# Analysis of Algorithms - Assignment 4

**Zhenrui Zheng**

Chinese University of Hong Kong, Shenzhen
`225040512@link.cuhk.edu.cn`

## Contents

# Shortest String Containing Three Substrings

## 1.1 Algorithm Description

We need to find the shortest string $s$ such that all three strings $a$, $b$, and $c$ are continuous substrings of $s$.

The key insight is to consider all possible orderings of the three strings and compute the maximum overlap between consecutive strings using the KMP prefix function. For each ordering, we merge strings by finding the longest suffix of the first string that matches a prefix of the second string.

```
# Input: Three strings a, b, c
# Output: Length of shortest string containing all three as substrings
def shortest_string(a, b, c):
    S = [a, b, c]
    minLen = float('inf')

    # Try all permutations
    for s1, s2, s3 in permutations(S):
        merged = s1
        merged = merge(merged, s2)
        merged = merge(merged, s3)
        minLen = min(minLen, len(merged))

    return minLen

def merge(s1, s2):
    # Compute prefix function pi for pattern s2
    # Find longest suffix of s1 that matches prefix of s2 using pi
    overlap = find_max_overlap(s1, s2)
    return s1 + s2[overlap:]
```

## 1.2 KMP Prefix Function for Overlap Calculation

To compute the maximum overlap between two strings $s_1$ and $s_2$, we use the KMP prefix function. We construct the prefix function $\pi$ for the pattern $s_2$, then search for $s_2$ in $s_1$ starting from position $|s_1| - |s_2| + 1$ (to find suffix matches).

The prefix function $\pi[i]$ for a string $s$ is defined as the length of the longest proper prefix of $s[1..i]$ that is also a suffix of $s[1..i]$.

```
# Input: String s of length n
# Output: Prefix function array pi[0..n-1]
def kmp_prefix_function(s):
    n = len(s)
    pi = [0] * n
    k = 0

    for i in range(1, n):
        while k > 0 and s[k] != s[i]:
            k = pi[k - 1]
        if s[k] == s[i]:
            k += 1
        pi[i] = k

    return pi
```

To find the maximum overlap where a suffix of $s_1$ matches a prefix of $s_2$, we construct the string $s = s_2 + \# + s_1$ where $\#$ is a special delimiter character not in the alphabet. We compute the prefix function $\pi$ for $s$, and the maximum overlap is the maximum value of $\pi[i]$ for $i > |s_2| + 1$ (i.e., within the $s_1$ portion). This gives us the length of the longest suffix of $s_1$ that matches a prefix of $s_2$.

## 1.3 Time Complexity Analysis

For each permutation of the three strings:

- Computing the prefix function for merging two strings: $O(|s_2|)$
- Finding the overlap: $O(|s_1| + |s_2|)$
- Total for one permutation: $O(|a| + |b| + |c|)$

Since there are $3! = 6$ permutations, the total time complexity is $O(|a| + |b| + |c|)$, which is linear in the total input size.

# Maximum XOR of Two Signals

## 2.1 Algorithm Description

We need to find the maximum XOR value between any two signals. This problem can be efficiently solved using a binary Trie (prefix tree) data structure.

The key idea is to build a Trie containing all signal values in their binary representation, then for each signal, traverse the Trie to find the value that maximizes the XOR with the current signal.

```
# Input: Array A of n signal values
# Output: Maximum XOR value between any two signals
def max_xor(A):
    T = Trie()
    # Build Trie by inserting all values
    for x in A:
        T.insert(x)

    maxXOR = 0
    for x in A:
        candidate = T.find_max_xor(x)
        maxXOR = max(maxXOR, x ^ candidate)

    return maxXOR
```

## 2.2 Trie Construction and Query

The Trie stores binary representations of numbers, where each node has two children (for bits 0 and 1). To maximize XOR, we always try to take the opposite bit at each level.

```
class TrieNode:
    def __init__(self):
        self.child = [None, None]  # for bits 0 and 1
        self.value = None

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, x):
        node = self.root
        # Process bits from MSB to LSB (assuming 64-bit integers)
        for i in range(63, -1, -1):
            bit = (x >> i) & 1
            if node.child[bit] is None:
                node.child[bit] = TrieNode()
            node = node.child[bit]
        node.value = x  # Store value at leaf node

    def find_max_xor(self, x):
        node = self.root
        for i in range(63, -1, -1):
            bit = (x >> i) & 1
            target_bit = 1 - bit  # Opposite bit for maximum XOR
            if node.child[target_bit] is not None:
                node = node.child[target_bit]
            else:
                node = node.child[bit]
        return node.value  # Return stored value at leaf
```

## 2.3 Time Complexity Analysis

- **Trie Construction:** For each of $n$ values, we insert it into the Trie. Each insertion takes $O(\log M)$ time where $M$ is the maximum value ($M \leq 10^{18}$, so $\log M \leq 60$). Total: $O(n \log M)$
- **Query Phase:** For each of $n$ values, we query the Trie once. Each query takes $O(\log M)$ time. Total: $O(n \log M)$
- **Overall Complexity:** $O(n \log M)$, which is efficient for $n \leq 50000$ and $M \leq 10^{18}$

# Network Cable Management

## 3.1 Algorithm Description

We need to support two operations:

- **M** $i$ $j$**:** Connect the group containing computer $i$ to the group containing computer $j$, with $i$'s group appearing before $j$'s group
- **C** $i$ $j$**:** Query whether computers $i$ and $j$ are in the same group. If yes, output the number of computers between them; otherwise output $-1$

This problem can be solved using a weighted Union-Find data structure, where the weight represents the depth (distance from the root) of each computer in its group.

## 3.2 Weighted Union-Find Data Structure

We maintain two arrays:

- $parent[i]$: The parent of computer $i$ in the union-find tree
- $depth[i]$: The depth (distance from root) of computer $i$ in its group
- $size[i]$: The size of the group containing computer $i$ (for path compression optimization)

```
# Arrays: parent[1..n], depth[1..n], size[1..n]
# Initialize: parent[i] = i, depth[i] = 0, size[i] = 1 for all i

def find(x):
    if parent[x] != x:
        old_parent = parent[x]
        root = find(parent[x])  # Path compression
        depth[x] = depth[x] + depth[old_parent]  # Update depth to root
        parent[x] = root
        return root
    return x

def union(i, j):
    root_i = find(i)
    root_j = find(j)
    if root_i == root_j:
        return  # Already in same group
    parent[root_j] = root_i  # Connect j's group to i's group
    depth[root_j] = size[root_i]  # Set depth of j's root
    size[root_i] = size[root_i] + size[root_j]

def query(i, j):
    root_i = find(i)
    root_j = find(j)
    if root_i != root_j:
        return -1
    dist_i = depth[i]  # Depth already computed by find
    dist_j = depth[j]  # Depth already computed by find
    return abs(dist_i - dist_j) - 1  # Number of computers between them
```

## 3.3 Correctness Analysis

The algorithm maintains the invariant that for any computer $x$ in a group, $depth[x]$ represents the distance from $x$ to its immediate parent. When we connect group $j$ to group $i$, we set $depth[root_j] = size[root_i]$, which correctly positions all computers in group $j$ after all computers in group $i$.

When querying the distance between two computers $i$ and $j$ in the same group:

- We compute $dist_i$ and $dist_j$ as the distances from $i$ and $j$ to the root
- The number of computers between them is $|dist_i - dist_j| - 1$
- This works because computers are arranged in a linear order within each group

## 3.4 Time Complexity Analysis

- **Find with Path Compression:** Amortized $O(\alpha(n))$ where $\alpha$ is the inverse Ackermann function, effectively constant for practical purposes
- **Union:** $O(\alpha(n))$ amortized time
- **Query:** $O(\alpha(n))$ amortized time for finding roots, plus $O(h)$ time for computing depths, where $h$ is the height of the tree. With path compression, this is effectively $O(\alpha(n))$
- **Overall:** For $T$ operations, the total time complexity is $O(T \cdot \alpha(n))$, which is nearly linear and efficient for $T \leq 500000$

# Placing Tiles

## 4.1 Problem Formulation

We are given an $n \times m$ chessboard where each cell is either empty or contains an obstacle. A cell $(i, j)$ is black if $i + j$ is even, and white otherwise. We need to place L-shaped tiles such that:

- Each tile's corner cell is placed on a black cell
- Each tile covers one black cell and two orthogonally adjacent white cells
- Tiles do not overlap
- Tiles do not cover obstacle cells

Our goal is to maximize the number of tiles placed.

## 4.2 Linear Programming Formulation

We model this problem as a linear programming problem. Let $\mathcal{T}$ be the set of all valid tile placements, where each tile placement $t \in \mathcal{T}$ is defined by:

- A black corner cell $(i, j)$ where $i + j$ is even
- Two orthogonally adjacent white cells that form the L-shape with the corner

For each valid tile placement $t \in \mathcal{T}$, we define a decision variable $x_t \geq 0$ representing whether tile $t$ is placed (or the number of times tile $t$ is placed, though in the optimal solution it will be 0 or 1).

Let $\mathcal{C}$ be the set of all non-obstacle cells. For each cell $c \in \mathcal{C}$, let $\mathcal{T}_c \subseteq \mathcal{T}$ be the set of tile placements that cover cell $c$.

The linear programming formulation is:

$$\text{maximize} \quad \sum_{t \in \mathcal{T}} x_t \quad \text{(maximize total number of tiles placed)} \tag{4.1}$$

$$\text{subject to} \quad \sum_{t \in \mathcal{T}_c} x_t \leq 1 \quad \forall c \in \mathcal{C} \quad \text{(each non-obstacle cell covered by at most one tile)} \tag{4.2}$$

$$x_t \geq 0 \quad \forall t \in \mathcal{T} \quad \text{(tile count must be non-negative)} \tag{4.3}$$

**Explanation of the formulation:**

- **Objective function (4.1):** Maximizes the total number of tiles placed, which is the sum of all decision variables $x_t$ over all valid tile placements $t \in \mathcal{T}$.

- **Coverage constraints (4.2):** For each non-obstacle cell $c \in \mathcal{C}$, the sum of all tile variables $x_t$ that cover cell $c$ must be at most 1. This ensures that no cell is covered by more than one tile, preventing overlaps.
- **Non-negativity constraints (4.3):** All decision variables $x_t$ must be non-negative, ensuring a valid solution.

## 4.3 Properties of the Linear Program

This is a standard linear programming problem in canonical form:

- The objective function is linear and we are maximizing
- All constraints are linear inequalities
- The feasible region is a polytope (bounded polyhedron)
- Since the constraint matrix has only 0-1 entries and the right-hand side is 1, the optimal solution will be integral (each $x_t$ will be 0 or 1) due to the total unimodularity property of the constraint matrix

## 4.4 Implementation Considerations

To solve this linear program:

1. **Generate all valid tile placements:** For each black cell $(i, j)$, enumerate all possible L-shaped placements with corner at $(i, j)$:
    - Check if $(i, j)$ is black and not an obstacle
    - For each of the four possible L-shapes (corner with two adjacent white cells in different orientations), verify that:
        - The two white cells are valid (within board bounds, not obstacles)
        - The cells are orthogonally adjacent to the corner
2. **Build the constraint matrix:** For each non-obstacle cell $c$, create a constraint row where entry $(c, t)$ is 1 if tile $t$ covers cell $c$, and 0 otherwise.
3. **Solve the LP:** Use a linear programming solver (e.g., simplex method, interior point method) to find the optimal solution.

## 4.5 Time Complexity

- **Number of variables:** At most $O(nm)$ valid tile placements (each black cell can have up to 4 possible L-shapes)
- **Number of constraints:** $O(nm)$ (one for each non-obstacle cell)
- **LP solving:** Using the simplex method or interior point methods, the time complexity is polynomial in the number of variables and constraints. For $n, m \leq 50$, this is efficient.

The linear programming approach provides an optimal solution to the tile placement problem, maximizing the number of tiles that can be placed while satisfying all constraints.