N Queens

```
#include <iostream>

#include <vector>

using namespace std;

bool is_safe(vector<vector<int>>& board, int row, int col, int N) {

    // Check column

    for (int i = 0; i < row; i++)

        if (board[i][col] == 1) return false;

    // Check upper-left diagonal

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j] == 1) return false;

    // Check upper-right diagonal

    for (int i = row, j = col; i >= 0 && j < N; i--, j++)

        if (board[i][j] == 1) return false;

    return true;

}

bool solve_n_queens(vector<vector<int>>& board, int row, int N) {

    if (row == N) {

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                cout << (board[i][j] ? "Q " : ". ");
            cout << endl;
        }
        cout << endl;
        return true;
    }
    bool found = false;
    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col, N)) {
            board[row][col] = 1;
            found |= solve_n_queens(board, row + 1, N);
            board[row][col] = 0; // Backtrack
        }
    }
    return found;
}
void n_queens(int N) {
    vector<vector<int>> board(N, vector<int>(N, 0));
    if (!solve_n_queens(board, 0, N))
        cout << "No solution found." << endl;
}
int main() {
    int N;
    cin >> N;
    n_queens(N);
    return 0;
}
```

M X N Queens

```
#include <iostream>
#include <vector>
using namespace std;
bool is_safe(vector<vector<int>>& board, int row, int col, int M, int N)
{
    // Check column
    for (int i = 0; i < row; i++)
```

```
        if (board[i][col] == 1) return false;

    // Check upper-left diagonal

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j] == 1) return false;

    // Check upper-right diagonal

    for (int i = row, j = col; i >= 0 && j < N; i--, j++)

        if (board[i][j] == 1) return false;

    return true;

}

bool solve_n_queens(vector<vector<int>>& board, int row, int M, int
N, int placed) {

    if (placed == N) {

        for (int i = 0; i < M; i++) {

            for (int j = 0; j < N; j++)

                cout << (board[i][j] ? "Q " : ". ");

            cout << endl;

        }

        cout << endl;

        return true;

    }
    if (row >= M) return false;
    bool found = false;
    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col, M, N)) {
            board[row][col] = 1;
            found |= solve_n_queens(board, row + 1, M, N, placed + 1);
            board[row][col] = 0; // Backtrack
        }
    }
    return found;
}
void n_queens_mxn(int M, int N) {
    vector<vector<int>> board(M, vector<int>(N, 0));
    if (!solve_n_queens(board, 0, M, N, 0))
        cout << "No solution found." << endl;
}
int main() {
    int M, N;
    cin >> M >> N;

    n_queens_mxn(M, N);

    return 0;

}
```

Cost optimised board

```
#include <iostream>

#include <vector>

#include <climits>

using namespace std;

int min_cost = INT_MAX;

vector<vector<int>> best_board;

bool is_safe(vector<vector<int>>& board, int row, int col, int N) {

    for (int i = 0; i < row; i++)

        if (board[i][col] == 1) return false;

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

        if (board[i][j] == 1) return false;

    for (int i = row, j = col; i >= 0 && j < N; i--, j++)

        if (board[i][j] == 1) return false;
    return true;
}
void solve(vector<vector<int>>& board, vector<vector<int>>& cost,
int row, int N, int current_cost) {
    if (row == N) {
        if (current_cost < min_cost) {
            min_cost = current_cost;
            best_board = board;
        }
        return;
    }
    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col, N)) {
            board[row][col] = 1;
            solve(board, cost, row + 1, N, current_cost + cost[row][col]);
            board[row][col] = 0;
        }
    }
}
int main() {
    int N;
    cin >> N;
    vector<vector<int>> board(N, vector<int>(N, 0));
    vector<vector<int>> cost(N, vector<int>(N));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            cin >> cost[i][j];
    solve(board, cost, 0, N, 0);
    cout << "Minimum Cost: " << min_cost << endl;
    for (auto& row : best_board) {
        for (int cell : row) cout << (cell ? "Q " : ". ");
        cout << endl;
    }
    return 0;
}
```

Baby Lizards Variant

```
#include <iostream>
```

```
#include <vector>

using namespace std;

vector<vector<int>> trees;

bool is_safe(vector<vector<int>>& board, int row, int col, int N) {

   int i, j;

   // Check column (upwards)

   for (i = row - 1; i >= 0; i--) {

      if (board[i][col] == 1) return false;

      if (trees[i][col] == 1) break; // Stop at a tree

   }

   // Check upper-left diagonal

   for (i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

      if (board[i][j] == 1) return false;

      if (trees[i][j] == 1) break;

   }

   // Check upper-right diagonal

   for (i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) {

      if (board[i][j] == 1) return false;

      if (trees[i][j] == 1) break;
```

```
    }
    return true;
}
bool solve(vector<vector<int>>& board, int row, int N) {
    if (row == N) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j] == 1)
                    cout << "Q ";
                else if (trees[i][j] == 1)
                    cout << "T ";
                else
                    cout << ". ";
            }
            cout << endl;
        }
        cout << endl;
        return true;
    }
    bool found = false;
    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col, N)) {
            board[row][col] = 1;
            found |= solve(board, row + 1, N);
            board[row][col] = 0;
        }
    }

    return found;
}
int main() {
    int N, T, r, c;
    cin >> N >> T;
    vector<vector<int>> board(N, vector<int>(N, 0));
    trees.assign(N, vector<int>(N, 0));
    for (int i = 0; i < T; i++) {
        cin >> r >> c;
        trees[r][c] = 1;
    }

    if (!solve(board, 0, N))

        cout << "No solution exists\n";

    return 0;

}
```

Toroidal Board

```
#include <iostream>
#include <vector>
using namespace std;

bool is_safe(vector<vector<int>>& board, int row, int col, int N) {
    int i, j;

    // Standard column check
    for (i = 0; i < row; i++)
        if (board[i][col] == 1) return false;

    // Standard diagonals
    for (i = row - 1, j = col - 1; i >= 0; i--, j = (j - 1 + N) % N)
        if (board[i][j] == 1) return false;
    for (i = row - 1, j = col + 1; i >= 0; i--, j = (j + 1) % N)
        if (board[i][j] == 1) return false;
    // **Toroidal Column Check (Wraps around)**
    for (i = N - 1; i > row; i--)
        if (board[i][col] == 1) return false;

    // **Toroidal Left Diagonal (Wraps around)**
    for (i = N - 1, j = (col - (N - row) + N) % N; i > row; i--, j = (j - 1 + N)
% N)
        if (board[i][j] == 1) return false;

    // **Toroidal Right Diagonal (Wraps around)**
    for (i = N - 1, j = (col + (N - row)) % N; i > row; i--, j = (j + 1) % N)
        if (board[i][j] == 1) return false;

    return true;
}

bool solve(vector<vector<int>>& board, int row, int N) {
    if (row == N) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                cout << (board[i][j] ? "Q " : ". ");
            cout << endl;
        }
        cout << endl;
        return true;
    }

    bool found = false;
    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col, N)) {
            board[row][col] = 1;
            found |= solve(board, row + 1, N);
            board[row][col] = 0;
        }
    }

    return found;
}

int main() {
    int N;
    cin >> N;
    vector<vector<int>> board(N, vector<int>(N, 0));

    if (!solve(board, 0, N))
        cout << "No solution exists\n";

    return 0;
}
```

Print only one solution

```
bool solve(vector<vector<int>>& board, int row, int N) {
    if (row == N) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                cout << (board[i][j] ? "Q " : ". ");
        cout << endl;
    }
    return true;  //  Return immediately after first solution
  }

  for (int col = 0; col < N; col++) {
    if (is_safe(board, row, col, N)) {
      board[row][col] = 1;
      if (solve(board, row + 1, N))
        return true;  // Stop recursion after first valid board
      board[row][col] = 0;
    }
  }

  return false;
}
```

String Matching

Naive

```
#include <iostream>
using namespace std;

void naive_string_matcher(string T, string P) {
  int N = T.length(), M = P.length();

  for (int i = 0; i <= N - M; i++) {
    int j;
    for (j = 0; j < M; j++) {
            if (T[i + j] != P[j])
                break;
        }
        if (j == M)
            cout << i << endl;
    }
}

int main() {
    string T, P;
    cin >> T >> P;
    naive_string_matcher(T, P);
    return 0;
}
```

Rabin Karp Algorithm

```
#include <iostream>
using namespace std;

#define d 256

void rabin_karp_matcher(string T, string P, int q) {
    int N = T.length();
    int M = P.length();
    int h = 1, p = 0, t = 0;

    // Compute (d^(M-1)) % q
    for (int i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Compute initial hash values for pattern and first window of text
    for (int i = 0; i < M; i++) {
        p = (d * p + P[i]) % q;
        t = (d * t + T[i]) % q;
    }

    // Slide over text to check matches
    for (int i = 0; i <= N - M; i++) {
        if (p == t) {
            bool match = true;
            for (int j = 0; j < M; j++) {
                if (T[i + j] != P[j]) {
                    match = false;
                    break;
                }
            }
            if (match) cout << i << endl;
        }



        if (i < N - M) {
            t = (d * (t - T[i] * h) + T[i + M]) % q;
            if (t < 0) t += q;
        }
    }
}
```

```
int main() {
    string T, P;
    int q;
    cin >> T >> P >> q;
    rabin_karp_matcher(T, P, q);
    return 0;
}
```

Rabin Karp with Spurious Hits

```
#include <iostream>
using namespace std;

int spurious_hits = 0;

int p(char ch) {
    if (ch >= '0' && ch <= '9') return ch - '0';
    if (ch >= 'A' && ch <= 'Z') return ch - 'A' + 10;
    if (ch >= 'a' && ch <= 'z') return ch - 'a' + 36;
    return -1;
}

void rabin_karp_matcher(string T, string P, int d, int q) {
    int n = T.length(), m = P.length();
    if (m > n) return;

    long long pattern_hash = 0, text_hash = 0, h = 1;
    for (int i = 0; i < m - 1; i++)
        h = (h * d) % q;

    for (int i = 0; i < m; i++) {
        pattern_hash = (pattern_hash * d + p(P[i])) % q;
        text_hash = (text_hash * d + p(T[i])) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (pattern_hash == text_hash) {
            if (T.substr(i, m) == P)
                cout << i << endl;
            else
                spurious_hits++;
        }

        if (i < n - m) {
            text_hash = (text_hash - p(T[i]) * h) * d + p(T[i + m]);
            text_hash = (text_hash % q + q) % q;
        }
    }
}

int main() {
    string T, P;
    int d, q;
    cin >> T >> P;
    cin >> d >> q;
    rabin_karp_matcher(T, P, d, q);
    cout << spurious_hits << endl;
    return 0;
}
```

KMP Algorithm

```
#include <iostream>
#include <vector>
using namespace std;

// Compute the LPS (Longest Prefix Suffix) array
void compute_LPS(string P, vector<int>& LPS) {
    int M = P.length(), len = 0;
    LPS[0] = 0;
    int i = 1;

    while (i < M) {
        if (P[i] == P[len]) {
            LPS[i++] = ++len;
        } else {
            if (len != 0)
                len = LPS[len - 1];
            else
                LPS[i++] = 0;
        }
    }
}
```

```
// KMP Pattern Matching Algorithm
void KMP_matcher(string T, string P) {
   int N = T.length(), M = P.length();
   vector<int> LPS(M);

   compute_LPS(P, LPS);

   int i = 0, j = 0;
   while (i < N) {
      if (T[i] == P[j]) {
         i++, j++;
      }
      if (j == M) {
         cout << i - j << endl;
         j = LPS[j - 1];
      } else if (i < N && T[i] != P[j]) {
         if (j != 0)
            j = LPS[j - 1];
         else
            i++;
      }
   }
}

int main() {
   string T, P;
   cin >> T >> P;
   KMP_matcher(T, P);
```

```
    return 0;
}
```

Non Overlapping indices Match text

```
#include <iostream>
using namespace std;
bool check_pattern(string T, string P, int s) {
    int m = P.length();
    for (int i = 0; i < m; i++) {
        if (P[i] != T[s + i])  // Fixed indexing issue
            return false;
    }
    return true;
}
void naive_string_matcher(string T, string P) {
    int n = T.length(), m = P.length();
    for (int i = 0; i <= n - m; i++) {
        if (check_pattern(T, P, i)) {
            cout << i << endl;
            i += m - 1;  // Move index to prevent overlapping matches
        }
    }
}
int main() {
    string T, P;
    cin >> T >> P;
    naive_string_matcher(T, P);
    return 0;
}
```

Rabin Karp

```
#include <iostream>
using namespace std;
#define PRIME 101
// Function to calculate the hash value of a string
long long calculate_hash(string str, int len) {
    long long hash_value = 0;
    for (int i = 0; i < len; i++) {
        hash_value = hash_value * 10 + (str[i] - '0');
    }
    return hash_value;
}
//  search
void rabin_karp(string T, string P) {
    int n = T.length(), m = P.length();
    if (m > n) return;
    long long pattern_hash = calculate_hash(P, m);
    long long text_hash = calculate_hash(T, m);
    for (int i = 0; i <= n - m; i++) {
        // Check hash match
        if (pattern_hash == text_hash) {
            if (T.substr(i, m) == P)  // Verify actual substring match
                cout << i << endl;
        }
        // Update hash using the rolling hash method
        if (i < n - m) {
            text_hash = (text_hash - (T[i] - '0') * pow(10, m - 1)) * 10 + (T[i
+ m] - '0');
```

```
        }
    }
}
int main() {
    string T, P;
    cin >> T >> P;
    rabin_karp(T, P);
    return 0;
}
```

K Mismatches

```
#include <iostream>
#include <cmath>
using namespace std;
const int d = 256;
void rabin_karp_approximate(string T, string P, int q, int k) {
    int N = T.length(), M = P.length();
    int h = 1, P_hash = 0, T_hash = 0;
    for (int i = 0; i < M - 1; i++)
    h = (h * d) % q;
for (int i = 0; i < M; i++) {
    P_hash = (d * P_hash + P[i]) % q;
    T_hash = (d * T_hash + T[i]) % q;
}
for (int i = 0; i <= N - M; i++) {
    if (P_hash == T_hash) {
        int mismatches = 0;
        for (int j = 0; j < M; j++) {
            if (T[i + j] != P[j])
                mismatches++;
            if (mismatches > k)
                break;
        }
        if (mismatches <= k)
            cout << i << endl;
    }
    if (i < N - M) {
        T_hash = (d * (T_hash - T[i] * h) + T[i + M]) % q;
```

```
            if (T_hash < 0)
                T_hash += q;
        }
    }
}
int main() {
    string T, P;
    int q, k;
    cin >> T >> P >> q >> k;
    rabin_karp_approximate(T, P, q, k);
    return 0;
}
```

LRS

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Suffix {
    int index;

    string suffix;

};

bool cmp(Suffix a, Suffix b) {

    return a.suffix < b.suffix;

}

string lrs(string T) {

    int N = T.length();

    vector<Suffix> suffixes(N);

    for (int i = 0; i < N; i++)

        suffixes[i] = {i, T.substr(i)};

    sort(suffixes.begin(), suffixes.end(), cmp);

    string longest = "";

    for (int i = 0; i < N - 1; i++) {

        string lcp = "";

        int j = 0;

        while (j < suffixes[i].suffix.size() && j < suffixes[i + 1].suffix.size()
&&

               suffixes[i].suffix[j] == suffixes[i + 1].suffix[j]) {

            lcp += suffixes[i].suffix[j];
            j++;
        }
        if (lcp.length() > longest.length())
            longest = lcp;
    }
    return longest;
}
int main() {
    string T;
    cin >> T;
    string result = lrs(T);
    if (result.empty())
        cout << "No repeating substring" << endl;
    else
        cout << result << endl;
    return 0;
}
```