Consider the question: what would have to happen if two different values, $0 \leq x_1 < x_2 < n$, with $f(x_1) = f(x_2)$? Set these expressions equal to each other and we get:

$$ax_2 + b \equiv ax_1 + b \pmod{n}$$
$$a(x_2 - x_1) \equiv 0 \pmod{n}$$

Thus, in order for an ordered pair $(a, b)$ to be an invalid key, it must be the case that $n \mid a(x_2 - x_1)$, where $0 < x_2 - x_1 < n$. Notice that if $\gcd(a, n) = 1$, then this can not happen.

Alternatively, if $\gcd(a, n) > 1$, then we can set $x_2 - x_1 = \frac{n}{\gcd(a,n)}$ and get a solutions for $x_1$ and $x_2$ where $0 \leq x_1 < x_2 < n$, with $f(x_1) = f(x_2)$. To prove that a specific solution exists, a valid solution is simply $x_1 = 0$ and $x_2 = \frac{n}{\gcd(a,n)}$.

Thus, we can conclude that $a$ can only take on values where $\gcd(a, n)$. Specifically, $a$ can be one of $\varphi(n)$ values. We can choose $b$ independently and there are no restrictions on $b$, thus, $b$ can be one of $n$ values, ranging from $0$ to $n-1$, inclusive. Thus, the total number of valid keys is $n\varphi(n)$. Since the input value of $n$ ranges up to $10^9$ and it's possible that $\varphi(n) \sim n$, while both $n$ and $\varphi(n)$ can be stored in ints, their product can be close to $10^{18}$ and must be calculated using longs (so cast one item to long or just store both as longs from the get go.) Also, we must use a reasonably efficient algorithm to calculate $\varphi(n)$ which runs in $O(\sqrt{n})$ time. A program would get a time limit exceeded if it did a for loop from 1 to $n$ checking the $\gcd$ of each value with $n$. Rather, the prime factorization formula given in class should be used. The prime factorization and the corresponding calculation of $\varphi$ can be executed in $O(\sqrt{n})$ time by stopping looking for unique prime divisors when reaching the square root of $n$ and taking care to not forget the last prime factor if it wasn't yet discovered.

**Fact**
As covered in class, the number of times a prime $p$ divides into $n!$ can be formally represented as

$$\left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \cdots$$

In code, this is just a while loop where we divide $n$ by $p$, add the result to an accumulator and repeat until $n$ goes down to 0. This runs pretty quickly ($O(\lg n)$ time). So, we can run a prime sieve to generate all primes up to $n$, and then for each of the primes, compute the sum shown above.
Fermat
At first one might think they have to find the two perfect squares that add up to a prime, but the introduction explains that ALL primes of the form 4c+1 CAN BE expressed as the sum of two squares and NO primes of the form 4c+3 CAN BE expressed as the sum of two primes. Thus, one key goal of the question is to identify numbers that are prime AND equivalent to 1 mod 4.

Run a regular prime sieve where sieve[i] = 1 if i is prime and sieve[i] = 0 if i is not prime.

Then create a second array sieve2, which is a copy of sieve except that you change sieve2[i] = 0 for all i ≡ 3 (mod 4). (This can be done by starting a loop at 3 and setting every 4th value to 0.)

Finally, to be able to handle the given queries, create a cumulative frequency array out of both sieve arrays. That way, each pair of queries can be answered in O(1) time.

Ground Game
This problem is the banger in the set. After reading in the string, process it character by character, keeping track of the current depth. Moving right or left makes no change to the current depth, moving down adds 1 to it and moving up subtracts 1 from it. After each update, see if the new depth is bigger than the currently stored max depth. If so, update the max depth.

Perfection in Numbers
The first issue many people had with this problem is that they didn't read the bounds. The input value can be as large as 10^{12}. Thus, the input value can't fit in an int and trying to read it in as an int in Java (nextInt()) causes a run time error.

Once you realize you have to read in the value as a long, the next hurdle is that the standard brute force solution of checking each divisor up to half of n takes too long, since we can't do 5 \times 10^{11} simple operations. But the key is realizing that divisors of numbers come in pairs, except for the square root in the case of perfect squares. Thus, we need to only check for divisors up to the square root of the input value and this loop runs at most 10^6 times. When we find a divisor, there are three cases: (a) The divisor is 1, just add 1 to the running sum since it's corresponding divisor n shouldn't be counted, (b) The divisor, k is less than the square root of n, in which case we want to add both k and n/k to our running sum, or (c) The divisor k is such that k \times k = n. In this case we only want to add k once, not twice.

We can avoid case (a) by starting our sum at (1) and starting to look for divisors at 2. The check between cases (b) and (c) is just a single if statement in the loop.

Profits
This is exactly the MCSS problem taught in class. In this problem however, they specify that the contiguous subsequence must be non-empty. Thus, instead of setting the initial result to 0, it must be set to the first value in the list, just in case all values in the list are negative. Also, since there are up to 250,000 values in the list, the O(n) algorithm must be used.