Problem 2

> Refs:
>
> [1] http://web.ntnu.edu.tw/~algo/Substring.html#6
> [2] https://www.youtube.com/watch?v=CpZh4eF8QBw

1.

```
function query(l1, l2, n)
    if subs_cmp(l1, l2) ≥ n
        return True
    else
        return False

cmp_history = N*N array filled with -1
function subs_cmp(l1, l2)
    if l2 < l1
        swap(l1, l2)
    if l2 > N
        return 0
    if cmp_history[l1][l2] == -1 /* array has initial value -1 */
        if S[l1] ≠ S[l2]
            cmp_history[l1][l2] = 0
        else
            cmp_history[l1][l2] = 1 + subs_cmp(l1+1, l2+1)
    return cmp_history[l1][l2]
```

Explanation

The function `subs_cmp(l1, l2)` returns `m` such that `m` is the largest number that satisfies `S[l1..l1+m-1] == S[l2..l2+m-1]`. The workflow of it is direct comparison of the two strings with caching.
`query(l1, l2, n)` simply check if `subs_cmp(l1, l2)` is larger than `n`.

Space complexity

`cmp_history` is a `N*N` array storing single values in each slot, therefore the space complexity is $O(N^2)$

Time complexity

Each query takes $O(N)$ time because it's directly comparing two strings. Therefore time complexity for Q queries should be $O(QN)$. Caching should help reducing constants but I am not sure if time complexity could be tighter.

2.

`X = {8, 0, 0, 0, 3, 0, 0, 0}`

3.

```
function generateX(S)
    N = S.len
    X = array(N)
    l = 1 /* left bound of furthest interval */
    r = 1 /* right bound of furthest interval */
    for i=2 to N
        if i > r /* not in current interval */
            l = i
            r = i
            while r ≤ N and S[r] == S[r-l+1] /* project r to its position at prefix
*/
                r += 1
            r -= 1
            X[i] = r-l+1
        else
            i_prime = i-l+1
            if i+X[i_prime]-1 < r /* doesn't extend over current interval */
                X[i] = X[i_prime]
            else
                l = i
                while r ≤ N and S[r] == S[r-l+1]
                    r += 1
                r -= 1
                X[i] = r-l+1
    X[1] = N
    return X
```

Explanation

> **prefix-substring at** `i`: Longest substring starts from `S[i]` such that it is also the prefix of `S`
>
> **interval**: The the prefix-substring furthest to the right we have found so far. Bounded by `l` and `r`

The workflow is:

- Skipping `X[1]` because it's definitely `N`
- Iterate `i` from `2` to `N`. For each `i`:
  - Check if `i` is in the interval:
    - If not, set both bounds to `i`. Then compare the substring to prefix character by character and increase `r` accordingly. Substract `1` from `r`. Set `X[i] = r-l+1`.

        - If it is in the interval, let `i_prime` be that corresponding position of `i` in prefix. Check if `X[i_prime]` makes the new prefix-substring touches the right bound of interval:
            - If not, then the prefix-substring at `i` must be the same as that at `i_prime`. The next `X[i_prime]+1` characters are the same thus the `X[i] == X[i_prime]`.
            - If it does, move the left bound to `i`. Then compare the substring to prefix character by character starting from `r` and increase `r` accordingly. Substract `1` from `r`. Set `X[i] = r-l+1`.
- Set `X[1]=N`.
- Return `X`

Space Complexity

Extra variables used are `N`, `l`, `r`, `i`, `i_prime` and are all single values. Therefore extra space complexity is $O(1)$.

Time Complexity

After the `for` loop ends, the `while` loop in it would take $O(N)$ time. Because each time `while` is executed `r` would increase by `1`, but `r` has upper bound `N` and `r -= 1` would be executed at most `N-2` times (in every iteration of `for`).
Not considering the `while` inside, the `for` loop clearly takes $O(N)$ time to complete. Therefore total time complexity is $O(N)$ (`while`) $+O(N)$ (`for` without `while) $+O(1) = O(N)$.

4.

```
function pattern_count(p, t)
    c = "$"
    S = p+c+t
    X = generateX(S) /* from previous subproblem */
    p_l = p.len
    cnt = 0
    for i=1 to S.len
        if X[i] == p_l
            cnt += 1
    return cnt
```

> Assuming `p` and `t` only contains uppercase alphabets.
> m : Length of `p`.
> n : Length of `t`.

Explanation

The workflow is:

1. Concatenate `p`, `c`, and `t`. `c` can be any character not in the character set of `p` and `t`. Here I choose `c = "$"`. The concatenated string is called `S`.
2. Build the `X` array of `S`.

3. Traverse `X`, check if `X[i] == p.len`. If true, the prefix-substring of `S` at `i` contains `p`, therefore add `1` to `cnt`.
4. Return `cnt`

By making the pattern the prefix of a string, we can utilize `X` to match pattern. And since `S[p.len+1]` is a character not in either `p` or `t`, `X[i]` is guaranteed to be less than `p.len`, thus we can use `X[i] == p_l`.

Space Complexity

`S` and `X` both take $O(m) + O(1) + O(n) = O(m + n)$ space. Other extra variables just store single value therefore take $O(1)$ space. Total extra space complexity is $O(m + n)$.

Time Complexity

`generateX(S)` and the `for` loop both take $O(m + n)$ time. Other operations take $O(1)$ time combined. Therefore total time complexity is $O(m + n)$.
