

## Algorithms in Bioinformatics - Sequences: Implementation of global and local alignment

---

### Finding global and local alignments in quadratic space

To implement the recursive functions for finding a global or local alignment, we choose to use Python, as this has proven effective for these tasks. Both the global and local alignments were implemented by first creating the dynamic programming tables (both for calculation of the scores and traceback), iterating through every cell (row-wise) and finally, if desired, the optimal alignment is retrieved by walking through the traceback table. The implementation of global and local alignments were made by first making the global alignment and then altering a copy where necessary, so the copy would perform a local alignment. These differences are shown in the table below.

**Comparison of global vs. local alignments**

	<b>Global</b>	<b>Local</b>
Initial conditions	$S(i, 0) = i \cdot \text{gapcost}$ $S(0, j) = j \cdot \text{gapcost}$	$S(i, 0) = S(0, j) = 0$
Recursive function	$S(i, j) = \max \begin{cases} S(i-1, j-1) + \text{Sub} \\ S(i-1, j) + \text{gapcost} \\ S(i, j-1) + \text{gapcost} \end{cases}$	$S(i, j) = \max \begin{cases} S(i-1, j-1) + \text{Sub} \\ S(i-1, j) + \text{gapcost} \\ S(i, j-1) + \text{gapcost} \\ 0 \end{cases}$
Score for alignment and starting point for traceback	S(n,m)	Maximum found score

### Finding the alignments in linear space

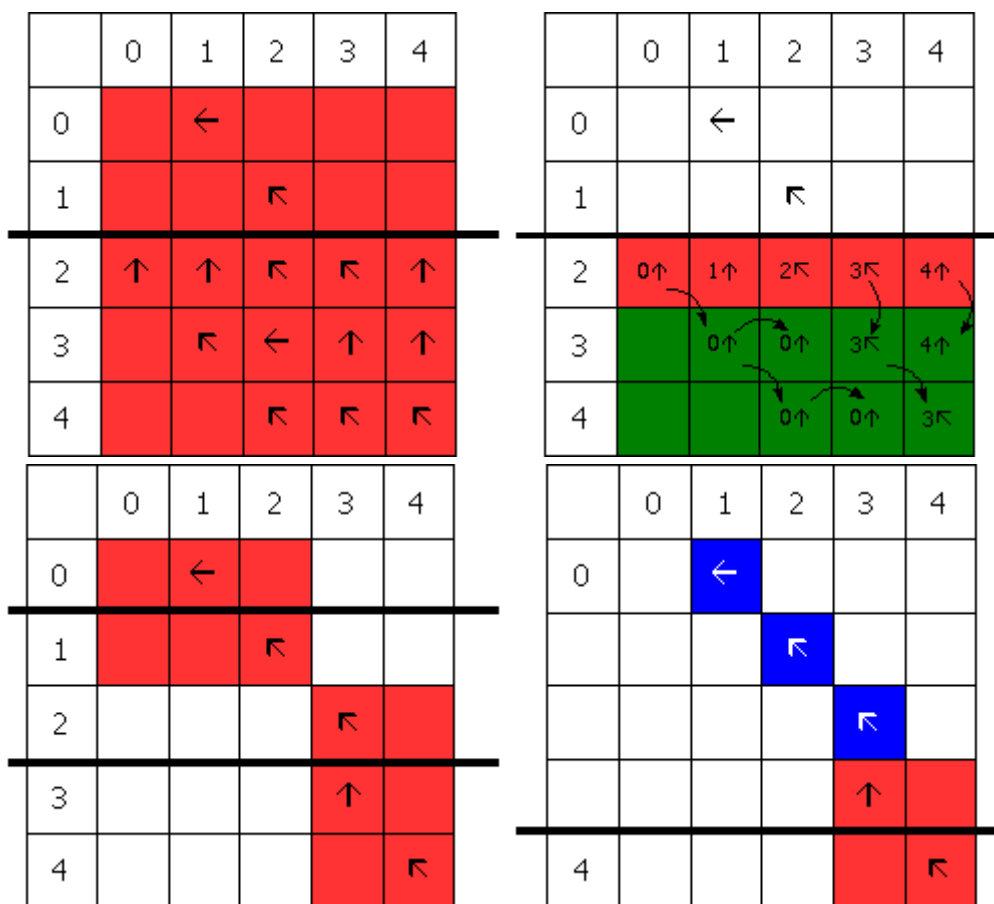
When it is not necessary to find the optimal alignment, the score can be found without creating the entire dynamic programming. This can be done, because to calculate the value of any given cell, it is only necessary to know the value of the cell to the left, above and to the left of this. With this in mind, the score can be found by using two rows (or columns) and switching them.

However, if it is required to find the optimal alignment, this approach does not suffice, as the traceback requires the entire table. In this project, we used Hirschberg idea, which is a divide-and-conquer approach, where the division is found with the method described by Christian Storm Pedersen.

A method for finding the middle edge of a dynamic programming table (a table as described earlier) is defined. This method returns the table column number where the path for the best scoring alignment crosses the middle row. In the first iteration it runs over the entire dynamic programming table. When the middle point is found the same method is run on each of the subproblems. The subproblems being the area of the dynamic programming table which is

above to the left of the middle point and below to the right. The two other "corners" of the table are omitted because it is known that the path does not go into these areas. This is continued until some base cases are reached. At this point all the "middel points" are found and can be concatenated as the two alignments.

In the figures below the iterations are shown. First figure on the left, shows which edge we want to divide the subproblems by (row 2) and the path of the optimal alignment. The figure to the right, shows how the pointers (number+arrow) are brought forward from row 2 (curved arrows) according to the path of the optimal alignment. The alignment is then broken into two subproblems (A[0..1] by B[0..3] and A[2..4] by B[3..4]), shown by the area coloured red in the third figure. Because we now know, which regions the optimal alignment is not in, these regions are left out (coloured white). The fourth figure shows how the subproblems may be divided into further subproblems, or if a base case is found, the alignment is found (coloured blue).



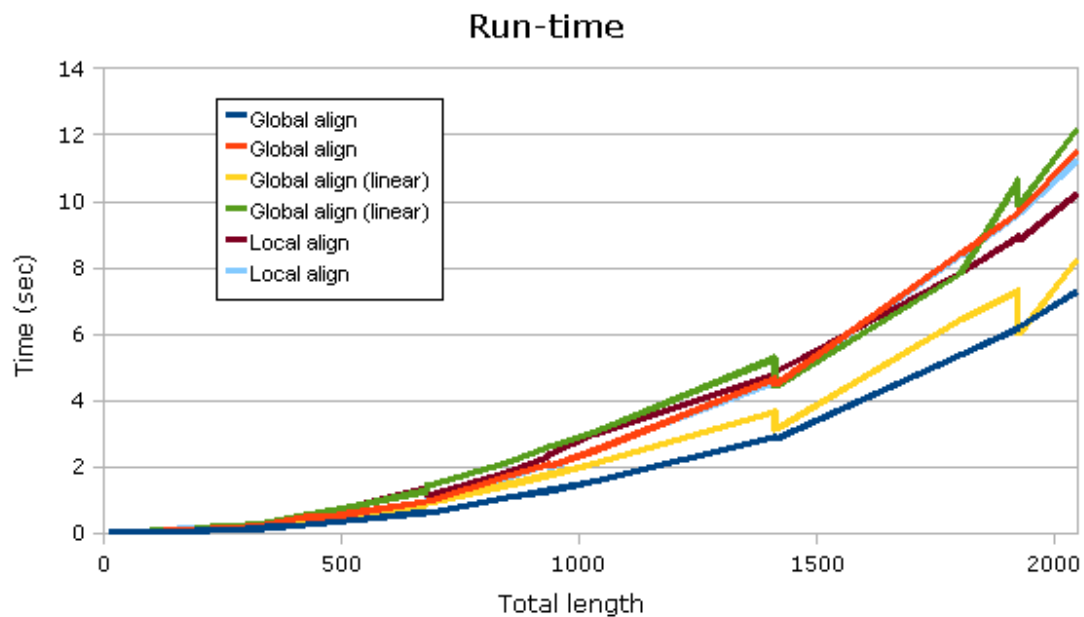
The traceback function ends up running in quadratic time and using linear space.

When in theory it is a simple algorithm to implement, it becomes a bit tough because the indices require some effort to keep track of and defining the base cases are a bit tricky. We have found two:

1. If either substring is empty, simply align the non-empty substring against gaps.
2. If one of the substrings are of length 1, use a "simple" alignment to determine where the single symbol is placed with regard to the longer substring.

## Experiments of time and space usage

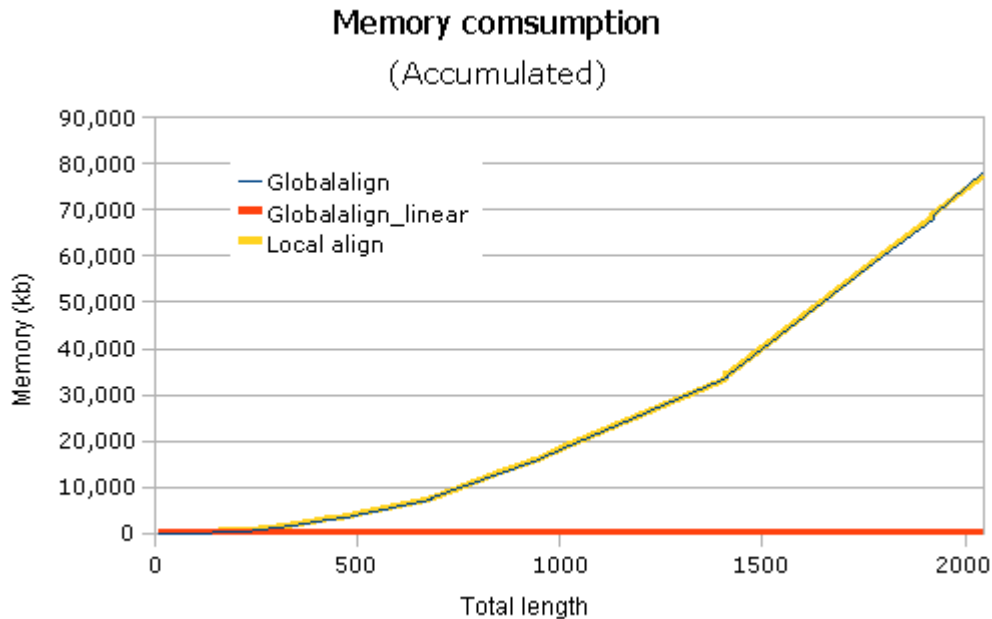
To verify the running time and space usage, these were investigated with various lengths of random generated strings. For the time usage, the python command clock() was called prior and following the alignment. This experiment was performed both via PuTTY connected to fresh-horse.daimi.au.dk and on a laptop running Windows, with total lengths varying from 12 to 2048 characters. The results are shown below:



Results of time usage under varying string lengths. The slowest of each performance is from the laptop.

These results resembles the expected quadratic time.

For the memory usage, the estimation was problematic, as Python itself does not provide methods to estimate the memory usage. For estimation in Windows, we have yet to find a method that does not require installation of additional software or packages. For Unix, a crude method was found, utilising the "ps"-command.



Results of memory usage under varying string lengths.

The results show a vast difference of memory usage between linear and quadratic space consumption methods. This was also expected. However we did not anticipate the difference to be so profound.

In summary, the results have shown us that the linear space methods are on par with the quadratic space methods with regard to speed and outperforms them in memory usage. No reason whatsoever to use quadratic space methods.

## Local alignment in linear space

Changing the local alignment trace back method to run in linear space ought not to be too difficult. When the dynamic programming table is filled (in linear space) the maximum score and corresponding  $i, j$  are remembered as the ending point of the local alignment. Then the strings ( $A[0\dots i]$ ,  $B[0\dots j]$ ) are reversed and a new dynamic programming table is filled until a new maximum score is encountered. This is the starting point of the local alignment, and truncating the strings between the starting and ending point, a global alignment can be performed on the truncated string, which runs in linear space.

## A short user manual

The software package consists of several files:

- `globalalign.py*`: Function `globalalign(...)` - the regular, quadratic space global alignment.
- `globalalign_linear.py*`: Function `globalalign_linear(...)` - global alignment using linear space.
- `localalign.py*`: Function `localalign.py(..)` - local alignment using quadratic space.

- `utils.py`: Utility package; the three above files requires this file. Contains functions for parsing fasta and phylip-files.
- `dna.phy`: Example of scoring of DNA in phylip-format.

Files marked with asterisk (\*) can be called from commandline.

## Commandline use

`globalalign.py`, `globalalign_linear.py` and `localalign.py` can be used directly from commandline, all using the syntax, ex.:

```
python localalign.py -1<str1> -2<str2> -m<filename> -g<gapcost> [-t -f<filename>]
```

where

- `str1` & `str2`: Strings to be aligned. If `str1` or `str2` are filenames, these will be read as fasta-files.
- `-m`: Phylip-file containing distance-matrix for sub-costs.
- `-g`: Gapcost
- `-t`: Disable traceback, optional.
- `-f`: Output sequences to a fasta-file; disables output to screen.
- `--help`: Outputs help text.
- `--demo`: Demonstration of program.

If `--demo` is applied, all other options are ignored and an alignment of `ATTGGGCGCTGG` and `CGGCGCA` is performed with gapcost value of `-2` and a distance matrix as shown below.

	A	C	T	G
A	10	2	2	5
C	2	10	5	2
T	2	5	10	2
G	5	2	2	10

After calculating the score and alignment, the results are outputted to the terminal (except alignment, if this was chosen to be saved in a fasta-file).

## Input methods

As it is not desirable to type long sequences directly into the command prompt, it is possible to give a filename instead. The only supported format is the fasta-format, and in case of multiple sequences per file, only the first sequence is used.

The distance matrix can only be passed as a reference to a file, where the matrix is given in phylip format. The parser requires the first line of the file is the length of the alphabet (this line is in fact just ignored!), while the remaining files are read as a space-delimited table.

Example of a phylip-file with the above distance matrix:

```
4
A 10 2 5 2
C 2 10 2 5
G 5 2 10 2
T 2 5 2 10
```

## Function calls

If already running in Python, the files `globalalign.py`, `globalalign_linear.py` and `localalign.py` can be imported. The main functions have same name as the filename (notice that all three files also contains a function called "main", but this is responsible for the commandline evaluation). The functions have the same arguments, ex.:

```
localalign(A, B, subcost, gapcost[,backtrack=True]):
```

where

- *A* and *B* are the strings to be aligned.
- *subcost* is the distancematrix, as double-dictionary, eg. `subcost = {'A': {'A': 10, 'T': 2, ...}, 'T': {'A': 2, ...}, ...}`  
Please note, that the subcost is case-sensitive and does not support wildcards!
- *gapcost* a integer or float representing the gap cost.
- *backtrack*, optional, can be disabled if only the score is required. In the quadratic space methods, the traceback table is no initialised.

The functions returns the score and alignment as a tuple, eg. `(50, ('ATCG', 'A-CG'))`.