Solutions by Thomas Breydo (thomasbreydo.com).

Your Journey Begins Here

This was meant to be an introductory problem. We made explanatory videos that show you how to use Hackerrank and solve this problem:

- https://youtu.be/TRUKo-oeR44 (uses Java)
- https://youtu.be/ezZ62SzsXqo (uses Python)
- https://youtu.be/YrXuhwnfYUY (uses JavaScript)

What to Bring on Vacation

This was also meant to be an introductory problem and we made explanatory videos for it as well:

- https://youtu.be/90-RDs7IiWk (uses Java)
- https://youtu.be/XJT9RuhjG-I (uses Python)
- https://youtu.be/rlvyOQxwpsE (uses JavaScript)

Largest Perfect Square

Method 1

We can loop over all numbers k between 0 and n, stopping as soon as

$$(k+1)^2 > n.$$

At this point, k^2 will be the largest perfect square less than or equal to n, since $(k+1)^2$ is larger than n. It can be shown that this method runs in $\mathcal{O}(\sqrt{n})$ time. This is fast enough given our constraints for n.

Method 2

We can avoid any loops by noticing that, at the end of the previous solution, we always have

$$k = \lfloor \sqrt{n} \rfloor.$$

As with the previous solution, the largest perfect square less than or equal to n will be

$$k^2 = \lfloor \sqrt{n} \rfloor^2.$$

This method runs in $\mathcal{O}(1)$ time, which is much better. Here is the Python implementation:

```python
def largestPerfectSquare(n):
    return math.floor(math.sqrt(n)) ** 2
```

Here is the C++ implementation:

```cpp
int largestPerfectSquare(int n) {
    int k = floor(sqrt(n));
    return k * k;
}
```

Money for a Vacay

We can compute the amount of money you have after the transactions as follows:

$$total = \sum(transactions) + startBalance.$$

We then check if $total < 1145$ to see if we should return `YES` or `NO`.

Here is the Python implementation:

```python
def enoughForVacation(startBalance, transactions):
    return 'NO' if sum(transactions) + startBalance < 1145 else 'YES'
```

Here is the C++ implementation:

```cpp
string enoughForVacation(int startBalance, vector<int> transactions) {
    int total = startBalance;
    for (int t : transactions) total += t;
    return total >= 1145 ? "YES" : "NO";
}
```

Colorful Criminals

We can check each description in the *log* for one that matches *desc*. Here is the Python implementation:

```python
LETTER_TO_NAME = {"m": "Maxine", "t": "Tabitha", "r": "Ryan"}

def findKiller(log, desc):
    killer_wearing = desc.split()
    for line in log:
        first_letter, *wearing = line.split("-")
        if wearing == killer_wearing:
            return LETTER_TO_NAME[first_letter]
```

Donna's Function

A naive recursive approach such as the following would take too long:

```
def D(n):
    if n == 1:
        return 3
    if n == 2:
        return 9
    return D(n - 1) - D(n - 2)
```

Thankfully, the values for $D(n)$ cycle every 6 values of n:

- $n = 1 \Rightarrow D(n) = 3$
- $n = 2 \Rightarrow D(n) = 9$
- $n = 3 \Rightarrow D(n) = 6$
- $n = 4 \Rightarrow D(n) = -3$
- $n = 5 \Rightarrow D(n) = -9$
- $n = 6 \Rightarrow D(n) = -6$
- $n = 7 \Rightarrow D(n) = 3$ (start of a new cycle)
- $n = 8 \Rightarrow D(n) = 9$
- $n = 9 \Rightarrow D(n) = 6 \ldots$

We can use this to write the following $\mathcal{O}(1)$ solution:

```
def D(n):
    answers = [3, 9, 6, -3, -9, -6]
    return answers[(n - 1) % 6]
```

Float Over the Mountains

(Solution by Adam Boesky.)

After getting past the scientific terminology, this problem becomes fairly basic implementation. Because energy is only gained or lost when we change altitudes, we can loop through the altitudes and calculate the energy that is gained or lost for each change in altitude and calculate the sum. The routine for calculating the energy is as follows.

Let the altitude before the transfer be A_1 and the altitude after be A_2. The first step is to calculate the change in temperature

$$\Delta T(A_1, A_2) = (T_0 - CA_2) - (T_0 - CA_1).$$

where C is the temperature lapse rate $= 0.0065$ and T_0 is the temperature at sea level $= 288.15$. The Python implementation of this is as follows:

```
def calculate_d_T(alt, next_alt):
   T_alt = 288.15 - (0.0065 * alt)
```

```
    T_next_alt = 288.15 - (0.0065 * next_alt)
    return T_next_alt - T_alt
```

Then, we must calculate the heat energy transfer which is simply calculated as

$$q = mc\Delta T$$

where m is mass of the air inside the balloon $= 2686.2$ and c is the specific heat of the air $= 0.717$. The Python implementation of this is as follows:

```
def calculate_d_E(d_T):
     return (2686.2 * 0.717 * d_T)
```

Finally, we calculate the amount of energy gained or lost as a function of the heat energy transfer which incorporates the heating and cooling efficiency. If we are heating the balloon, the energy $E = q \div 0.95$. If we are cooling the balloon, the energy $E = q \times 0.25$. We then add this value to the total energy.

Using the functions above, the Python implementation of this is:

```
def calculateEnergy(d_alts):
    alt = 300 + d_alts[0] # Initial altitude
    E_tot = 0 # Energy
    for i in range(1, len(d_alts)):
        next_alt = 300 + d_alts[i] # Next altitude
        d_T = calculate_d_T(alt, next_alt) # Change in temp
        d_E = calculate_d_E(d_T) # Heat energy transferred
        if d_E >= 0: # Add the energy transferred taking efficiency into account
            E_tot = E_tot + (d_E / 0.95)
        else:
            E_tot = E_tot + (d_E * 0.25)
        alt = next_alt
    return E_tot
```

Zip away to Zurich

A clever trick to solve this problem is to note that the answer is equal to the XOR sum of all sold tickets. This is because

$$a \oplus a \oplus b \oplus b \oplus \cdots \oplus y \oplus y \oplus z = z.$$

(In other words, all duplicate tickets cancel with themselves when XOR-ed.) As such, taking the XOR sum all numbers is equivalent to finding the number that appears only once.

Here is the implementation in Python:

```python
def availableTicket(soldTickets):
    return functools.reduce(lambda x, y: x ^ y, soldTickets)
```

Here is the implementation in C++:

```cpp
int availableTicket(vector<int> soldTickets) {
    int ans = 0;
    for (int x : soldTickets) ans ^= x;
    return ans;
}
```

The Bishop's Walk

The idea is that we can simulate moving the bishop in each of the four diagonal directions until it reaches an obstacle or an edge, keeping track of the total number of squares it can visit as we go.

Here is the implementation in Python:

```python
def valid(pos, n):
    return n >= pos[0] >= 1 and n >= pos[1] >= 1

def moveCount(n, k, r_b, c_b, obstacles):
    occupied = set((row, col) for row, col in obstacles)
    moves = [(-1, -1), (-1, 1), (1, -1), (1, 1)]
    ans = 0
    for move in moves:
        cur = [r_b, c_b]
        while True:
            cur[0] += move[0]
            cur[1] += move[1]
            if not valid(cur, n) or tuple(cur) in occupied:
                break
            ans += 1
    return ans
```

Here is the implementation in C++:

```cpp
bool valid(pi& pos, int n) {
    return pos.first >= 1 && pos.first <= n && pos.second >= 1 && pos.second <= n;
}

int moveCount(int n, int k, int r_b, int c_b, vector<vector<int>> pieces) {
    vpi moves = {{-1, -1}, {-1, 1}, {1, -1}, {1, 1}};
    set<pi> occupied;
    for (int i = 0; i < pieces.size(); ++i) {
        occupied.insert({pieces[i][0], pieces[i][1]});
    }
```

```
    int ans = 0;
    for (pi move : moves) {
        pi cur = {r_b, c_b};
        while (true) {
            cur.first += move.first;
            cur.second += move.second;
            if (!valid(cur, n) || occupied.count(cur) != 0) break;
            ++ans;
        }
    }
    return ans;
}
```

Plates of Hash (Browns): Part I

We can simulate each step of the hashing process as specified in the problem statement. Here is the implementation in Python:

```
def hashPlate(plate, key):
    return key[int(
        ''.join([c if c.isnumeric() else str(ord(c)) for c in plate])
    ) % len(key)]
```

Plates of Hash (Browns): Part II

Each character can be one of 36 options (26 letters and 10 digits). Since we only consider four-character plates, we can brute force the answer by checking all 36^4 plates with our `hashPlate` function from Part I. Here is the Python implementation:

```
def hashPlate(plate, key):
    return key[int(
        ''.join([c if c.isnumeric() else str(ord(c)) for c in plate])
    ) % len(key)]

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

def numberOfPlates(key, c):
    ans = 0
    for c1 in ALPHABET:
        for c2 in ALPHABET:
            for c3 in ALPHABET:
                for c4 in ALPHABET:
                    if hashPlate(c1 + c2 + c3 + c4, key) == c:
                        ans += 1
```

```
    return ans
```

Note, Python's `itertools` can simplify our quadruple four-loop:

```
import itertools

def hashPlate(plate, key):
    return key[int(
        ''.join([c if c.isnumeric() else str(ord(c)) for c in plate])
    ) % len(key)]

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

def numberOfPlates(key, c):
    ans = 0
    for c1, c2, c3, c4 in itertools.product(ALPHABET, repeat=4):
        if hashPlate(c1 + c2 + c3 + c4, key) == c:
            ans += 1
    return ans
```

Rudolph the Red-Nosed Reindeer

For all $0 \leq i < n$, let $a_i =$ `reindeerIntervals[i]`, the interval at which the i^{th} reindeer arrives. Then, the i^{th} reindeer arrives at timestamps that are $0 \pmod{a_i}$. For example, if $a_{13} = 5$, then the 13^{th} reindeer arrives at timestamps that are $0 \pmod 5$, which are $0, 5, 10, 15$, and so on.

For t to be valid, the i^{th} reindeer must arrive at time $t + i$, so

$$t + i \equiv 0 \pmod{a_i}$$

for all $0 \leq i < n$. This problem reduces to finding the smallest t that satisfies these n modular constraints.

We can satisfy the n conditions by satisfying the largest, then the second largest, and so on, until we have satisfied the smallest.

1. Set $t := 0$.
2. Increment t by 1 until $t + j \equiv 0 \pmod{a_j}$, where a_j is the largest a_i.
3. We have now satisfied the constraint when $i = j$.
4. From here, we increment t by a_j so that we don't lose $t + j \equiv 0 \pmod{a_j}$. Continue incrementing until $t + k \equiv 0 \pmod{a_k}$, where a_k is the second-largest a_i
5. We have now satisfied the constraint when $i = j$ and the constraint when $i = k$ was maintained.

6. From here, we increment t by

$$\text{lcm}(a_j, a_k)$$

so that we don't lose $t + j \equiv 0 \pmod{a_j}$ nor $t + k \equiv 0 \pmod{a_k}$. (This is the key speed-up: we don't need to try any numbers that we know won't satisfy any of the constraints we have already satisfied.)

7. With each new constraint we satisfy, we start to increment t by $\text{lcm}(a_j, a_k, \ldots, a_x)$ to avoid messing up the constraints we have already satisfied (namely, $a_j, a_k, \ldots,$ and a_x). We move on to the next-largest constraint.

Here's the implementation in Python:

```python
def findTimestamp(reindeerTimes):
    a = [(i, a_i) for i, a_i in enumerate(reindeerTimes)]
    # Sort in increasing order because pop() will take from the end
    a.sort(key=lambda x: x[1])
    incr = 1
    t = 0
    while a:
        i, a_i = a.pop()
        while (t + i) % a_i != 0:
            t += incr
        incr = lcm(incr, a_i)
    return t

def lcm(a, b):
    c = math.gcd(a, b)
    return a * b // c
```