GENIOMHE

Sequence algorithms

by Samuel Ortion

Prof.: Fariza Tahi

2024

Contents

Motifs algorithms

1 Motif

1.1. Searching a substring in a string

Algorithm 1 Brute-force search of a motif in a sequence

```
function FindMotif($S$: Array($n$), $M$: Array($m$))
    returns a list of position
    $pos \leftarrow \{\}$
    $i \leftarrow 0$
    while $i < n - m + 1$ do
        $j \leftarrow 0$
        while $j < m$ and $S[i + j] = M[j]$ do
            $j{+}{+}$
        end while
        if $j = m$ then
            $pos \leftarrow pos \cup \{i\}$
        end if
        $i{+}{+}$
    end while
    return $pos$
end function
```

1.2. Using matrices to search motifs

Let S_1 and S_2 be two sequences.

$S_1 = \text{ACGUUCC}$ $S_2 = \text{GUU}$

Let $n = |S_1|$, $m = |S_2|$ The complexity of this algorithm is $\mathcal{O}(n \cdot m)$ to build the matrix, and it requires also to find the diagonals and thus it is a bit less efficient than the Algorithm 1.

To find repetitions, we can use a comparison matrix with a single sequence against itself. A repetition would appear as a diagonal of ones, not on the main diagonal.

Let $S = \text{ACGUUACGUU}$. Let's write the comparison matrix.

Algorithm 2 Knuth-Morris-Pratt algorithm

```
function KMP_Search($S$: Array($n$), $M$: Array($m$))
    returns Integer
    $table \leftarrow$ KMP_Table($M$)
    $c \leftarrow 0$                                    ▷ Count the number of matches
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ do
        if $S[i] = M[i]$ then
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
        end if
        if $j = m$ then
            $c \leftarrow c + 1$
            $j \leftarrow table[j - 1]$
        else if $j < n$ and $M[j] \neq S[i]$ then
            if $j \neq 0$ then
                $j \leftarrow table[j - 1]$
            else
                $i \leftarrow i + 1$
            end if
        end if
    end while
    return $c$
end function
function KMP_Table(M: Array(m))
    Returns Array(m)
    $previous \leftarrow 0$
    $table \leftarrow$ array of zeros of size m
    for $i = 0; i < m; i++$ do
        if $M[i] = M[previous]$ then
            $previous \leftarrow previous + 1$
            $table[i] \leftarrow previous$
            $i \leftarrow i + 1$
        else
            if $previous = 0$ then
                $previous \leftarrow table[previous - 1]$
            else
                $table[i] \leftarrow 0$
                $i \leftarrow 1$
            end if
        end if
    end for
end function
```

	A	C	G	U	U	C	C
G	0	0	1	0	0	0	0
U	0	0	0	1	1	0	0
U	0	0	0	1	1	0	0

Table 1.1. Comparison matrix

	A	C	G	U	U	A	C	G	U	U	G	U	U
A	1	0	0	0	0	1	0	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0	0
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1
A	1	0	0	0	0	1	0	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0	0
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1
G	0	0	1	0	0	0	0	1	0	0	1	0	0
U	0	0	0	1	1	0	0	0	1	1	0	1	1
U	0	0	0	1	1	0	0	0	1	1	0	1	1

Algorithm 3 Construct a comparison matrix

```
function ComparisonMatrix($S$: Array($n$))
    $M \leftarrow$ Array($n$, $n$)
    for ($i = 0$; $i < n$; $i++$) do
        for $j = 0$; $j < n$; $j++$ do
            if $S[i] = S[j]$ then
                $M[i][j] = 1$
            else
                $M[i][j] = 0$
            end if
        end for
    end for
    return $M$
end function
```

Algorithm 4 Construct the top half of a comparison matrix

```
function ComparisonMatrix($S$: Array($n$))
    $M \leftarrow$ Array($n$,$n$)
    for ($i = 0$; $i < n$; $i++$) do
        for j=i; j < n; j++ do
            if S[i] = S[j] then
                M[i][j] = 1
            else
                M[i][j] = 0
            end if
        end for
    end for
    return M
end function
```

Algorithm 5 Find repetitions (with a set of visited segments)

```
function FindRepetions($S$: Array($n$))
    returns A list of start and end positions for repeated sequences
    $M$ = ComparisonMatrix(S)
    $pos = \{\}$
    $visited = \{\}$
    for ($i_{start} = 0$; $i_{start} < n$; $i_{start}++$) do
        for ($j_{start} = i_{start} + 1$; $j_{start} < n$; $j_{start}++$) do
            if $M[i_{start}][j_{start}] = 1$ and $(i_{start}, j_{start}) \notin visited$ then
                $i = i_{start}$
                $j = j_{start}$
                while $M[i][j] = 1$ do
                    $i++$
                    $j++$
                    $visited = visited \cup \{(i, j)\}$
                end while
                $pos = pos \cup \{(i_{start}, i), (j_{start}, j)\}$
            end if
        end for
    end for
end function
```

Algorithm 6 Find repetitions with an exploration of diagonals

```
function FindRepetions($S$: Array($n$))
    returns A list of start and end positions for repeted sequences
    $M$ = ComparisonMatrix(S)
    $pos = \{\}$
    for ($diag = 1$; $diag < n$; $diag++$) do
        $j = diag$
        $i = 0$
        while $i < n$ and $j < n$ do
            if $M[i][j] = 1$ then
                $i_{start} = i$
                $j_{start} = j$
                while $i < n$ and $j < n$ and $M[i][j] = 1$ do
                    i++
                    j++
                end while
                $pos = pos \cup \{((i_{start}, i-1), (j_{start}, j-1))\}$
            end if
            $i++$
            $j++$

        end while
    end for
end function
```

1.3. Automata

An automaton is a tuple $\langle S, s_0, T, \Sigma, f \rangle$

- S the set of states
- s_0 the initial state
- T the set of terminal states
- Σ the alphabet
- f the transition function $f : (s_1, c) \to s_2$

Example Given the language L on the alphabet $\Sigma = \{A, C, T\}$, $L = \{A^*, CTT, CA^*\}$

Definition 1: Deterministic automaton

An automaton is deterministic, if for each couple $(p, a) \in S \times \Sigma$ it exists at most a state q such as $f(p, q) = q$

Definition 2: Complete automaton

An automaton is complete, if for each couple $(p, a) \in S \times \Sigma$ it exists at least a state q such as $f(p, q) = q$.

Algorithm 7 Find repetitions with an exploration of diagonals, without nested while

```
function FindRepetions($S$: Array($n$))
    returns A list of start positions for repeted sequences and match length
    $M$ = ComparisonMatrix(S)
    $pos = \{\}$
    for ($diag = 1$; $diag < n$; $diag++$) do
        $j = diag$
        $i = 0$
        $l = 0$
        while $i < n$ and $j < n$ do
            if $M[i][j] = 1$ then
                $l++$
            else
                if $l > 0$ then
                    $pos = pos \cup \{(i - l, j - l, l)\}$
                    $l = 0$
                end if
            end if
            $i++$
            $j++$
        end while
        if $l > 0$ then
            $pos = pos \cup \{((i - l, j - l, l))\}$
        end if
    end for
    return $pos$
end function
```

Algorithm 8 Find repetitions

```
function FindRepetions($S$: Array($n$))
    returns A list of start and end positions for repeted sequences
    $M$ = ComparisonMatrix(S)
    $pos = \{\}$
    for $i_{start} = 0$; $i_{start} < n$; $i_{start}++$ do
        for $j_{start} = i_{start} + 1$; $j_{start} < n$; $j_{start}++$ do
            if $M[i_{start}][j_{start}] = 1$ then
                $i = i_{start}$
                $j = j_{start}$
                while $M[i][j] = 1$ do
                    $M[i][j] = 0$                ▷ Ensure that the segment is not explored again
                    $i++$
                    $j++$
                end while
                $pos = pos \cup \{((i_{start}, i - 1), (j_{start}, j - 1))\}$
            end if
        end for
    end for
end function
```

Algorithm 9 Check wether a word belong to a language for which we have an automaton

```
function WordInLanguage($W$: Array($n$), $A$: $\langle S, s_0, T, \Sigma, f\rangle$)
    returns A Boolean valued to true if the word is recognized by the language automaton
    $s \leftarrow s_0$
    $i \leftarrow 0$
    while $i < n$ do
        $a \leftarrow W[i]$
        if $\exists f(s, a)$ then
            $s \leftarrow f(s, a)$
        else
            return false
        end if
        i++
    end while
    if $s \in T$ then
        return true
    else
        return false
    end if
end function
```

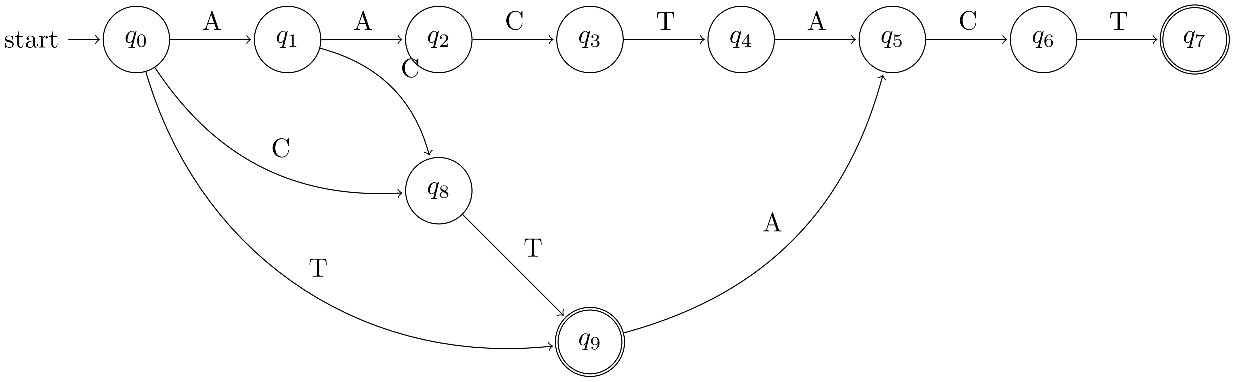

Figure 1.1. Suffix automaton for S = AACTACT

1.3.1. Suffix Automaton

Let S = AACTACT

A suffix automata recognize all suffix of a given sequence.

The suffix language of S is $\{S, ACTACT, CTACT, TACT, ACT, CT, T\}$.

The complexity of the pattern matching algorithm is $\mathcal{O}(n + m)$, because building the automaton is $\mathcal{O}(m)$

1.3.2. Automata for motif search

Let M be a motif M = ACAT.

The alphabet of motif is the same as the alphabet of the sequence. The search automaton is complete. If the there exists a letter c in the sequence that is not in the motif alphabet, we can make a virtual transition from each state to the initial state whenever we encounter an unknown letter.

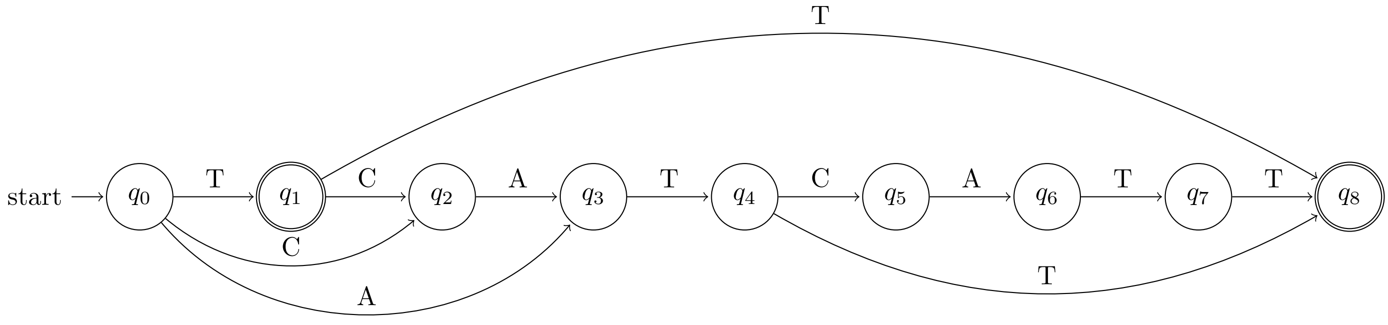

Figure 1.2. Suffix automaton for S = TCATCATT

Algorithm 10 Check if a sequences matches a motif, from a suffix automaton $\mathcal{O}(m)$, built from the automaton

```
function CheckMotifInSuffixAutomaton($W$: Array($m$), $A$: $\langle S, s_0, T, \Sigma, f \rangle$)
    returns Boolean valued to true if the motif is in the sequence
    $s \leftarrow s_0$
    $i \leftarrow 0$
    while $i < m$ and $\exists f(s, W[i])$ do
        $s \leftarrow f(s, W[i])$
        $i++$
    end while
    if $i = n$ then
        return true
    else
        return false
    end if
end function
```

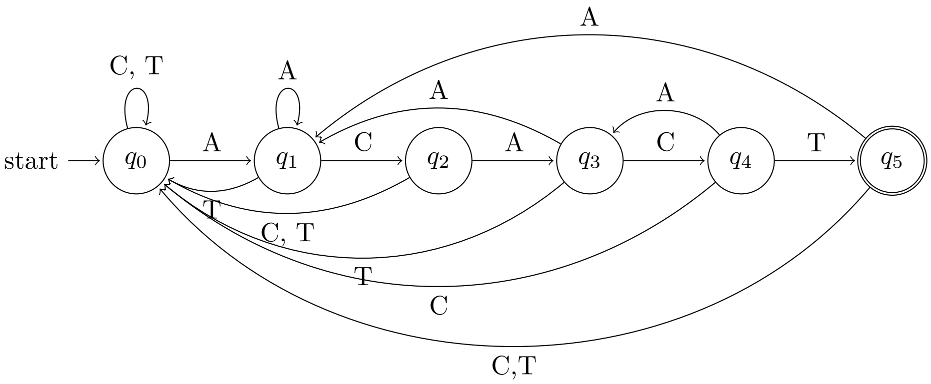

Figure 1.3. Motif search automaton for M = ACAT

Algorithm 11 Search a motif in a sequence with an automaton

```
function SearchMotif($S$: Array($n$), $A$: $\langle S, s_0, T, \Sigma, f\rangle$, $P$: Array($m$))
    returns A set of positions where the motif has been found
    $s \leftarrow s_0$
    $i \leftarrow 0$
    $pos \leftarrow \{\}$
    while $i < n$ do
        if $s \in T$ then
            $pos \leftarrow pos \cup \{i - m\}$
        end if
        $s \leftarrow f(s, S[i])$
        $i++$
    end while
    return $pos$
end function
```

Algorithm 12 Check if the a motif automaton recognizes only the prefix of size $m - 1$ of a motif P of size m

```
function SearchMotifLastPrefix($S$: Array($n$), $A$: $\langle S, s_0, T, \Sigma, f\rangle$, $P$: Array($m$))
    returns A set of positions where the motif has been found
    $s \leftarrow s_0$
    $i \leftarrow 0$
    $T_{new} \leftarrow \{\}$
    for $s \in S$ do
        for $a \in \Sigma$ do
            for $t \in T$ do
                if $\exists f(s, a)$ and $f(s, a) = t$ then
                    $T_{new} \leftarrow T_{new} \cup s$
                end if
            end for
        end for
    end for
    while $i < n$ do
        if $s \in T_{new}$ then
            return true
        end if
        $s \leftarrow f(s, S[i])$
        $i++$
    end while
    return false
end function
```

Algorithm 13 Check if the a motif automaton recognizes only the prefix of size $m-1$ of a motif P of size m, knowing the sequence of the motif

```
function SearchMotifLastPrefix($S$: Array($n$), $A$: $\langle S, s_0, T, \Sigma, f\rangle$, $P$: Array($m$))
    returns A set of positions where the motif has been found
    $s \leftarrow s_0$
    $i \leftarrow 0$
    while $i < n$ and $f(s, P[m-1]) \notin T$ do
        $s \leftarrow f(s, S[i])$
        $i++$
    end while
    if $f(s, P[m-1]) \in T$ then
        return true
    else
        return false
    end if
end function
```

2

Longest common subsequence

Let $S_1 = \text{ATCTGAT}$ and $S_2 = \text{TGCATA}$. In this case the longest common subsequence of S_1 and S_2 is $TCTA$.

Algorithm 14 Construct a longest common subsequence matrix

```
function LCSQ_Matrix($S_1$: Array($n$), $S_2$: Array($m$))
    $M \leftarrow$ Array($m+1$, $n+1$)
    for ($i = 0$; $i < n+1$; $i++$) do
        for $j = 0$; $j < m+1$; $j++$ do
            if $i = 0$ or $j = 0$ then
                $M[i][j] = 0$
            else
                if $S_1[i] = S_2[j]$ then
                    $match = M[i-1][j-1] + 1$
                else
                    $match = M[i-1][j-1]$
                end if
                $gap_1 = M[i-1][j]$
                $gap_2 = M[i][j-1]$
                $M[i][j] = \max\{match, gap_1, gap_2\}$
            end if
        end for
    end for
    return $M$
end function
```

Algorithm 15 Construct a longest common subsequence matrix keeping the path in memory

```
function LCSQ_Matrix_Path($S_1$: Array($n$), $S_2$: Array($m$))
    $M \leftarrow$ Array($m+1$, $n+1$)
    $P \leftarrow$ Array($m+1$, $n+1$)
    for ($i = 0$; $i < n+1$, $i++$) do
        $M[i][0] \leftarrow 0$
    end for
    for ($j = 0$; $j < m+1$; $j+$) do
        $M[0][j] \leftarrow 0$
    end for
    for ($i = 1$; $i < n+1$; $i++$) do
        for ($j = 1$; $j < m+1$; $j++$) do
            if $i = 1$ or $j = 0$ then
                $M[i][j] = 0$
            else
                if $S_1[i-1] = S_2[j-1]$ then
                    $M[i][j] \leftarrow M[i-1][j-1] + 1$
                    $P[i][j] \leftarrow '\nwarrow'$
                else if $M[i][j-1] \geq M[i-1][j]$ then
                    $M[i][j] \leftarrow M[i][j-1]$
                    $P[i][j] \leftarrow '\leftarrow'$
                else
                    $M[i][j] \leftarrow M[i-1][j]$
                    $P[i][j] \leftarrow '\downarrow'$
                end if
            end if
        end for
    end for
    return $M, P$
end function
```

Algorithm 16 Backtrack the longest common subsequence

```
function LCSQ($S_1$: Array($n$), $S_2$: Array($m$))
    $M, P \leftarrow$ LCSQ_Matrix($S_1$, $S_2$)
    $L \leftarrow Array(M[n][m])$
    $k \leftarrow 0$
    $i \leftarrow n$
    $j \leftarrow m$
    while $i > 0$ and $j > 0$ do
        if $P[i][j] = '\nwarrow'$ then
            $L[k] \leftarrow S_1[i]$
            $i--$
            $j--$
            $k++$
        else if $P[i][j] = '\leftarrow'$ then
            $j--$
        else
            $i--$
        end if
    end while
    return $L$
end function
```

Sequence alignment

3 Definitions

A function d is a distance between two sequences x and y in an alphabet Σ if

- $x, y \in \Sigma^*, d(x, x) = 0$
- $\forall x, y \in \Sigma^* \ d(x, y) = d(y, x)$
- $\forall x, y, z \in \Sigma^* \ d(x, z) \leq d(x, y) + d(x, z)$

Here we are interested by the distance that is able to represent the transformation of x to y using three types of basic operations:

- Substitution
- Insertion
- Deletion

Example:

- $sub(a, b) = \begin{cases} 0 & \text{if} a = b \\ 1 & \text{otherwise} \end{cases}.$
- $del(a) = 1$
- $ins(a) = 1$

Let $X = x_0 x_1 \ldots x_{m-1}$, $Y = y_0 y_1 \ldots y_{n-1}$

An alignment is noted as $z = \begin{pmatrix} \bar{x}_0 \\ \bar{y}_0 \end{pmatrix} \ldots \begin{pmatrix} \bar{x}_{p-1} \\ \bar{y}_{p-1} \end{pmatrix}$ of size p. $n \leq p \leq n + m$

$\bar{x}_i = x_j$ or $\bar{x}_i = \varepsilon$ for $0 \leq i \leq p - 1$ and $0 \leq j \leq m - 1$

$\bar{y}_i = y_j$ or $\bar{y}_i = \varepsilon$ for $0 \leq i \leq p - 1$ and $0 \leq j \leq n - 1$

$X' = \bar{x}_0 \bar{x}_1 \ldots \bar{x}_i \ldots \bar{x}_{p-1}$ $Y' = \bar{y}_0 \bar{y}_1 \ldots \bar{y}_i \ldots \bar{y}_{p-1}$ for $0 \leq i \leq p - 1$, $\nexists i$, such that $\bar{x}_i = \bar{y}_i = \varepsilon$

4

Sequence alignment

Algorithm 17 Needleman-Wunsch Algorithm

```
procedure FillMatrix($S_1$: Array($m$), $S_2$: Array($n$))   ▷ $sub(a, b)$ is the substitution score, $del(a)$ and $ins(a)$ are the deletion and insertion penalty, in regard with the reference $S_1$ sequence
    $M$ = Array($m + 1$, $n + 1$)   ▷ Initialize the matrix first column and first row
    $P$ = Array($m$, $n$)   ▷ Store the direction of the cell we chose to build the next cell up on.
    $M[0][0] = 0$
    for $(i = 1;\ i < m + 1;\ i++)$ do
        $M[i][0] = M[i-1][0] + gap\_penalty$
    end for
    for $(j = 1;\ j < n + 1;\ j++)$ do
        $M[0][j] = M[0][j-1] + gap\_penalty$
    end for   ▷ Fill the remaining matrix
    for $(i = 1;\ i < m + 1;\ i++)$ do
        for $(j = 1;\ j < n + 1;\ j++)$ do
            $delete = M[i-1][j] + gap\_penalty$
            $insert = M[i][j-1] + gap\_penalty$
            $substitute = M[i-1][j-1] + sub(S_1[i-1], S_2[j-1])$
            $choice = \min\{delete, insert, substitute\}$
            if $substitute = choice$ then
                $P[i-1][j-1] = '\nwarrow'$
            else if $deletion = choice$ then
                $P[i-1][j-1] = '\leftarrow'$
            else
                $P[i-1][j-1] = '\uparrow'$
            end if
            $M[i][j] = choice$
        end for
    end for
end procedure
```

Algorithm 18 Needleman-Wunsch Algorithm (Backtrack)

```
**procedure** ShowAlignment($S_1$: Array($m$), $S_2$: Array($n$))
    $extend_1 =''$
    $extend_2 =''$
    $i = m$
    $j = n$
    **while** $i > 0$ and $j > 0$ **do**
        **if** $P[i-1][j-1] =' \nwarrow '$ **then**
            $extend_1 = S_1[i-1] \circ extend_1$
            $extend_2 = S_2[j-1] \circ extend_2$
            $i--$
            $j--$
        **else if** $P[i-1][j-1] =' \uparrow '$ **then**
            $extend_1 = S_1[i-1] \circ extend_1$
            $extend_2 =' -' \circ extend_2$
            $i--$
        **else**
            $extend_1 =' -' \circ extend_1$
            $extend_2 = S_2[j-1] \circ extend_2$
            $j--$
        **end if**
    **end while**
    **while** $i > 0$ **do**
        $extend_1 = S_1[i-1] \circ extend_1$
        $extend_2 =' -' \circ extend_2$
        $i--$
        Insert(0, $alignment$,$tuple$)
    **end while**
    **while** $j > 0$ **do**
        $extend_1 =' -' \circ extend_1$
        $extend_2 = S_2[j-1] \circ extend_2$
        $j--$
    **end while**
    print($extend_1$)
    print($extend_2$)
**end procedure**
FillMatrix($S_1$, $S_2$)
ShowAlignment($S_1$, $S_2$)
```

Algorithm 19 Needleman-Wunsch Algorithm, using proper notation

```
procedure FillMatrix($S_1$: Array($m$), $S_2$: Array($n$))
    $M$ = Array($m + 1$, $n + 1$)
    $P$ = Array($m$, $n$)    ▷ Store the direction of the cell we chose to build the next cell up on.
    $M[0][0] = 0$
    for ($i = 1$; $i < m + 1$; $i + +$) do
        $M[i][0] = M[i-1][0] + gap\_penalty$
    end for
    for ($j = 1$; $j < n + 1$; $j + +$) do
        $M[0][j] = M[0][j-1] + gap\_penalty$
    end for
    for ($i = 1$; $i < m + 1$; $i + +$) do
        for ($j = 1$; $j < n + 1$; $j + +$) do
            $delete = M[i-1][j] + gap\_penalty$
            $insert = M[i][j-1] + gap\_penalty$
            $substitute = M[i-1][j-1] + sub(S_1[i-1], S_2[j-1])$
            $M[i][j] = \min\{substitute, insert, delete\}$
        end for
    end for
end procedure
```

Algorithm 20 Needleman-Wunsch Algorithm, using proper notation (Backtrack)

```
procedure BacktrackAlignment($S_1$: Array($m$), $S_2$: Array($n$))
    $alignment = LinkedList$
    $i = m$
    $j = n$
    while $i > 0$ and $j > 0$ do
        if $M[i-1][j-1] = M[i][j] - sub(S_1[i-1], S_2[j-1])$ then
            $tuple = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix}$
            $i--$
            $j--$
        else if $M[i-1][j-1] = M[i][j-1] - gap\_penalty$ then
            $tuple = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
            $i--$
        else
            $tuple = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
            $j--$
        end if
        Insert(0, $alignment$,$tuple$)
    end while
    while $i > 0$ do
        $tuple = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
        $i--$
        Insert(0, $alignment$,$tuple$)
    end while
    while $j > 0$ do
        $tuple = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
        $j--$
        Insert(0, $alignment$,$tuple$)
    end while
end procedure
FillMatrix($S_1$, $S_2$)
BacktrackAlignment($S_1$, $S_2$)
```

Algorithm 21 Backtrack a single alignment in a recursive way

```
$S_1$: Array($m$), $S_2$: Array($n$), $M$: Array($m+1$, $n+1$),
function BacktrackRecurse($i$, $j$)
    if $i > 0$ and $j > 0$ then
        $substitute = M[i-1][j-1]$
        $delete = M[i-1][j]$
        $insert = M[i][j-1]$
        $min = \min\{substitute, delete, insert\}$
        if $substitute = min$ then
            $z$ = BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j-1$)
            $z = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix} \circ z$
        else if $delete = min$ then
            $z$ = BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j$)
            $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
        else
            $z$ = BacktrackRecurse($S_1$, $S_2$, $M$, $i$, $j-1$)
            $z = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix} \circ z$
        end if
    else if $i > 0$ then
        $z$ = BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j$)
        $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
    else if $j > 0$ then
        $z$ = BacktrackRecurse($S_1$, $S_2$, $M$, $i$, $j-1$)
        $z = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix} \circ z$
    else
        return []
    end if
    return $z$
end function
function Backtrack($S_1$: Array($m$), $S_2$: Array($n$), $M$: Array($m+1$, $n+1$))
    return BacktrackRecurse($S_1$, $S_2$, $M$, $m$, $n$)
end function
```

Algorithm 22 Backtrack all the optimum alignments in a recursive way

```
procedure BacktrackRecurse($S_1$: Array($m$), $S_2$: Array($n$), $M$: Array($m+1$, $n+1$), $i$, $j$)
    if $i > 0$ and $j > 0$ then
        $substitute = M[i-1][j-1]$
        $delete = M[i-1][j]$
        $insert = M[i][j-1]$
        $min = \min\{substitute, delete, insert\}$
        if $substitute = min$ then
            $value = \begin{pmatrix} S_1[i-1] \\ S_2[j-1] \end{pmatrix}$
            $z' = value \circ z$
            BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j-1$, $z'$)
        end if
        if $delete = min$ then
            $value = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
            $z' = value \circ z$
            BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j$, $z'$)
        end if
        if $insert = min$ then
            $value = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
            $z' = value \circ z$
            BacktrackRecurse($S_1$, $S_2$, $M$, $i$, $j-1$, $z'$)
        end if
    else if $i > 0$ then
        $value = \begin{pmatrix} S_1[i-1] \\ \varepsilon \end{pmatrix}$
        $z' = value \circ z$
        BacktrackRecurse($S_1$, $S_2$, $M$, $i-1$, $j$, $z'$)
    else if $j > 0$ then
        $value = \begin{pmatrix} \varepsilon \\ S_2[j-1] \end{pmatrix}$
        $z' = value \circ z$
        BacktrackRecurse($S_1$, $S_2$, $M$, $i$, $j-1$, $z'$)
    end if
    print($z$)
end procedure
procedure Backtrack($S_1$: Array($m$), $S_2$: Array($n$), $M$: Array($m+1$, $n+1$))
    return BacktrackRecurse($S_1$, $S_2$, $M$, $m$, $n$, [])
end procedure
```