**Pairwise Alignment**

The most commonly occurring genomic mutations substitute a single nucleotide for another. For example, single-nucleotide polymorphisms (or SNPs), account for the majority of intra-species variation; accordingly, SNPs determine most human genetic variation.

Given two nucleotide sequences *S*1 and *S*2 of the same length from members of the same or closely related species, it is easy to calculate the number of SNPs separating them: simply align the strings and count the number of mismatches. The resulting number is called the *Hamming distance* between the two strings, written *d*H(S1, *S*2) (Fig. 1).

However, the Hamming distance model does not account for the fact that insertions and deletions also occur frequently as genomic mutations. In 1965, Vladimir Levenshtein (who was studying error-correcting codes, hardly a biological field at the time) introduced a natural metric that would account for these additional operations. Given strings *S*1 and *S*2, Levenshtein assigned a unit cost to any substitution/insertion/deletion of a symbol. He then asked a question of how to ompute the minimum total cost of any transformation of *S*1 into *S*2 by the three operations. This minimum cost is called the *edit* *distance* between *S*1 and *S*2 and is denoted *d*L(*S*1, *S*2)(Fig. 2a); a transformation with this minimum cost is called *optimal*.

For any transformation of *S*1 into *S*2, note that we can arrange all operations to occur in order with respect to *S*1 (Fig. 2b). As a result, we will be able to represent a Levenshtein transformation of *S*1 into *S*2 by aligning the sequences as shown in Fig. 2c. Substituted characters are still aligned. However, a character insertion in *S*1 will be represented by adding a “-“ to *S*1 opposite from the character in *S*2 scheduled for addition to *S*1; a deletion of a character in *S*1 is represented by adding a “-“ to *S*2 opposite from the character of *S*1 scheduled for deletion from *S*1 (Fig. 2c). The alignment results in *extended strings* S1’ and *S*2’ having the same length. The extended string of S contains all symbols of *S*i’ in order interspersed with space symbols. The cost of the associated transformation is just the number of mismatches between *S*1’ and *S*2’, or *d*H(S1’, *S*2’).

Note that when building an alignment from left to right, we have only three choices at each step:

1. Align the next two characters from *S*1 and *S*2 (substitution)
2. Align the next character from *S*2 with an inserted space in *S*1 (insertion)
3. Align the next character from *S*1 with an inserted space in *S*2 (deletion)

**PROBLEM**: Calculate *d*L(*S*1, *S*2) for the nucleotide strings *S*1 and *S*2 given in the associated file.[[1]](#footnote-1) Also, provide a 2-row matrix representing an optimal alignment whose rows are the extended strings corresponding to this alignment (i.e., row 1 is *S*1’ and row 2 is *S*2’).

**GAGCCTACTAACGGGAT**

**CATCGTAATGACGGCCT**

**Fig. 1**: The Hamming distance between two nucleotide sequences above is simply the total number of mismatches between them (highlighted in red), or 7.

|  |  |
| --- | --- |
| (a) **Chimpanzee** **Chompanzee** **Chompanzees** **Chompaniees** **Chompaniees** **hompsaniees** **homosaniees** **homosapiees** **homosapiens** | (b) **Chimpanzee** **Chimpanzee** **hompanzee** **homoanzee** **homosanzee** **homosapzee** **homosapiee** **homosapien** **homosapiens** |
| (c)**Chimp-anze-e****-homosapiens**  |

**Fig. 2**: (a) We may transform “chimpanzee” into “homosapiens” with just 8 character substitutions (red), insertions (blue), and deletions (green). But how do we know if this transformation is optimal? (b) A first step toward simplifying the problem is to order operations according to where they occur in the changing sequence. (c) Better yet, we may form an alignment of the extended strings defined by the preceding transformation, where “-“ (space sign) represent insertion or deletion of character. The Hamming distance between these extended strings is 8. Does this present an optimal transformation of “chimpanzee” into “homosapiens”?

**Draft of expected solution:**

An efficient solution is produced from a dynamic programming layout with the following formulas:

dist(i,0) = dist(0,i) = i

 dist(i-1,j)+1

dist(i,j) = min dist(i,j-1)+1

 dist(i-1,j-1) + (0 if match else 1)

parent(i,0) = (i-1,0)

parent(0,j) = (0,j-1)

 dist(i-1,j)+1

parent(i,j) = argmin(i,j) dist(i,j-1)+1

 dist(i-1,j-1) + (0 if match else 1)

Alignment reconstruction can be performed recursively by following parent links.

**Draft of expected checking process:**

To check a student’s solution, we must run the (master) solution first on our randomly generated strings to obtain the edit distance.

The user's solution is then considered correct iff:

1. The value for the edit distance is correct.
2. The alignment consists of extended strings of the input data whose total number of mismatches is equal to the value given for the edit distance in (1.).

**Dataset:** A dataset can be generated randomly by the following function in Python:

import random

import string

# Random sequence generator – two sequences of length N and M

def generate(seed, N, M):

 random.seed(seed)

 print ''.join([random.choice(string.letters) for x in range(N)])

 print ''.join([random.choice(string.letters) for x in range(M)])

A master solution in C++ is attached on the following two pages.

// find Levenshtein distance and alignment matrix

#include <string>

#include <iostream>

#include <vector>

#include <algorithm>

#include <cassert>

using namespace std;

enum direction {UP, LEFT, UPLEFT}; // parental directions

int main() {

 string s1, s2;

 cin >> s1 >> s2;

 vector<vector<int>> dist(s1.size() + 1, vector<int>(s2.size() + 1));

 vector<vector<direction>> par(s1.size() + 1, vector<direction>(s2.size() + 1));

 for (size\_t i = 0; i <= s1.size(); ++i) {

 dist[i][0] = i;

 par[i][0] = up\_;

 }

 for (size\_t j = 0; j <= s2.size(); ++j) {

 dist[0][j] = j;

 par[0][j] = left\_;

 }

 for (size\_t i = 1; i <= s1.size(); ++i) {

 for (size\_t j = 1; j <= s2.size(); ++j) {

 int ins = dist[i-1][j] + 1;

 int del = dist[i][j-1] + 1;

 int match = dist[i-1][j-1] + (s1[i-1] == s2[j-1] ? 0 : 1);

 if (ins < del && ins < match) {

 dist[i][j] = ins;

 par[i][j] = UP;

 }

 else if (del < match) {

 dist[i][j] = del;

 par[i][j] = LEFT;

 }

 else {

 dist[i][j] = match;

 par[i][j] = UPLEFT;

 }

 }

 }

 // output distance

 cout << dist[s1.size()][s2.size()] << endl;

 // output alignment matrix

 string res1, res2;

 int i = s1.size();

 int j = s2.size();

 while (i != 0 || j != 0) {

 if (par[i][j] == UP) {

 i -= 1;

 res1 += s1[i];

 res2 += '-';

 }

 else if (par[i][j] == LEFT) {

 j -= 1;

 res1 += '-';

 res2 += s2[j];

 }

 else {

 assert(par[i][j] == UPLEFT);

 i -= 1;

 j -= 1;

 res1 += s1[i];

 res2 += s2[j];

 }

 }

 reverse(res1.begin(), res1.end());

 reverse(res2.begin(), res2.end());

 cout << res1 << endl;

 cout << res2 << endl;

return 0;

}

1. A particular dataset for this problem is not yet given, as any random nucleotide string of sufficient length to prevent a brute-force approach will suffice. For this reason, we provide a program to randomly generate strings. [↑](#footnote-ref-1)