BFAM

2020 Mar 10 - Hackers rank (2 median level puzzles + some quant questions)
2020 Mar 13 – Onsite interview, meeting Johan / Jim and An

Q1. Knight moves

Given a chess board of size `NxN`, in each move, a knight can either :

- move 2 column positions to the left or right *AND* 1 row position up or down, *OR*
- move 1 column position to the left or right *AND* 2 row positions up or down

Given starting coordinate A and ending coordinate B, find the minimum number of moves needed, or `-1` for no solution. Here is my dynamic programming solution (region growing) in Python.

```
def minMoves(n, startRow, startCol, endRow, endCol):
    num_moves = [[float('inf')]*n for m in range(n)]
    queue = [[startRow, startCol, 0]]
    while len(queue) > 0 :
        node = queue.pop(0)

        if node[0] == endRow and node[1] == endCol : return node[2]
        if node[0]+1 < n and node[1]+2 < n and num_moves[node[0]+1][node[1]+2] > node[2]+1 :
            num_moves[node[0]+1][node[1]+2] = node[2]+1
            queue.append([node[0]+1, node[1]+2, node[2]+1])
        if node[0]+2 < n and node[1]+1 < n and num_moves[node[0]+2][node[1]+1] > node[2]+1 :
            num_moves[node[0]+2][node[1]+1] = node[2]+1
            queue.append([node[0]+2, node[1]+1, node[2]+1])
        if node[0]-1 > -1 and node[1]+2 < n and num_moves[node[0]-1][node[1]+2] > node[2]+1 :
            num_moves[node[0]-1][node[1]+2] = node[2]+1
            queue.append([node[0]-1, node[1]+2, node[2]+1])
        if node[0]-2 > -1 and node[1]+1 < n and num_moves[node[0]-2][node[1]+1] > node[2]+1 :
            num_moves[node[0]-2][node[1]+1] = node[2]+1
            queue.append([node[0]-2, node[1]+1, node[2]+1])
        if node[0]+1 < n and node[1]-2 > -1 and num_moves[node[0]+1][node[1]-2] > node[2]+1 :
            num_moves[node[0]+1][node[1]-2] = node[2]+1
            queue.append([node[0]+1, node[1]-2, node[2]+1])
        if node[0]+2 < n and node[1]-1 > -1 and num_moves[node[0]+2][node[1]-1] > node[2]+1 :
            num_moves[node[0]+2][node[1]-1] = node[2]+1
            queue.append([node[0]+2, node[1]-1, node[2]+1])
        if node[0]-1 > -1 and node[1]-2 > -1 and num_moves[node[0]-1][node[1]-2] > node[2]+1 :
            num_moves[node[0]-1][node[1]-2] = node[2]+1
            queue.append([node[0]-1, node[1]-2, node[2]+1])
        if node[0]-2 > -1 and node[1]-1 > -1 and num_moves[node[0]-2][node[1]-1] > node[2]+1 :
            num_moves[node[0]-2][node[1]-1] = node[2]+1
            queue.append([node[0]-2, node[1]-1, node[2]+1])
```

Q2. Factorial remainder

Given an integer `N`, determine the number of positive integers less than or equal to `N` for which the following holds true :

`(x-1)! % x = x-1` where `x <= N` under constraint that `1 <= N <= 10^5`

For example when `N=15`, the output is 7, here are the cases :

```
(1-1)! % 1 = 0! % 1 = 1 % 1 = 0 = 1-1
(2-1)! % 2 = 1! % 2 = 1 % 2 = 1 = 2-1
(3-1)! % 3 = 2! % 3 = 2 % 3 = 2 = 3-1
(5-1)! % 5 = 4! % 5 = 24 % 5 = 4 = 5-1
(7-1)! % 7 = 6! % 7 = 720 % 7 = 6 = 7-1
(11-1)! % 11 = 10! % 11 = 3628800 % 11 = 10 = 11-1
(13-1)! % 13 = 12! % 13 = 479001600 % 13 = 12 = 13-1
```

<u>Solution 1 – William theorem</u>

William theorem states that integer fulfilling property `(x-1)! % x = x-1` must be prime numbers, therefore it is equivalent to counting prime numbers.

```
def factorial_remainder(n):
    prime = [1]*n
    for m in range(2,n) :
        x = m
        while x <= n-m :
            x = x + m
            prime[x-1] = 0
    return sum(prime)
```

Solution 2 – Direct method

Since Python can handle big number and modulus of big number, we can adopt the direct implementation (but slow).

```
def factorial_remainder(n):
    count = 1
    value = 1
    for m in range(2,n+1) :
        value = value * (m-1)
        if value % m == m-1 : count = count+1
    return count
```

Solution 3 – Direct method

As I suspect the line `value % m == m-1` in above solution may be slow, I then make it iterative as follows. Given that :

```
(x-2)! % (x-1) = N[x-1] ... R[x-1]
```

then we derive the formula for `x-1` on top of `x-2`, we have :

```
  (x-1)! % x
= (x-2)!(x-1) % x
= (N[x-1](x-1) + R[x-1])(x-1) % x
=  N[x-1](x-1)^2 % x + R[x-1](x-1) % x
=  N[x-1](x^2-2x+1) % x + R[x-1](x-1) % x
=  N[x-1](x-2) + R[x-1] + N[x-1] % x - R[x-1] % x
=  N[x-1](x-2) + R[x-1] + (N[x-1] - R[x-1]) % x
```

Hence we have an iterative implementation as :

```
def factorial_remainder(n):
    # (x-1)! = Nx+R
    x = 1
    N_old = 1
    R_old = 0
    count = 1

    # Lets update  N and R iteratively
    for x in range(2,n+1) :
        N_new =  N_old * x + R_old + math.floor((N_old * (-2*x+1) - R_old) / x)
        R_new = (N_old * (-2*x+1) - R_old) % x
        N_old = N_new
        R_old = R_new
        if R_new == x-1 : count = count + 1
    return count
```

Both `N_new` and `N_old` are big numbers. Big number in Python supports modulus operation, but not division operation, so the above implementation will fail (throw) in line `math.floor((N_old * (-2*x+1) - R_old) / x)` when `N_old` becomes too big.

Onsite interview question – by Jim

Given a list of integers `[1, 2, ..., N]` find size of maximum subset, such that no two elements in the subset differ by :

- `disallowed_diff[0]`
- `disallowed_diff[1]`
- `disallowed_diff[2] ...`

Original question by Jim has `N = 13` and `disallowed_diff = {5,8}`. My solution is exhaustive search on a multipartite graph

- time dimension = growing size of subset
- nodes within a layer = `{ subset0, subset1, subset2, ... }` where `sizeof(subset-n) = time-index`

In fact, we don't need to build the whole multipartite graph, we just need to keep this layer and next layer only.

```
def fulfill_constraint(subset, n) :
    for x in subset :
        if x == n : return False
        if abs(x-n) == 5 : return False
        if abs(x-n) == 8 : return False
    return True

# This is a recombining tree, better to be implemented as set-of-set, saving time.
# However Python does not support set of set, perhaps C++ is better.
# Consider two subsets in the 5th layer (1,4,5,8,9) and (9,8,5,4,1),
# they should be considered as the same node, otherwise a waste of time.

def max_subset_no_specific_diff_allowed(N, disallowed_diff) :
    layer = set()
    layer.add(()) # empty tuple
    max_layer = 0
    max_subset = 0

    for n in range(1,N+1) :
        new_layer = set()
        for subset in layer : # subset is a tuple
            for m in range(1,N+1) :
                if fulfill_constraint(subset, m) :
                    temp = list(subset + (m,))
                    temp.sort()
                    new_subset = tuple(temp)
                    new_layer.add(new_subset)
                    if len(new_subset) > max_subset : max_subset = len(new_subset)

        layer = new_layer
        # print('\n[layer', n, ']\n', layer, '\n')
        if len(layer) > max_layer : max_layer = len(layer)
        if len(layer) == 0 : break
    return max_layer, max_subset

print(max_subset_no_specific_diff_allowed(13, [5,8]))
print(max_subset_no_specific_diff_allowed(20, [5,8]))
```

<u>BFAM's possible answer (my speculation)</u>

I guess Jim did not expect an algorithmic answer for general N case, instead he expected deduction method specifically for `N = 13` and `disallowed_diff = {5,8}`. This is greedy algorithm.

```
if 01 is _included in our set, then 01+05=06 and 01+08=09 are forbidden    ... [1]              [6,9]
as 09 is forbidden in our set, then 09-05=04 can be included in our set    ... [1,4]            [6,9]
as 04 is _included in our set, then 04+05=09 and 04+08=12 are forbidden    ... [1,4]            [6,9,12]
as 12 is forbidden in our set, then 12-05=07 can be included in our set    ... [1,4,7]          [6,9,12]
as 07 is _included in our set, then 07-05=02 and 07+05=12 are forbidden    ... [1,4,7]          [2,6,9,12]
as 02 is forbidden in our set, then 02+08=10 can be included in our set    ... [1,4,7,10]       [2,6,9,12]
as 10 is _included in our set, then 10-08=02 and 10-05=05 are forbidden    ... [1,4,7,10]       [2,5,6,9,12]
as 05 is forbidden in our set, then 05+08=13 can be included in our set    ... [1,4,7,10,13]    [2,5,6,9,12]
as 13 is _included in our set, then 13-08=05 and 13-05=08 are forbidden    ... [1,4,7,10,13]    [2,5,6,8,9,12]
as 08 is forbidden in our set, then 08-05=03 can be included in our set    ... [1,3,4,7,10,13]  [2,5,6,8,9,12]
as 03 is _included in our set, then 03+05=08 and 03+08=11 are forbidden    ... [1,3,4,7,10,13]  [2,5,6,8,9,11,12]
```

We can retry with another initial number instead of 1, we can derive possibly a different subset with size 6.