```
------------------------------------------------------------------
dpalign.cpp: dynamic programming based string alignment
------------------------------------------------------------------

#include <...>
using namespace std;

#include "matrix.h"

class dpalign {
  public:
    dpalign(int);

    void compute_alignment(string &, string &);
    void print_alignment();

  private:
    int mode;

    string ex;
    string ey;
    int m, n;

    matrix<int> cost;  // edit costs
    matrix<int> link;  // alignment info

    const int VERT;    // 1 -- see below
    const int HORZ;    // 2 -- see below
    const int DIAG;    // 4 -- see below

    int (dpalign::*DEL)(char);        // function ptr
    int (dpalign::*INS)(char);        // function ptr
    int (dpalign::*SUB)(char, char);  // function ptr

    int DELcost(char c) { return (*this.*DEL)(c); }
    int INScost(char c) { return (*this.*INS)(c); }
    int SUBcost(char c1, char c2) {return (*this.*SUB)(c1,c2);}

    // Levenshtein cost function (all edits cost 1)
    int DEL1(char c) { return 1; }
    int INS1(char c) { return 1; }
    int SUB1(char c1, char c2) { return c1==c2 ? 0 : 1; }

    // Longest common subsequence cost function (no subs)
    int DEL2(char c) { return 1; }
    int INS2(char c) { return 1; }
    int SUB2(char c1, char c2) { return c1==c2 ? 0 : m+n; }

    // Difference cost function (data dependent edit costs)
    int DEL3(char c) { return c-'0'; }
    int INS3(char c) { return c-'0'; }
    int SUB3(char c1, char c2) { return abs(c1-c2); }
};

dpalign::dpalign(int n_mode) : VERT(1), HORZ(2), DIAG(4) {
    mode = n_mode;

    switch (mode) {
    case 1:
      DEL = &dpalign::DEL1;
      INS = &dpalign::INS1;
      SUB = &dpalign::SUB1;
      break;

    case 2:
      DEL = &dpalign::DEL2;
      INS = &dpalign::INS2;
      SUB = &dpalign::SUB2;
      break;

    case 3:
      DEL = &dpalign::DEL3;
      INS = &dpalign::INS3;
      SUB = &dpalign::SUB3;
      break;
    }
}

------------------------------------------------------------------
Hint: The edit costs depend on the mode. Function pointers are
used to avoid having to repeatedly check which functions to use.

For deletions, the set up is as follows:

int (dpalign::*DEL)(char);                          function ptr
int DELcost(char c) { return (*this.*DEL)(c); }   use (wrapper)
DEL = &dpalign::DEL1;                               initialization
------------------------------------------------------------------

Hint: dpalign(int n_mode) : VERT(1) { .. } sets VERT when object
is instantiated but before constructor is executed. Being const,
VERT cannot be set within the function itself. Same applies to
HORZ and DIAG.
------------------------------------------------------------------
```

```
void dpalign::compute_alignment(string &x, string &y) {
    ex = "-" + x;
    ey = "-" + y;

    m = x.length();
    n = y.length();

    cost.assign(m+1, n+1);
    link.assign(m+1, n+1);

    cost[0][0] = 0;
    link[0][0] = 0;

    for (int i=1; i<=m; i++) {
      cost[i][0] = cost[i-1][0] + DELcost(ex[i]);
      link[i][0] = VERT;
    }

    for (int j=1; j<=n; j++) {
      cost[0][j] = cost[0][j-1] + INScost(ey[j]);
      link[0][j] = HORZ;
    }

    for (int i=1; i<=m; i++) {
      for (int j=1; j<=n; j++) {
        cost[i][j] = cost[i-1][j-1] + SUBcost(ex[i],ey[j]);
        link[i][j] = DIAG;

        int delcost = cost[i-1][j] + DELcost(ex[i]);
        if (delcost < cost[i][j]) {
          cost[i][j] = delcost;
          link[i][j] = VERT;
        }

        int inscost = cost[i][j-1] + INScost(ey[j]);
        if (inscost < cost[i][j]) {
          cost[i][j] = inscost;
          link[i][j] = HORZ;
        }
      }
    }

    cout << "m = " << m << "\n"
         << "n = " << n << "\n";
    cout << "D[m][n] = " << cost[m][n] << "\n";
}
```

```
void dpalign::print_alignment() {
    stack<char> alignment_x;
    stack<char> alignment_y;

    int i=m, j=n, link_ij=0;
    while ((link_ij=link[i][j]) != 0) {
      if (link_ij == DIAG) {
        alignment_x.push(ex[i]);
        alignment_y.push(ey[j]);
        i = i-1, j = j-1;
      } else if (link_ij == VERT) {
        alignment_x.push(ex[i]);
        alignment_y.push(ey[0]);
        i = i-1;
      } else { // link_ij == HORZ
        alignment_x.push(ex[0]);
        alignment_y.push(ey[j]);
        j = j-1;
      }
    }

    cout << "** ";
    while (!alignment_x.empty()) {
      cout << alignment_x.top();
      alignment_x.pop();
    }
    cout << "\n";

    cout << "** ";
    while (!alignment_y.empty()) {
      cout << alignment_y.top();
      alignment_y.pop();
    }
    cout << "\n\n";
}
```

TODO: Work out how to compute and print ALL optimal alignments.

Hint: Compute_alignment must store multiple equal cost links in one cell. Print_alignment must explore all possibilities in a systematic manner.

```
int main(int argc, char *argv[]) {
  int mode = -1;
  if (argc == 2) mode = atoi(&argv[1][1]);

  if (mode < 1 || 3 < mode) {
    cerr << "usage: " << argv[0]
         << " -1|2|3\n";
    return 0;
  }

  dpalign DPA(mode);
  string x, y;

  while (1) {
    cout << "DPA> ";
    cin >> x >> y;

    if (cin.eof()) break;

    DPA.compute_alignment(x,y);
    DPA.print_alignment();
  }

  cout << "\n";
}
```

```
Option -1 computes the alignment of the strings that requires
the fewest edits (be that substitions, deletions or insertions)

unix> ./dpalign -1
DPA> kajsfh afshss

D = 5

** kajs-fh
** -afshss

The alignment cost of 5 arises from the deletion and insertion
indicated by the dash marks plus the 3 substitutions: j=f, f=s
and h=s.

-----------------------------------------------------------------

Option -2 computes the alignment that contains the LONGEST
COMMON SUBSEQUENCE for the strings. Explored in Lab 8.

unix> ./dpalign -2
DPA> kashfjahs afhjkfa

|lcs| =  4

** ka-sh--fjahs
** -af-hjkf-a--

An alignment cost of 8 arises from the 8 deletions/insertions
indicated by the dash marks.

The longest common subsequence has a length of 4 and consists
of the matching symbols: .a..h..f.a..

-----------------------------------------------------------------

Option -3 produces the optimal alignment based on minimizing
overall symbol differences. Explored in HW12.

unix> ./dpalign -3
DPA> 71264712 76124124

D = 13

** 7126-47-12
** 7--6124124
```