SACO 2007 Day 2 Solutions

1 Factorise

The easiest way to solve this problem comes from noticing that the constraints are low enough to test every value less than N to see if it is a factor of N. So, loop from $i = 2$ to $i = N$. If i has no remainder when it is divided into N, then it is clearly a factor of N. The remainder when N is divided by i is written as N `mod` i. If i is a factor of N, keep dividing N by i until it is no longer a factor. Keep track of the number of times you do this. When you can no longer divide N by i, output i and the number of times it divides into N. Then increment i and repeat. Notice that since you are increasing i, the factors will always be output in sorted order.

1.1 Example

As an example, take $N = 45$. Notice that $N = 9 \times 5 = 3 \times 3 \times 5$.

- Start with $i = 2$. Then, N `mod` $i = 45$ `mod` $2 = 1$, so 2 does not divide into 45.
- Next move on to $i = 3$. Since 45 `mod` $3 = 0$, 3 is a factor. Next divide 45 by 3 to get 15. 15 `mod` $3 = 0$, so 3 divides 15. Again, divide 15 by 3 to give 5. Clearly, 3 does not divide into 5, so we are done with 3. Output that 3 divides into 45 a total of 2 times.
- Moving on to $i = 4$, we can see that 4 does not divide into 5.
- Finally, $i = 5$. Since 5 is a factor of 5, we output that 5 divides into N once.

1.2 Pseudo-code

```
i = 2

while N != 1:
    num_of_divides = 0

    while N % i == 0:
        N = N/i
        num_of_divides = num_of_divides+1

    if num_of_divides > 0:
        fout.write('%d %d\n' % (i, num_of_divides))

    i = i + 1
```

2 French Taunter

Using a brute force algorithm, whereby for each of the W words you attempt to match it in the array using `indexOf()`,`in`, `find()` or other similar methods of physical matching would work fine for small values of N and M, however as N and M tend to their upper constraints, the amount of possibilities of placings for words increases very rapidly, making the running time of a brute force solution increase beyond the allowed running time.

The more efficient solution, and the intended solution, is to implement an efficient data structure such as a *trie*, *hashtable* or *dictionary*. (A sorted array with a binary look-up should work fine in Pascal where a hashtable or dictionary does not exist). This allows you to preprocess either the possible W words, or as the model solution does, all possible words in the grid, into a structure whereby you can search for them quickly, based rather on the length of the word searching for, as opposed to the number of words that exist is the list.

3 Living Dead Parrot

The key observation in parrots is that if you know there are A_1 alive parrots in the range $[a, b]$ and there are A_2 alive parrots in the range $[a, c]$ (where $c < b$) then there are $A_1 - A_2$ alive parrots in the range $[c + 1, b]$.

Using this fact, you can use a divide and conquer approach to solving the problem. This approach is very similar to the binary search which you may have learned in computer science.

3.1 Psuedo-code

```
alive_parrots = boolean array of size N
set each value in alive_parrots to false

function search(a, b, num_alive):
  if (num_alive == 0)
     return
  if (b - a + 1 == num_alive)
     set alive_parrots to true in range [a,b]

  c = (a+b)/2
  q = query(a,c)

  search(a, c, q)
  search(c+1, b, num_alive - q)

search(1, N, query(1,N))
```

4 How Not To Be Seen

A brute-force solution would generate all possible placements of the spies and check which of them satisfy the required conditions. This solution should get 50%.

To get 100%, however, you have to use a technique known as Dynamic Programming. The idea is to break the large problem into many smaller problems that we can easily solve and then combine the smaller solutions to get the solution to the large problem.

Start by defining $A_{n,m,k}$ as the number of placements of k spies in an $n \times m$ garden. The placements can be described by the following four possibilities:

- There are $A_{n-1,m,k}$ placements with no spies in row n.
- There are $mA_{n-1,m-1,k-1}$ placements with one spy in row n that is *not* in the sight of any other spies. No other spies can be placed in row n or in the column of this spy. There are m possible places to put the spy on row n.
- There are $\frac{m(m-1)}{2}A_{n-1,m-2,k-2}$ placements with two spies in row n. No other spies can be placed in row n or in the columns of these two spies. There are m possible placements for the first spy and $m-1$ for the second spy. To adjust for spies taking the same positions, but just swapping positions, we halve this number.
- There are $m(n-1)A_{n-2,m-1,k-2}$ placements with one spy which is in the sight of another spy in another row. No other spies can be placed in row n or in the column of this spy. There are m possible paces to put the spy on row n and $n-1$ possible places of the spy in sight.

Combining these four possibilities we get the formula:

$$\begin{aligned} A_{n,m,k} = \quad & A_{n-1,m,k} + \\ & mA_{n-1,m-1,k-1} + \\ & \tfrac{m(m-1)}{2} A_{n-1,m-2,k-2} + \\ & m(n-1)A_{n-1,m-2,k-2} \qquad (1) \end{aligned}$$

We know that $A_{0,m,k} = A_{n,0,k} = 0$ and $A_{n,0,0} = A_{0,m,0} = A_{n,m,0} = 1$ for all n, m, k. Consider the n^{th} row of the garden. Using these as the base cases we can iterate through the n, m, k in increasing order calculating all the values $A_{n,m,k}$ and the answer will be $A_{N,M,K}$.

4.1 Pseudo-code

All values $A_{n,m,k}$ with any of n, m, k out of bounds are 0.

```
Set base cases as outlined above
for n in 1..N
    for m in 1..M
        for k in 1..K
            A[n][m][k] = <insert formula 1 here>
```

5 The Crimson Permanent Assurance

See presentation.