Some Math (Number Theory)

I. GCD and LCM

The greatest common divisor of two positive integers is the largest number that divides evenly into both. The least common multiple of two positive integers is the smallest number such that both of the integers divide evenly into it.

If we think about finding the gcd of two values, a and b, imagine laying out two sticks of lengths a and b, one below the other, lined up on the left end:

```
______________________________________________________ (a)
_______________ (b)
```

For a number to divide into both and and b evenly, we'd have to "chop" both sticks into equal segments of length d. If b is fairly short, we can repeat b a few times in the drawing:

```
______________________________________________________ (a)
______________|______________|______________|__________ (3b + r)
      b              b               b           r
```

Our cut lines will go every d units and they'll cut exactly at each of the boundaries of the three b sticks. Since the last "cut" would have to end at a, then it must also end at the segment labeled r. So, if d divides into a, and d divides into b, then what the drawing above infers is that d must also divide into r. Essentially, this gives us the equation:

gcd(a, b) = gcd(b, a%b), so long as a is not divisible by b. (Namely, that leftover piece isn't length 0.)

If the leftover piece is length 0, then smaller number is the gcd. With this in mind, we can code a GCD method

Here is a recursive solution (Euclid's Algorithm) to determine the gcd of two values:

```java
public static long gcd(long a, long b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

The least common multiple of two integers is smallest number that both numbers divide into evenly. We won't provide the proof here, but the least common multiple of two integers a and b is always their product divided by their gcd. Thus, here is the lcm code:

```java
public static long lcm(long a, long b) {
    return a/gcd(a,b)*b;
}
```

In many subproblems that arise in competitive programming, it’s useful to calculate the gcd or lcm of two integers (or more). (We can do more by calling the function that takes 2 values in multiple times. In essence, gcd(a,b,c) = gcd(gcd(a, b), c). Same for lcm.

II. Prime Testing, Prime Sieve, Prime Factorization

A prime number is one that is only divisible by one and itself. If we are testing a single number of primality, we do trial division until the square root of the number. To see this, note that if n = ab, where a > 1 and b > 1, at least one of the two is less than or equal to the square root. If both were greater, than the product of ab and would greater than $\sqrt{n}\sqrt{n} = n$, but it would be impossible for n to be greater than n. Thus, it follows if n has a non-trivial divisor, then it must have at least one non-trivial divisor less than or equal to the square root of n. Here is a function that performs this task:

```
public static boolean isPrime(int n) {

        // We can stop at the square root...we avoid ints.
        for (int i=2; i*i<=n; i++)

                // Not prime...
                if (n%i == 0)
                        return false;

        // We are good if we get here.
        return true;
}
```

If we want to generate a list of all primes from 2 to n, we can use the Sieve of Eratosthenes (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). It works as follows:

1) Write down all the numbers from 2 to n.
2) Go through each number, in order.
3) For each of these, if it’s not crossed off, circle it.
4) Then, cross off each multiple of that number. Thus, when we circle 2 at the beginning of the algorithm, then cross off 4, 6, 8, 10, and so forth, until we get to the last even number less than or equal to n.

The numbers not crossed off at the end of these (the circled ones) are all the primes in the range.

As previously mentioned, we can technically stop our outer loop when we get to the square root of the number we are checking because any non-prime has at least one non-trivial divisor less than or equal to its square root.

If a number is already crossed off, there is no need to cross off its multiples. For example, when we get to 6, all of its multiples were crossed off when we circled 2, so there's no need to cross them off again. Can you think about what to edit in the code above to skip these unnecessary loop iterations?

After implementing the prime sieve, we are left with a boolean array such that isP[i] is set to true if and only if i is prime. For some questions, this formation of the data is good enough to solve problems. For other problems, it's necessary to have a list of integers (or an array) with the prime numbers in successive order.

Here is a function that implements a basic prime sieve for all primes upto n and returns an ArrayList which stores the primes in order.

```
public static ArrayList<Integer> pList(int n) {

        // Assume everything is prime at first.
        boolean[] isP = new boolean[n+1];
        Arrays.fill(isP, true);

        // Go through each number.
        for (int i=2; i*i<=n; i++)

                // Cross off multiples of i. (2i, 3i, etc.)
                for (int j=2*i; j<=n; j+=i)
                        isP[j] = false;

        // Copy over all primes into list.
        ArrayList<Integer> list = new ArrayList<Integer>();
        for (int i=2; i<=n; i++)
                if (isP[i])
                        list.add(i);

        return list;
}
```

Once we can check for primality, we can also calculate the prime factorization of an integer by repeatedly dividing out prime factors until the number left is prime. We'll leave this as an exercise for you to do.

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.

Here is some Java code that takes in an array and returns the cumulative frequency version of the array without changing the original:

```
public static int[] makeCF(int[] array) {

        // Store here.
        int[] cf = new int[array.length+1];

        // Just add this value to the previous total.
        for (int i=1; i<cf.length; i++)
                cf[i] = cf[i-1] + array[i-1];

        // Ta da!
        return cf;

}
```

Also, if we just wanted to overwrite the original array (with a slight different interpretation where index 0 of the cumulative frequency array stores the sum upto index 0 in the original array, then we could do):

```
for (int i=1; i<array.length; i++)
    array[i] += array[i-1];
```

So, if the before picture of array was:

3	2	8	6	5	1

The after picture would be:

3	5	13	19	24	25

The method, given the first array would produce this one:

0	3	5	13	19	24	25

Maximum Contiguous Subsequence Sum

Maximum Contiguous Subsequence Sum: given (a possibly negative) integers A_1, A_2, …, 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, $\underline{11, -4, 13}$, -5, 2}. Then the output is: 20.

If the input is {1, -3, $\underline{4, -2, -1, 6}$}. 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.

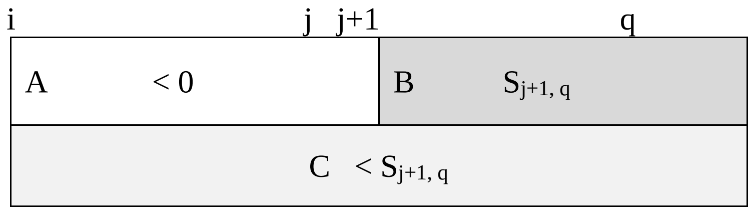
Alternatively, we can also achieve an $O(n^2)$ by realizing that if we have calculated the sum of the contiguous subsequence a[i..j] it only takes us one more step to calculate the sum of the contiguous subsequence a[i..j+1]. Here is the corresponding code:

```
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;
}
```

<u>MCSS Linear Algorithm Clarification</u>
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.