Cumulative Frequency Array: For Range Sum Queries

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 upto 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	5
1	2	6	5
6	3	9	2

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

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

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

2	5	9	14
3	8	18	28
9	17	36	48

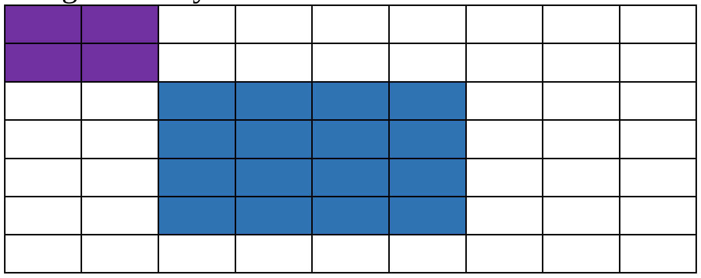
Now, in this new array, each entry represents the sum of the rectangle defined by the top left corner and that square. For example, the 18 in row 2, column 3 of the new array, highlighted in green represents the sum of the rectangle shown in yellow in the original array.

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.

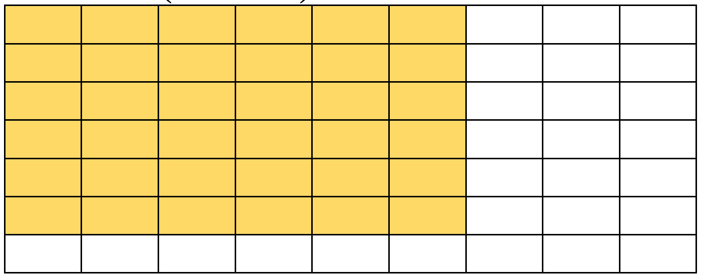
A visual example follows on the next page.

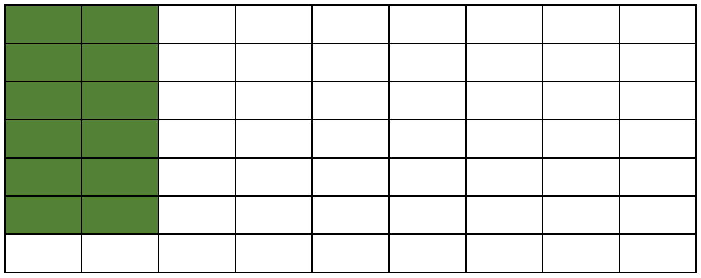
Original Array:

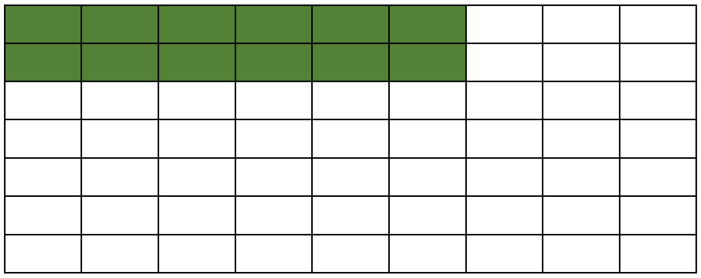


We can obtain the sum of the blue values as follows: Imagine getting the sum of the items in yellow (a single look up in the cumulative frequency array), subtracting out the sum of the items in green (two look ups in the cumulative frequency array), and then adding back the sum of the items in purple,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

Example of a Sweep: Sorted List Matching Problem

Given two sorted lists of names, output the names common to both lists.

Perhaps the standard way to attack this problem is the following:

For each name on list #1, do the following:

a) Search for the current name in list #2.

b) If the name is found, output it.

If a list is unsorted, steps a and b may take O(n) time. Can you tell me why?

BUT, we know that both lists are already sorted. Thus we can use a binary search in step a. From CS1, we learned that this takes O(log n) time, where n is the total number of names in the list. For the moment, if we assume that both lists are of equal size, then we can safely say that the size of list #2 is about ½ the total input size, so technically, our search would take O(log n/2) time, where n is the TOTAL SIZE of our input to the problem. Using our log rules however, we find that $\log_2 n = (\log_2 n/2) + 1$. Thus, it's fairly safe to assume for large n that our running time is simply $O(\log_2 n)$.

Now, that is simply the running time for 1 loop iteration. But how many loop iterations are there? (Assume that there are n/2 names on each list, again, where n is the TOTAL SIZE of the input.) Under our assumption, there will be n/2 loop iterations, so our total running time would be $O(n \log_2 n)$. Why did I not divide the expression in the Big-O by 2?

A natural question becomes: Can we do better? The answer is yes. What is one piece of information we have that our first algorithm does NOT assume?

That list #1 is sorted. You'll notice that our previous algorithm will work regardless of the order of the names in list #1. But, we KNOW that this list is sorted also. Can we exploit this fact so that we don't have to do a full binary search for each name?

Consider how you'd probably do this task in real life...

List #1	List #2
Adams	Boston
Bell	Davis
Davis	Duncan
Harding	Francis
Jenkins	Gamble
Lincoln	Harding
Simpson	Mason
Zoeller	Simpson

You'd read that Adams and Boston are the first names on the list. Immediately you'd know that Adams wasn't a match, and neither would any name on the list #1 alphabetically before Boston. So, you'd read Bell and go on to Davis. At this point you'd deduce that Boston wasn't on the list either, so you'd read the next name on list #2 – voila!!! A match! You'd output this name and simply repeat the same idea. In particular, what we see here is that you ONLY go forward on your list of names. And for every "step" so to speak, you will read a new name off one of the two lists. Here is a more formalized version of the algorithm:

```
1) Start two "markers", one for each list, at the beginning of both lists.
2) Repeat the following steps until one marker has reached the end of its list.
   a) Compare the two names that the markers are pointing at.
   b) If they are equal, output the name and advance BOTH markers one spot.
      If they are NOT equal, simply advance the marker pointing to the name that comes earlier
         alphabetically one spot.
```

<u>Algorithm Run-Time Analysis</u>
For each loop iteration, we advance at least one marker.
The maximum number of iterations then, would be the total number of names on both lists, which is n, using our previous interpretation.

For each iteration, we are doing a constant amount of work. (Essentially a comparison, and/or outputting a name.)

Thus, our algorithm runs in O(n) time – an improvement over our previous algorithm.

A final question one must ask is, can we solve this question in even less time? If yes, what is such an algorithm, if no, how can we prove it?

Our proof goes along these lines: In order to have an accurate list, we must read every name on one of the two lists. If we skip names on BOTH lists, we can NOT deduce whether we would have matches between those names or not. In order to simply "read" all the names on one list, we would take O(n/2) time. But, in order notation, this is still O(n), the running time of our second algorithm. Thus, we know we can not do better in terms of time, (within a constant factor), of our second algorithm.

Running Sums: 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, <u>11, -4, 13</u>, -5, 2}. Then the output is: 20.

If the input is {1, -3, <u>4, -2, -1, 6</u>}. 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 O(N^3) Algorithm (brute force method)

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

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

      // Cycle through all possible values of start and end indexes
      // for the sum.
      for (i = 0; i < a.length; i++) {
            for (j = i; j < a.length; j++) {
                  sum = 0;

                  // Find sum A[i] to A[j].
                  for (k = i; k <= j; k++)
                        sum += a[k];
                  if (sum > max) {
                       max = sum;
                       start = i; // Although method doesn't return these
                       end = j;  // they can be computed.
                  }
            }
      }
      return max;
}
```

<u>General Observation Analysis</u>

Look at the three loops: the *i* loop executes *SIZE* (or N) times. The *j* loop executes *SIZE-1* (or N-1) times. The *k* loop executes *SIZE-1* times in the worst case (when i = 0). This gives a rough estimate that the algorithm is O(N^3).

Precise Analysis Using Big-Oh Notation

In all cases the number of times that, sum += a[k], is executed is equal to the number of ordered triplets (i, j, k) where $1 \le i \le k \le j \le N^2$ (since *i* runs over the whole index, *j* runs from *i* to the end, and *k* runs from *i* to *j*). Therefore, since *i*, *j*, *k*, can each only assume 1 of n values, we know that the number of triplets must be less than $n(n)(n) = N^3$ but $i \le k \le j$ restricts this even further. By combinatorics it can be proven that the number of ordered triplets is n(n+1)(n+2)/6. Therefore, the algorithm is $O(N^3)$.

The $O(N^2)$ Algorithm

Algorithm

```
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;
}
```

Discussion of the technique and analysis

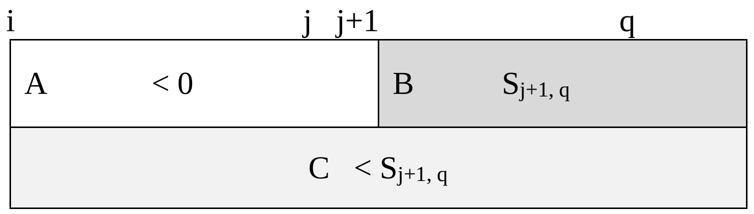
We would like to improve this algorithm to run in time better than $O(N^3)$. To do this we need to remove a loop! The question then becomes, "how do we remove one of the loops?" In general, by looking for uncessary calculations, in this specific case, uncessary calculations are performed in the innerloop. The sum for the subsequence extending from *i* to *j – 1* was just calculated – so calculating the sum of the sequence from *i* to *j* shouldn't take long because all that is required is that you add one more term to the previous sum (i.e., add A_j). However, the cubic algorithm throws away all of this previous information and must recompute the entire sequence! Mathematically, we are utilizing: $\sum_{k=i}^{j} A_k = (\sum_{k=i}^{j-1} A_k) + A_j$.

The O(N) Algorithm (A linear algorithm)

Discussion of the technique and analysis

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.

So, using this how can we come up with an algorithm?

1) We can compute intermediate sums starting at i=0.
2) When a new value is added, adjust the MCSS accordingly.
3) If the running sum ever drops below 0, we KNOW that if there is a new MCSS than what has already been calculated, it will start AFTER index j, where j is the first time the sum dropped below zero.
4) So now, just start the new running sum from j+1.

Algorithm

```
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.