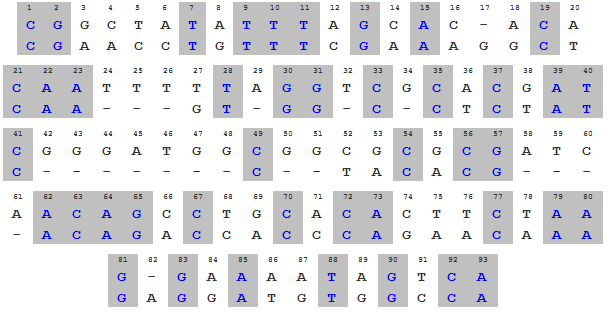
|  |
| --- |
| Ecole polytechnique |
| From ADN to formation of proteins: how to align sequences? |
| *Programming Project INF421* |
|  |
| **Jenny Mansour and Manon Romain** |
| **06/01/2017** |



|  |
| --- |
|  |

Table of contents

[I. Auxiliary considerations 3](#_Toc471561223)

[Recursive versus Iterative Implementation 3](#_Toc471561224)

[How to Read and Print Files? 3](#_Toc471561225)

[II. Longest common subsequences and edition distance 3](#_Toc471561226)

[Task 1. Naïve algorithm 3](#_Toc471561227)

[Task 2. Longest common subsequence 4](#_Toc471561228)

[Task 3. Optimal Alignment 4](#_Toc471561229)

[III. Refinements on alignment 5](#_Toc471561230)

[Task 4.  Substitution matrices 5](#_Toc471561231)

[Task 5. Affine gap penalty 7](#_Toc471561232)

[Task 6. Local Alignment 8](#_Toc471561233)

[IV. Basic Local Alignment Search Tool 8](#_Toc471561234)

[Task 7. Beginning Perfect Matches 8](#_Toc471561235)

[Task 8. BLAST 9](#_Toc471561236)

# Auxiliary considerations

## Recursive versus Iterative Implementation

At first, we did the implementation of all tasks using recursive algorithms, as it seemed more natural to use. However, when testing on the (long) test files provided our programs were raising a Stack Overflow error, even when we were just using tail-call. Doing a bit of research, we learnt that Java did not support the tail call optimization - that transforms tail call into loop when compiling the code - so we had to resort to using iterative algorithms for all our programs.

## How to Read and Print Files?

In the package InputOutput, we implemented two classes, ReadFile and PrintOptimalAlignment.

* ReadFile class needs a String with the name of a file (existing at the root of the project) to be constructed. This file should contain two lines: the first one represents the first sequence and the second one the second sequence. This function thus separates those two lines and saves them in two different class variables first\_line and second\_line.
* PrintOptimalAlignment takes in arguments a list of Booleans and two lists of characters that give the list of letters and hyphens of best alignment for the task considered. Thus, this function prints a window with the two sequences, one above the other, with a special color when the letters match in the alignment.

# Longest common subsequences and edition distance

## Task 1. Naïve algorithm

Let L1 be the first sequence, with n1 its length

Let L2 be the second sequence, with n2 its length.

Let Sub1 and Sub2 be arrays of length 2n1 and 2n2.

Add all subsequences possible for L1 and L2 in Sub1 and Sub2, ordered by decreasing length of the subsequences.

For an integer i going from 0 to 2n1:

For an integer j going from 0 to 2n2:

If Sub1[i] and Sub2[j] have the same length, compare the two strings :

If they are identical, call it *subsequence* and return *subsequence*

Return *subsequence* being the empty list

**Complexity**

The two loops have a 2n1+n2 complexity. In each loop, we compare two strings of length an integer between 0 and max (n1, n2).

Hence, the complexity of this naïve algorithm is more than 2n1+n2.  
It is in the range of S\*2n1+n2, S being the sum from 1 to max (n1, n2) of all integers.

Now let’s improve this complexity with dynamic programming. For that, we tried - in task 2 - to compute a matrix with the score of alignments. This matrix permits to the function computeOptimalAlignmentto solve our problem by solving sub-problems and from their solutions obtain the original solution.

## Task 2. Longest common subsequence

**Equivalence** To go from a word *s* to a word *t*, there is a minimal number of operations, knowing that the possible operations are inserting, deleting or transforming a letter. Inserting a letter in a list means adding a hyphen at the same position in the other list (to keep the letters of the common subsequence above each other). In addition, for the same reason, deleting a letter in a list adds a hyphen in the other one. Eventually, transforming a letter means having two letters above each other that are different and thus that do not belong in the common subsequence.

Thus, having a minimal number of those three operations is equivalent to having a minimal sum of hyphens (inserting/deleting) or different letters in both lists (transforming).

**Input** A file in which line one is the first sequence s1 (length n1) and line two the second one s2 (length n2).

**Output** None - A window is printed representing the subsequence that scores best using the Blosum50 matrix

**Algorithm**

See . Here we just compute the subsequence (without '-' when necessary).

## Task 3. Optimal Alignment

**Input** A file in which line one is the first sequence s1 (length n1) and line two the second one s2 (length n2).

**Output** None - A window is printed representing the subsequence that scores best using the Blosum50 matrix

**Algorithm**

The algorithm is the same than in , the only difference is that instead of the score that's in the Blosum50 matrix, the score is 1 if the letters are the same and 0 if we need to replace, insert or delete a letter.

# Refinements on alignment

We used very similar functions in all tasks:

* In tasks 4, 5 and 6, computeAlignmentMatrix computes two matrices : the first matrix contains the score of transformations – the score of transforming the sequence s1[0 … i-1] into s2[0 … j-1] is in the coefficient (i,j) – and the second one contains the last operation (*e* for equality, *r* for replacement, *d* for deletion and *i* for insertion)for this transformation.
* computeOptimalAlignmenttraces back the optimal alignment – in our case, two linked lists of characters (letters or "-") and a linked list of Booleans indicating whether the letters match – from the matrix computed before.
* OptimalAlignmentinitializes both matrices, uses the two functions described above, and prints – nicely – the optimal alignment.

## Task 4.  Substitution matrices

**Input** A file in which line one is the first sequence s1 (length n1) and line two the second one s2 (length n2).

**Output** None - A window is printed representing the subsequence that scores best using the Blosum50 matrix.

**Algorithm**

Let M be a empty square matrix of size (n1+1)\*(n2+1), each element will contain the score of the best alignment between s1[1…i] and s2[1…j].

Let Op be a empty square matrix of size (n1+1)\*(n2+1), each element will contain the last operation to perform to transform s1[1…i] into s2[1…j].

Implementation: char[][] (*e* for equality, *r* for replacement, *d* for deletion and *i* for insertion).

*Initialization:*

We initialize the first line and first column of our matrix: if we try to compute the best alignment between an empty string and a string s, the score will be the and the operation will be deleting (if s is the first sequence) or inserting (if it is the second one) all letters.

*Recurrence:*

We fill both matrix using the following recurrence formula:

Implementation: two for-loops.

Let's explain this recursive formula. When computing the best score of an alignment between s1[1…i] and s2[1…j], we have three options to consider:

* We can align ai and bj and find the best alignment s1[1…i-1] and s2[1…j-1], Then, we need to replace 'r' ai by bj (we write 'e' for equality if ai  and bj are the same just to color them properly when printing).

|  |  |  |
| --- | --- | --- |
|  |  | **Total score =** |
|  |

* We can delete ai and find the best alignment s1[1…i-1] and s2[1…j], Then, we need to delete 'd' ai .

|  |  |  |
| --- | --- | --- |
|  |  | **Total score =** |
|  |

* We can insert bj at position I and find the best alignment s1[1…i] and s2[1…j-1], Then, we need to insert 'i'.

|  |  |  |
| --- | --- | --- |
|  |  | **Total score =** |
|  |

When we are finished filling both M and Op, we need to backtrack to computes to best alignment. We store it in two list of characters (letters or '-') and one list of Booleans stating whether the letters are matching (we will color them later).

Implementation: LinkedList<Character> and LinkedList<Boolean>

Starting from the lengths of both strings in Op, and while i and j are not both equal to zero, if Opi, j is:

* 'r' or 'e' we add ai and bj to the appropriate lists and decrease i and j.
* 'i' we add '-'and bj to the appropriate lists and decrease j.
* 'd' we add ai and '-'to the appropriate lists and decrease i.

**Complexity**

As we fill the each element of the matrices, the complexity of computeAlignmentMatrix is

The complexity of the backtrack function is because we start from, and at each step decreases by one or two.

Therefore, the final time complexity is

## Task 5. Affine gap penalty

**Input** A file in which line one is the first sequence s1 (length n1) and line two the second one s2 (length n2).

**Output** None - A window is printed representing the subsequence that scores best using the Blosum50 matrix and affine gap penalty

**Algorithm**

The algorithm is very similar to the one in , but the recurrence formula is now:

Besides, we initialize the first line and column to 0 because we are told gap at the beginning doesn't count.

Let's explain the difference:

* If is the letter that ends a k-gap in the second sequence, the score is as described below. We will maximize the score of such alignment over all possible gaps (k going from 1 to i). Besides, ifis the last letter of the first word, the gap doesn't count. If the maximum is found there, the corresponding operation will be deleting 'd'.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | **Total score =** |
|  |  |  |

**Complexity**

Each step of the loop does operations, so according to the calculation below, the final complexity will be .

## 

## Task 6. Local Alignment

**Input** A file in which line one is the first sequence s1 (length n1) and line two the second one s2 (length n2).

**Output** None - A window is printed representing the local subsequence (not the whole sequence) that scores best using the Blosum50 matrix and affine gap penalty

**Algorithm**

The algorithm is the same as before (see), expect that we just want the optimal local alignment, so if the alignment between s1 and s2 has a negative score we don't select it.

The backtrack function is also different: it starts from the global maximum of the score matrix and goes up until it meets a zero, then it will not consider the beginning or end of the sequences that don't improve the local score.

# Basic Local Alignment Search Tool

## Task 7. Beginning Perfect Matches

**Input** A file in which line one is the new sequence g (length n) and line two the second one t(length m).

**Output:** None - Prints indices in the second sequence that correspond to beginning of perfect matches between an element of Sg and a subword of t.

Given the new sequence s, the database sequence t, the threshold *th* and an integer k, task7 computes the list of indices that are beginning of perfect k-matches between a seed and a k-subword of t. The seeds are given by the function findSeeds*.*

To perform this, we need some auxiliary functions:

* computeScorecomputes the score between two strings using the Blosum50 matrix.
* *breakInto* takes an integer k and string s and computes the stack of all possible k-subwords from s (a k-subword being k consecutive letters) by adding them with a loop in the stack. We choose to put the k first letters on top of the stack.
* *computeAllWords* is a function that takes an integer k, computes from the alphabet all possible k-words, and puts them in a stack. In order to o do that, this function calls an auxiliary function:
* *computeAllWordsAux* takes in argument the length of the words we are looking for, the alphabet (containing all letters of Blosum50 matrix alphabet), the current word we are building and the Stack of possible words we already computed. The function adds all possible letters at the end of the word if it is not long enough and otherwise stops when the current word has the correct length.

findSeedstakes in argument k, s and the threshold *th*. It returns in a Map (Integer → String) all k words that have a score with a k-subword of s that is superior to *th* times the score of the subword with itself*.* When a seed is found, we store it in the Map its key being the index of the corresponding subword in *s*.

The function task7 then creates a binary search tree, which contains all seeds, and computes all the k-subwords of t, with the function breakInto*.*

**Optimization of the complexity**

To optimize the complexity, we decided to store seeds in a Binary Search Tree. This implementation permits to search elements in logarithmic complexity, which is much better than what we had previously.

## Task 8. BLAST

**Input** A file in which line one is the new sequence g (length n) and line two the second one t(length m).

**Output** None - Prints all local alignments with sufficiently high scores

Task 8 is computed through findAlignments*.*

findAlignmentsimplements the BLAST algorithm. It takes in argument the length k of the subwords used, the new sequence *s*, the database sequence *t*, and the threshold *th* and *thl* used in the algorithm.

* First, the function finds the seeds of the new sequence and creates the Binary Search Tree (that scores seeds and their indexes in the sequence), through findSeeds.
* Then, it computes the k-subwords of the database sequence.
* It searches for k-words in the database matching our seeds. When it does, it stores the begin indexes of the match in both sequences in the Map IndexesPerfectMatches.
* Eventually, it tries to extend the alignments, by storing them in a list of IndexesLengthScore and comparing their score to the score of g with itself. The current match is the alignment of a seed and a k-subword of t and we try to extend backward and forward. If it has a sufficient score and hadn’t already been computed, we store it.

**N.B:** IndexesLengthScore is a new Data Structure that stores alignments as wanted; it stores the index of the first letter of the alignment in both sequences, its length and score (in four separate class variables respectively: indexInS, indexInT, length and score).

For this step, two auxiliary functions are needed: frontand back.

* fronttakes two strings s and t and a current local match between them and updates curr\_match until it reaches its maximal length (just from left to right).Until it reaches the end of one of the sequences, it continues and if we benefit from adding a letter, it does it.
* backdoes exactly the same thing but from right to left.