```
// Copyright 2023 Thomas O’Connor
#include "EDistance.hpp"

// Constructor
EDistance::EDistance(const string& lOp, const string& rOp) :
                    _M(lOp.size()+1), _N(rOp.size()+1), _MString(lOp), _NString(rOp) {
    matrix = new int[_M * _N];
    _NString.push_back(’-’); _MString.push_back(’-’);
    // set bounds of matrix
    for (int i = 1; i < _M; i++) {
        matrix[i*_N-1] = 2*(_M-i);
    }
    for (int j = 0; j < _N; j++) {
        matrix[_M*_N-_N+j] = 2*(_N-j-1);
    }
}

// interactor function
int EDistance::min(int a, int b, int c) const {
    int minVal = a;
    if (b < minVal) minVal = b;
    if (c < minVal) minVal = c;
    return minVal;
}

// interactor function
int EDistance::optDistance() {
    // begin on bounds size - 2; 1 for standard bounds and 1 for additional dash character
    for (int i = _M - 2; i >= 0; i--) {
        for (int j = _N - 2; j >= 0; j--) {
            // fill the matrix using the min method
            matrix[i*_N+j] = min(matrix[(i+1)*_N+j+1]+penalty(_MString.at(i), _NString.at(j)),
                                 matrix[(i+1)*_N+j]+2, matrix[(i*_N)+j+1]+2);
        }
    }
    return matrix[0];
}

// interactor function
string EDistance::alignment() {
    // traverse the matrix, collect points, add them to the list
    int i = 0, j = 0;
    while (i < _M-1 || j < _N-1) {
        // if at boundary of matrix:
        if (j == _N-1) {
            optPath.push_back(pair<int, int>(i+1, j));
            i++;
        } else if (i == _M-1) {
            optPath.push_back(pair<int, int>(i, j+1));
            j++;
        // else perform normal checks:
        // diagonal
        } else if (matrix[i*_N+j] == matrix[(i+1)*_N+j+1] +
                    penalty(_MString.at(i), _NString.at(j))) {
            optPath.push_back(pair<int, int>(i+1, j+1));
            i++; j++;
        // down
        } else if (matrix[i*_N+j] == matrix[(i+1)*_N+j] + 2) {
            optPath.push_back(pair<int, int>(i+1, j));
            i++;
        // right
        } else {
            optPath.push_back(pair<int, int>(i, j+1));
            j++;
        }
        // the above order is important because it ensures that diagonals are
        // prioritized unless at a border condition
    }
    // traverse the list, refrence the matrix, assemble string
    string outputString;
    pair<int, int> previousIter(0, 0);
    for (pair<int, int> iter : optPath) {
        // diagonal
        if (previousIter.first == iter.first-1 && previousIter.second == iter.second-1) {
            outputString.push_back(_MString.at(previousIter.first));
            outputString.append(" ");
            outputString.push_back(_NString.at(previousIter.second));
            outputString.append(" ");
            outputString.append(std::to_string(penalty(_MString.at(previousIter.first),
                                _NString.at(previousIter.second))));
            outputString.append("\n");
        // down
        } else if (previousIter.first == iter.first-1 && previousIter.second == iter.second) {
            outputString.push_back(_MString.at(previousIter.first));
            outputString.append(" - 2\n");
        // right
        } else {
            outputString.append("- ");
            outputString.push_back(_NString.at(previousIter.second));
            outputString.append(" 2\n");
        }
        previousIter = iter;
    }
    return outputString;
}

// debug function
void EDistance::printMatrix() {
    cout << "N/M ";
    for (char a : _NString) cout << setw(4) << a;
    cout << endl;
    for (int i = 0; i < _M; i++) {
        cout << setw(3) << _MString.at(i) << " ";
        for (int j = 0; j < _N; j++) {
            cout << setw(4) << matrix[i*_N+j];
        }
        cout << endl;
    }
}
```