Name & Surname:
ID:

CSE 321 - Quiz #4

December 6th, 2024

1. ***50 pts.*** The XOR cipher is a simple encryption technique that uses the exclusive OR (XOR) bitwise operation to encrypt and decrypt data. It is a symmetric key cipher, meaning the same key is used for both encryption and decryption.

   Encryption: Each bit of the plaintext is XORed with the corresponding bit of the key. If the key is shorter than the plaintext, it is repeated cyclically.

   $C_i = P_i \oplus K_i$

   where C is ciphertext, P is plaintext, K is key, and i is the index.

   Decryption: Each bit of the ciphertext is XORed with the corresponding bit of the key. If the key is shorter than the ciphertext, it is repeated cyclically.

   $P_i = C_i \oplus K_i$

   where P is plaintext, C is ciphertext, K is key, and i is the index.

   Sample Encryption Operation:
   Assume that the following is given:
   Plaintext: 101100001
   Key: 1100
   Since plaintext has 9 bits, the key repeats itself cyclically and becomes 110011001. Now the ciphertext can be obtained through XOR operation.
   $C = P \oplus K = 101100001 \oplus 110011001 = 011111000$

   Sample Decryption Operation:
   Assume that the following is given:
   Ciphertext: 011111000
   Key: 1100
   Since ciphertext has 9 bits, the key repeats itself cyclically and becomes 110011001. Now the plaintext can be obtained through XOR operation.
   $P = C \oplus K = 011111000 \oplus 110011001 = 101100001$

   Design a brute-force algorithm to decrypt a given ciphertext, with a length of n, and obtain the plaintext without knowing the key or its length. Provide the pseudo-code of your algorithm along with an explanation and analyze its worst-case time complexity.

   Assumptions:

   - You can confirm a plaintext candidate's correctness in constant time.

   - The key's length is either equal to or smaller than the length of plaintext/ciphertext.

Solution:

```
XOR_Decipher (ciphertext):
    Input: A ciphertext.
    Output: Deciphered text, plaintext.
    begin
        n ← length(ciphertext)
        for key in range (0, $2^n$) repeat
            plaintext_candidate ← XOR(ciphertext, key)
            correctness ← ConfirmCorrectness(plaintext_candidate)
            if correctness == True then
                return plaintext_candidate
            end if
        end for
    end
```

In order to obtain the plaintext from a given ciphertext, we must exhaustively try all possible keys. In the question, it is remarked that the key length is unknown. However, we know that even if the length of the key is shorter than the length of the cipher text, we should cyclically repeat the key and obtain a binary sequence with a length of n. As known, a binary sequence with n bits might have a value in a range of $[0, 2^n)$. Thus, we try each value within this range and try to find the actual key. Since *XOR* and *ConfirmCorrectness* functions can be executed in constant time, the complexity of the algorithm given above is $O(2^n)$.

2. ***50 pts.*** In software compilation, source files must be processed in a specific order due to dependencies between them. The task is to determine a proper order to compile the modules, ensuring that no dependency errors occur.

Example:
A project consists of modules A, B, and C with the following dependencies:
- Module A depends on Module B.
- Module B depends on Module C.

Thus, Module C must be compiled first, followed by Module B, and finally Module A.

Design a decrease and conquer algorithm to solve this problem. Provide the pseudo-code of your algorithm along with an explanation and analyze its worst-case time complexity.

Solution: This problem can be solved with Topological Sorting since the dependencies yield a Directed Acyclic Graph (DAG). The solution was already shared with you through previous homeworks, but you can also find the pseudo-code below.

```
ModuleSorter ($G$):
    Input: The dependency graph, $G$.
    Output: A proper path that includes all nodes in $G$ without causing dependency errors.
    begin
        in_degrees ← ComputeInDegrees(G)
        queue ← [ ]
        path ← [ ]
        for each vertex $v$ in $G$ repeat
            if in_degrees[v] == 0 then
                append $v$ to queue
            end if
        end for
        while queue is not empty repeat
            v ← Pop(queue)
            append $v$ to path
            for each neighbor $u$ of $v$ repeat
                in_degrees[u] = in_degrees[u] - 1
                if in_degrees[u] == 0 then
                    append $v$ to queue
                end if
            end for
        end while
        return path
    end

ComputeInDegrees ($G$):
    Input: A graph, $G$.
    Output: The dictionary of the in-degrees of the vertices, in_degrees.
    begin
        in_degrees ← { }
        V ← VertexList(G)
        for each vertex $v$ in $V$ repeat
            in_degrees[$v$] ← in_degrees[$v$] + 1
        end for
        return in_degrees
    end
```

This solution benefits the fact that there is at least one vertex with 0 in-degree in a DAG. A vertex with 0 in-degree means that the module represented by that vertex doesn't have dependencies and can be compiled first. Once we compile that module, we can also compile the ones that are only dependent on it. This operation is handled in the inner **for** loop. In this loop, we decrease the in-degrees of the neighbors of this vertex (let's call it v). After the decreasing operation, if the in-degree of a neighbor becomes 0, it means that the neighbor was only depending on v, therefore we can compile the module it represents.

As seen, this algorithm decreases the problem space by one at each step. I.e., at each step, we determine the order of a module and then focus on the others. Since this solution visits each node and edge (to find the in-degrees and determine the neighborhood), the time complexity is $O(v + e)$ where v is the number of vertices and e is the number of edges in the dependency graph.

A solution based on DFS is also accepted since DFS is a decrease-and-conquer algorithm itself. You can also check the solution to the first question in a previous homework (Homework 3, shared with you through Teams) to find additional information.