Shenendehowa Code Sprint
Official Solutions

May 2015

1 Patterns

Note that for any string of length n, the values $m = 1, n+1, 2n+1, 3n+1, ...$ all correspond to the same character in the string. Similarly, $m = 2, n+2, 2n+2, 3n+2, ...$ all correspond to the same character. So, we can simplify m by taking it modulo n (using the operator `%`). After the simplification, we just get the character at the corresponding position in the string with `charAt`. We subtract one from m before simplifying because Java strings are zero-indexed.

```
import java.util.*;

public class P1 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s = in.nextLine();
        long n = in.nextLong();
        n = (n - 1) % s.length();
        System.out.println(s.charAt((int)n));
    }
}
```

2 Cow Fencing

The fence must start one unit to the left of the leftmost cow and end one unit right of the rightmost cow. Making it any smaller would mean cows are either on the fence or outside the fence, neither of which are allowed. By the same argument, the fence must span from one unit below the bottommost cow to one unit above the topmost cow. So, the width of the rectangle enclosed is $(x_h+1)-(x_l-1) = x_h-x_l+2$, where x_h is the highest cow x-coordinate and x_l is the lowest cow x-coordinate. The height of the rectangle is $y_h - y_l + 2$ by the same math. We can easily get x_h, x_l, y_h, and y_l, which leads to rectangle dimensions and area.

```
import java.util.*;

public class P2 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt(), BIL = (int)1e9;
        long yl = BIL, yh = -BIL, xl = BIL, xh = -BIL;
        for (int i = 0; i < n; i++) {
            int a = in.nextInt(), b = in.nextInt();
            xl = Math.min(xl, a);
            xh = Math.max(xh, a);

            yl = Math.min(yl, b);
            yh = Math.max(yh, b);
        }
        System.out.println((xh - xl + 2) * (yh - yl + 2));
    }
}
```

3 Making Burritos

We first prove that this problem can be solved with a greedy algorithm—always serve the earliest orders first. Say each burrito takes k time to make and we have two orders, one coming in at time a and one at time b. Without loss of generality, let $a < b$ (if $a = b$, it clearly doesn't matter who we serve first). If we serve b first, we will finish both orders by $b + 2k$. If we serve the order a first, we will finish the first order at time $a + k$. If $a + k \leq b$, we will then finish both orders by time $b + k$. Otherwise, we will finish by time $a + 2k$. Both of these are less than $b + 2k$. So, we should always make a burrito for an earlier order before a later one.

Now we just have to simulate making the burritos in this order to get our answer. We first sort the orders by the time they came in. We create a variable p which stores when the previous burrito was finished being made. Now, we iterate through our sorted list of orders. For an order that comes in at t_i, we calculate when we can start the order. If the previous burrito was finished before t_i (so if $p < t_i$), then we start the current order at t_i. Otherwise, we start the current order as soon as we finish the previous order. This can be expressed as

$$\text{Start Time} = \max(p, t_i)$$

So we finish the current order at $\max(p, t_i) + k$; therefore, $p_{\text{next}} = \max(p_{\text{prev}}, t_i) + k$. Calculating p for the last order gives us our answer.

```
import java.util.*;

public class P5 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int btime = in.nextInt(), numb = in.nextInt();
        int[] otimes = new int[numb];

        for (int i = 0; i < numb; i++) otimes[i] = in.nextInt();
        Arrays.sort(otimes);
        int res = 0;
        for (int i = 0; i < numb; i++)
            res = Math.max(res, otimes[i]) + btime;
        System.out.println(res);
    }
}
```

4 Numbering the FIRST Game Manual

In order to get full credit on this problem, we must find a way to consider the numbers in bulk; processing them one-by-one and adding up how many digits each has is too slow. We can do this by considering all numbers with the same number of digits at once. Numbers in the interval [1, 10) have one digit. Numbers in the interval [10, 100) have two digits. In general, all numbers in the range $[10^k, 10^{k+1})$ have $k+1$ digits. Since n is up to 10^{15}, we only have at most 15 intervals to consider.

The interval of numbers with k digits is $[10^{k-1}, 10^k - 1]$. However, if n is below $10^k - 1$, it will end at n. So, our interval is $[10^{k-1}, \min(10^k - 1, n)]$. The size of this interval is therefore $\min(10^k - 1, n) - 10^{k-1} + 1 = \min(10^k, n+1) - 10^k$. The number of digits in that interval is $k \times (\min(10^k, n+1) - 10^k)$. We add this up for all k and we're done.

```
import java.util.*;

public class P3 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        long n = in.nextLong();
        long res = 0;
        for (long i = 1, j = 1; j <= n; i++, j *= 10)
            res += i * (Math.min(n + 1, 10 * j) - j);
        System.out.println(res);
    }
}
```

5 Goldbach's Conjecture

One sufficiently fast solution to this problem simply goes through every possible combination of positive integers a, b such that $a + b = x$. For each combination, we check if both a and b are prime. If they are, and improve our current answer, we update our answer. We test for primality of some integer n by first considering special edge cases (numbers below 4). Then, we immediately return false for any even numbers. Then, we loop through all odd numbers up to the floor of the square root of n. A composite number is guaranteed to have a factor in this range because a factor $f > \sqrt{n}$ must have some f_2 such that $f_2 f = n$. So, $f_2 = \frac{n}{f} < \frac{n}{\sqrt{n}} = \sqrt{n}$. If any of the odd numbers we loop through leave no remainder when divided into n, we return false.

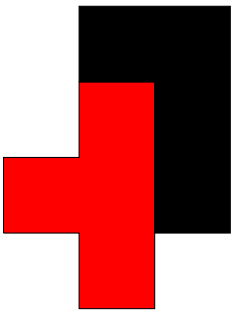
Testing x for primality using our method involves roughly $\sqrt{x}$ operations because of the `for` loop up to $\sqrt{x}$. So, each test requires at most $\sqrt{10^5} \approx 300$ operations. We need to do 10^5 prime tests, so we are doing about 3×10^7 operations. We also need to actually go through the combinations of a, b mentioned previously, but this is fast (roughly 10^5 operations). We're doing well below 10^8 operations, and they're all fairly simple—basically within a one-line `for` loop—so we can expect the solution to run in time.

```
import java.util.*;

public class P6 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        boolean[] isp = new boolean[n + 10];
        for (int i = 0; i <= n; i++) isp[i] = isPrime(i);

        int p = 0;
        for (int i = 0; i < n; i++)
            if (isp[i] && isp[n - i] && Math.min(i, n - i) > p)
                p = Math.min(i, n - i);
        System.out.println(p + " " + (n - p));
    }

    public static boolean isPrime(int n) {
        if (n < 2) return false;
        if (n == 2 || n == 3) return true;
        if (n % 2 == 0) return false;
        for (int i = 3; i * i <= n; i += 2)
            if (n % i == 0) return false;
        return true;
    }
}
```

6 Mega-Tetris

Although we're given full information on what each of the two pieces looks like, we only need to know the bottommost and topmost block in each column. The pieces in the middle don't matter. At some point, the top block must come in contact with the block below, or the bottom of the Tetris grid. For each column, we assume that column is the contact point, and calculate the height of the resulting figure if so. Taking the max of all these heights will give us the actual height of the resulting figure.

To calculate the height assuming a certain column is the contact point, we split into three cases: only the bottom piece is present in the column, only the top piece is present in the column, and both the top and bottom pieces are in the column. It's also possible that the column is completely empty, but this won't affect our answer so we don't consider it. The code dealing with each case is rather self-explanatory. In the sample input for the problem, every column results in the third case. The input for the block arrangement below has a case 1, a case 3, and a case 2, in that order.

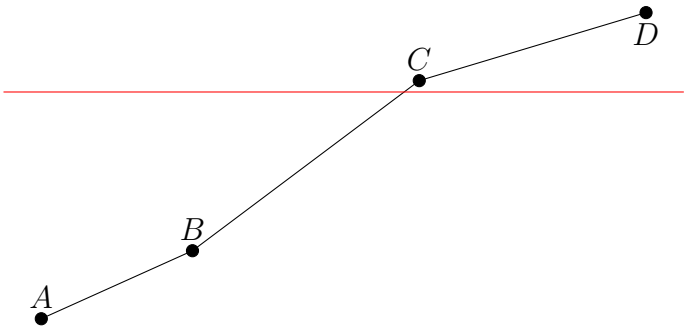


```
import java.util.*;

public class P7 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        in.nextLine();

        boolean[][] col_oc = new boolean[n][2];
        int[][] minh = new int[n][2], maxh = new int[n][2];
        for (int i = 0; i < n; i++) minh[i][0] = minh[i][1] = 1000;
        int topmax = 0, botmin = 1000; // top of top piece, bottom of bottom piece

        for (int i = 1; i >= 0; i--) // 1=top; 0=bottom piece
            for (int y = 0; y < n; y++) {
                String ln = in.nextLine();
                for (int x = 0; x < n; x++)
                    if (ln.charAt(x) == '1') {
                        col_oc[x][i] = true;
                        minh[x][i] = Math.min(minh[x][i], n - y);
                        maxh[x][i] = Math.max(maxh[x][i], n - y);
                        if (i == 1) topmax = Math.max(topmax, n - y);
                        if (i == 0) botmin = Math.min(botmin, n - y);
                    }
            }
```

```
        int res = 0; // where would the top block of the top piece be?
        for (int col = 0; col < n; col++) {
            if (col_oc[col][0] && !col_oc[col][1]) // Case 1
                res = Math.max(res, maxh[col][0] - botmin + 1);
            if (!col_oc[col][0] && col_oc[col][1]) // Case 2
                res = Math.max(res, topmax - minh[col][1] + 1);
            if (col_oc[col][0] && col_oc[col][1])  // Case 3
                res = Math.max(res, maxh[col][0] - botmin + topmax - minh[col][1] + 2);
        }
        System.out.println(res);
    }
}
```

7 FRC Spending

Note that for any interval in time $[A, B]$ that doesn't contain any t_1 or t_2, the rate of spending is constant, so total spending in that interval forms a straight line. We can envision the amount spent as a bunch of line segments linked together. Here's the sample input visualized this way:



A: $t = 1$; the first project begins
B: $t = 4$; the second project begins
C: $t = 6$; the first project ends
D: $t = 9$; the second project ends
Red line: \$100 spent

It's apparent now that our answer is the smallest integer x coordinate where the y coordinate at that point is at least k. We can solve this by sweeping over the line segments from left to right. We start with a y value of 0 and a slope of 0. We go to the next "important" x coordinate (one that contains a t_1 or t_2). For each project that is starting there, we increment our slope by s. For a project ending there, we decrement slope by s. We calculate our new y value using: $y_{now} = y_{prev} + m(x_{now} - x_{prev})$ where m is our current slope. If $y_{now} \geq k$, we know our answer is somewhere in $[x_{prev}, x_{now}]$. Using the same equation, we can calculate exactly where the intersection occurred, giving us our answer. This is a $O(p \log p)$ (due to sort) solution.

```
import java.util.*;

public class P8_2 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int p = in.nextInt();
        long k = in.nextLong();
        long[][] ps = new long[2*p][2]; // [][0]=time; [][1]=change in slope

        for (int i = 0; i < p; i++) {
            long t1 = in.nextLong(), t2 = in.nextLong(), s = in.nextLong();
            ps[2*i][0] = t1;
            ps[2*i][1] = s;
            ps[2*i + 1][0] = t2;
            ps[2*i + 1][1] = -s;
        }
        Arrays.sort(ps, new Comparator<long[]>() {
            public int compare(long[] a, long[] b) {
                return Long.compare(a[0], b[0]); // sort by time
            }
        });
        // c/p: current/previous; m=money, r=rate, t=time
        long cm = 0, cr = 0, pt = ps[0][0], pm = 0;
        for (int i = 0; i < 2 * p; ) {
            long ct = ps[i][0];
```

```
            cm += (ct - pt) * cr;
            if (cm >= k) {
                long at = (k - pm + cr - 1) / cr; // trick to round up
                System.out.println(pt + at);
                return;
            }

            for (; i < 2 * p && ps[i][0] == ct; i++)
                cr += ps[i][1];
            pm = cm;
            pt = ct;
        }
        System.out.println(-1);
    }
}
```