences from consideration



If A < 0 then C < B

If $\sum_{k=i}^{j} A_k < 0$, and if q > j, then $A_i \ldots A_q$ is not the MCSS!

Basically if you take the sum from A_i to A_q and get rid of the first terms from A_i to A_j your sum increases!!! Thus, in this situation the sum from A_{j+1} to A_q must be greater than the sum from A_i to A_q. So, no subsequence that starts from index i and ends after index j has to be considered. So – if we test for sum < 0 and it is – then we can break out of the inner loop. However, this is not sufficient for reducing the running time below quadratic!

Now, using the fact above and one more observation, we can create a O(n) algorithm to solve the problem.

If we start computing sums $\sum_{k=i}^{i} A_k$, $\sum_{k=i}^{i+1} A_k$, etc. until we find the first value j such that

$\sum_{k=i}^{j} A_k < 0$, then immediately we know that either

1) The MCSS is contained entirely in between A_i to A_{j-1} OR
2) The MCSS starts before A_i or after A_j.

From this, we can also deduce that unless there exists a subsequence that starts at the beginning that is negative, the MCSS MUST start at the beginning. If it does not start at the beginning, then it MUST start after the point at which the sum from the beginning to a certain point is negative.

Using this idea, we can solve the problem by automatically setting our sum to 0 and our potential new starting point to the position right after the running sum fell below 0.

Here is the code:

```
public static int MCSS(int [] a) {

     int max = 0, sum = 0, start = 0, end = 0, i=0;

     // Cycle through all possible end indexes.
     for (j = 0; j < a.length; j++) {

          sum += a[j]; // No need to re-add all values.
          if (sum > max) {
              max = sum;
              start = i; // Although method doesn't return these
              end = j;  // they can be computed.
          }
          else if (sum < 0) {
               i = j+1; // Only possible MCSSs start with an index >j.
               sum = 0; // Reset running sum.
          }
     }
     return max;
}
```

MCSS Linear Algorithm Clarification

Whenever a subsequence is encountered which has a negative sum – the next subsequence to examine can begin after the end of the subsequence which produced the negative sum. In other words, there is no starting point in that subsequence which will generate a positive sum and thus, they can all be ignored. To illustrate this, consider the example with the values

5, 7, -3, 1, -11, 8, 12

You'll notice that the sums 5, 5+7, 5+7+(-3) and 5+7+(-3)+1 are positive, but 5+7+(3)+1+(-11) is negative.

It must be the case that all subsequences that start with a value in between the 5 and -11 and end with the -11 have a negative sum. Consider the following sums:

7+(-3)+1+(-11) (-3)+1+(-11) 1+(-11) (-11)

Notice that if any of these were positive, then the subsequence starting at 5 and ending at -11 would have to be also. (Because all we have done is stripped the initial positive subsequence starting at 5 in the subsequences above.) Since ALL of these are negative, it follows that NOW MCSS could start at any value in between 5 and -11 that has not been computed.

Thus, it is perfectly fine, at this stage, to only consider sequences starting at 8 to compare to the previous maximum sequence of 5, 7, -3, and 1.

Note: No problem in the Problems section is included, since this same code is a portion of the solution to maxsum.java.

Problem #2: MaxSum (Two Dimensional Maximal Contiguous Subsequent Sum)

Now, consider solving the MCSS problem in a two dimensional array of numbers. The pure brute force solution runs in $O(n^6)$ time on an input array of size n x n. (There are $O(n^4)$ possible bounding rectangles and if we add each item in each rectangle from scratch it would take up to $O(n^2)$ to calculate each rectangle sum.)

Here is how we can use the ideas from both cumulative frequencies and the one dimensional MCSS problem to solve the problem in $O(n^3)$ time.

For just a moment, assume that our problem was simplified to find the best two dimensional MCSS rectangle knowing that the starting row and ending row is fixed:

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

Thus, in the picture above, we simply have to find the best rectangle that starts on row 2 (using 0-based indexing) and ends on row 4, inclusive. Thus, we only need to consider all possible combinations of starting and ending columns.

If we can, imagine "scrunching" each sub-column in yellow into one cell, where the value in that one cell was simply the sum of the values in the original column. Thus, our "super-row" with all the scrunched values looks like:

14	-2	-9	-2	2	-6	0	1	-1

To solve this simplified problem, we only need to run the $O(n)$ one dimensional MCSS solution on this "super-row", where n is the number of values in the scrunched row. Since there are only $O(n^2)$ possible settings of starting and ending row, the basic idea would be:

Loop through each possible combination of starting and ending row.

a) Calculate the scrunched values for the row.
b) Run the one dimensional MCSS algorithm on these scrunched values.
c) Update the best answer as necessary.

The only detail missing in this description is how we will calculate the values of the scrunched rows. Notice that if we don't calculate each value in $O(1)$ time for a total of $O(n)$ for the whole row, then our run-time is slower than the desired run-time. Luckily, each of the single values in the scrunched row above represent a contiguous subsequence sum in a column. This is where the cumulative frequency arrays come in handy. Simply precompute a cumulative frequency array for each row in advance. Then, when you need to find the sum of several values in a column, look up the appropriate difference of values in the cumulative frequency array in $O(1)$ time.

Problem #3: SqFree

The problem is as follows: Given two integers, low and high, with low ≤ high, determine the number of integers that aren't perfect squares in that range. There are up to 10000 queries and low and high are two positive integers in between 1 and 1000000, inclusive.

There are many ways to solve this problem within the given limits, but since this lecture is about cumulative frequency arrays, we'll show a solution that utilizes a cumulative frequency array.

First, create an array of size 1000001, storing a 1 in an index if that index is not a perfect square, and 0 if it is. The easiest way to do this is seed our array with 1s, and the write zeros in each index of the form i^2, for each i, $0 \le i \le 1000$.

Then, take this array and convert it to a cumulative frequency array so cumfreq[i] stores the number of non-squares less than or equal to i.

Here is the original array and the cumulative frequency array up to index 12:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
orig	0	0	1	1	0	1	1	1	1	0	1	1	1
cumfreq	0	0	1	2	2	3	4	5	6	6	7	8	9

Now, if we want to know the number of square free numbers in between low and high, inclusive, our answer is simply cumfreq[high] - cumfreq[low-1]. For example, the number of square free numbers in between 4 and 10, inclusive is cumfreq[10] - cumfreq[3] = 7 - 2 = 5. These five values are 5, 6, 7, 8 and 10.

Problem #4: MaxSquare

The problem is as follows: Given a rectangular grid of 0s and 1s, determine the size of the largest contiguous sub-square containing all 1s. The largest the rectangle can be is 1000 x 1000.

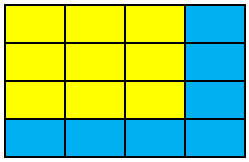
Notice that this problem is quite similar to the max sum problem in many ways. If we can calculate the sum of a sub-square of size $m \times m$ in O(1) time, we can simply check to see if this sum is m^2.

One idea to solve this problem is as follows:

Try "growing" the square from every possible top left corner.

Note that there are up to 1000000 possible corners. This seems problematic, since there will be quite a few squares we'll have to consider from every top left corner, but here's how we can make our algorithm more efficient:

From a given square, try adding a row and column iteratively:



For example, note that if the current square in yellow isn't all 1s, then we can immediately stop and not check the square in yellow and blue. If it IS all 1s, we can just add the items in blue in O(1) time using our cumulative frequency look up tables for rows and columns (or recalculate the whole new square in O(1) time).

Thus, each square calculation takes O(1) time. The only problem of course, is that if we grow one square 500 times and if we grow another square 600 times. BUT…remember if we grow one square 500 times, the next time we start from a new top left corner, there's no point in starting with a 1 x 1 square for our first calculation. We may as well start with the 500 x 500 square, thus we can't grow any following square 600 times, since we're already starting at 500.

Thus, as we maintain our best answer, it is increasing. Thus, if we are trying to calculate overall run time, we see that we have O(1) time for each starting square for the initial square calculation, followed by at most O(*min(r, c)*) occurrences ***total*** where any square grows. (The growth from size 1 x 1 to 2 x 2 can happen at most once, and in general the growth from size k x k to $(k+1)$ x $(k+1)$ can happen at most once, for any k.)

It follows that the run time of this solution os O(rc), where r and c are the number of rows and columns in the input grid.