Combinatorial pattern matching is the search for exact or approximate occurrences of a given pattern within a given text.

When it comes to biological sequences, both the pattern and the text are sequences and the pattern matching problem becomes one of finding the occurrences of a sequence within another sequence.

For instance, scanning a protein sequence for the presence of a known pattern can help annotate both the protein and the corresponding genome, and finding a sequence within another sequence can help in assessing their similarities and differences.

Definition

There are several ways in which a sequence can be contained in another sequence.

A sequence can be a prefix of a longer sequence, or it can be a suffix of the longer sequence.

Also, a sequence can be contained deeper within another sequence: it can be a suffix of a prefix or, equivalently, a prefix of a suffix of the other sequence.

Example

The sequence TATTTGATCATT is contained in the following DNA sequence.

```
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

It is a suffix of a prefix of the longer sequence, as shown in the following alignment.

```
          TATTTGATCATT
TTGATTACCTTATTTGATCATT
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

It is also a prefix of a suffix of the longer sequence, as illustrated by the following alignment.

```
          TATTTGATCATT
          TATTTGATCATTACACATTGTACGCTTGTG
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

Definition

A sequence can also be contained many times in another sequence, and the occurrences of the shorter sequence in the longer sequence may even overlap.

Example

There are four occurrences of the sequence ATT (top) but only two occurrences of the sequence ATTAC (middle) in the following longer DNA sequence (bottom).

```
   ATT     ATT     ATT    ATT
   ATTAC           ATTAC
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

There are also two overlapping occurrences of the short sequence ACA within the longer sequence.

```
                      ACA
                        ACA
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

Algorithm

The occurrences of a given sequence within another given sequence can be found by traversing the longer sequence, starting at each possible initial position in turn where the shorter sequence could occur.

The starting positions of the occurrences of a pattern sequence P of length m in a text sequence T of length n, with $m \leqslant n$, are collected in a list L.

```
function occurrences($P, T$)
    $m \leftarrow \text{length}(P)$
    $n \leftarrow \text{length}(T)$
    $L \leftarrow \emptyset$
    for $i \leftarrow 1$ to $n - m + 1$ do
        if $P[1, 2, \ldots, m] = T[i, i+1, \ldots, i+m-1]$ then
            $L \leftarrow L \cup \{i\}$
    return $L$
```

Remark

The algorithm for finding all occurrences of a sequence within a longer sequence by traversing part of the latter once for each possible starting position of the shorter sequence has the disadvantage that it can be rather slow in practice for large sequences, because it cannot always be established whether or not $P[1,2,\ldots,m]$ matches $T[i,i+1,\ldots,i+m-1]$ until m elementary comparison operations have been made and this test is repeated $n-m+1$ times altogether.

Finding all occurrences of a sequence in another sequence is indeed a classic pattern matching problem, for which faster algorithms are known.

Some of them are based on storing all the suffixes of a sequence in a compact representation called a suffix array.

Definition

The suffix array of a sequence is a permutation of all starting positions of the suffixes of the sequence, lexicographically sorted.

Despite its simplicity, the suffix array of a sequence is very useful for finding subsequences, because all the occurrences of a sequence within another sequence appear together in the suffix array, as prefixes of suffixes of the longer sequence.

Example

The DNA sequence

```
TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
```

has the following suffixes, starting at positions 1 through 40.

```
 [1]  TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
 [2]  TGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
 [3]  GATTACCTTATTTGATCATTACACATTGTACGCTTGTG
 [4]  ATTACCTTATTTGATCATTACACATTGTACGCTTGTG
 [5]  TTACCTTATTTGATCATTACACATTGTACGCTTGTG
 [6]  TACCTTATTTGATCATTACACATTGTACGCTTGTG
 ...  ...
[35]  CTTGTG
[36]  TTGTG
[37]  TGTG
[38]  GTG
[39]  TG
[40]  G
```

Example

In lexicographical order, these suffixes define the suffix array of the sequence.

```
[23] ACACATTGTACGCTTGTG
[25] ACATTGTACGCTTGTG
 [7] ACCTTATTTGATCATTACACATTGTACGCTTGTG
[32] ACGCTTGTG
[17] ATCATTACACATTGTACGCTTGTG
[20] ATTACACATTGTACGCTTGTG
 ...  ...
[10] TTATTTGATCATTACACATTGTACGCTTGTG
[14] TTGATCATTACACATTGTACGCTTGTG
 [1] TTGATTACCTTATTTGATCATTACACATTGTACGCTTGTG
[28] TTGTACGCTTGTG
[36] TTGTG
[13] TTTGATCATTACACATTGTACGCTTGTG
```

Example

The actual suffix array of the sequence is just the array of starting positions.

```
23 25  7 32 17 20  4 27 12 24 19 26  8 33  9 35
40 16  3 34 30 38 22  6 31 11 18 39 15  2 29 37
21  5 10 14  1 28 36 13
```

Algorithm

The suffix array of a sequence can be obtained by various methods.

The simplest way to obtain the suffix array of a sequence consists of sorting an array of positions by comparing the corresponding suffixes of the sequence.

The suffix array for a sequence S of length n is built by sorting an array A of positions 1 through n, where the comparison of array entries $A[i]$ and $A[j]$ is based on the comparison of suffixes $S[A[i], \dots, n]$ and $S[A[j], \dots, n]$.

```
procedure suffix_array($S, A$)
    $n \leftarrow$ length($S$)
    for $i \leftarrow 1$ to $n$ do
        $A[i] \leftarrow i$
    sort $A$ by comparing $S[A[i], \dots, n]$ with $S[A[j], \dots, n]$
```

Definition

Given a pattern sequence and the suffix array of another sequence, an occurrence of the pattern in the sequence can be obtained by traversing the suffix array and comparing the pattern sequence with the suffixes of the sequence that start at the positions stored in the suffix array.

Since the occurrences of the pattern sequence within the other sequence appear together, as prefixes of suffixes, in the suffix array of the longer sequence, however, an occurrence of the pattern sequence can be found much more quickly by performing a binary search on the suffix array of the longer sequence.

Algorithm

An occurrence of a pattern sequence P of length m in another sequence T of length n, with $m \leqslant n$, is obtained by binary search on the suffix array $A[1, \ldots, n]$ of T.

Starting with the index $i = (1 + n)/2$, the pattern P is compared to the first m characters of the suffix of T starting at position $A[i]$, and, as a result, either the search continues in $A[1, \ldots, i-1]$ because $P < T[A[i], \ldots, A[i] + m - 1]$ in lexicographical order, it continues in $A[i+1, \ldots, n]$ because $P > T[A[i], \ldots, A[i] + m - 1]$ in lexicographical order, or it finishes because an occurrence of P as a prefix of the suffix of T starting at position $A[i]$ was found.

The portion of the suffix array being searched is always $A[\ell, \ldots, r]$.

Algorithm

Once an occurrence, at position $A[i]$, of pattern sequence P in sequence T has been found, the remaining occurrences (if any) are obtained by extending the interval (i, j) of suffix array indices as much as possible, where initially $i = j$, profiting from the fact that all occurrences appear together in A.

The lower index i of the interval is decreased by one for each further occurrence $T[A[i-1], \dots, A[i-1]+m-1]$ towards the beginning of the suffix array, and the upper index $j = i$ is increased by one for each further occurrence $T[A[j+1], \dots, A[j+1]+m-1]$ towards the end of the suffix array.

The resulting interval (i, j) comprises the starting positions in suffix array A of all the occurrences of sequence P in sequence T, with $T[A[i], \dots, A[i]+m-1]$ as the first occurrence and with $T[A[j], \dots, A[j]+m-1]$ as the last occurrence.

Algorithm

```
function occurrences($P, T, A$)
    $m \leftarrow \text{length}(P)$
    $n \leftarrow \text{length}(T)$
    $\ell \leftarrow 1$
    $r \leftarrow n$
```

Algorithm

```
while $\ell \leqslant r$ do
    $i \leftarrow (\ell + r)/2$
    if $P < T[A[i], \ldots, A[i] + m - 1]$ then
        $r \leftarrow i - 1$
    else if $P > T[A[i], \ldots, A[i] + m - 1]$ then
        $\ell \leftarrow i + 1$
    else
        $j \leftarrow i$
        while $i > 1$ and $P = T[A[i-1], \ldots, A[i-1] + m - 1]$ do
            $i \leftarrow i - 1$
        while $j < n$ and $P = T[A[j+1], \ldots, A[j+1] + m - 1]$ do
            $j \leftarrow j + 1$
        return $(i, j)$
return $(-1, -1)$
```

Definition

The occurrences of a sequence in another sequence can be found even more quickly when the suffixes of the sequence share long prefixes, because when comparing $P[1,\ldots,m]$ with $T[A[i],\ldots,A[i]+m-1]$ in the previous algorithm, the length of the longest common prefix between them can be obtained as a by-product of their comparison, and it can be used in later iterations to shorten the suffixes that remain to be compared.

Definition

In fact, when searching for the occurrences of sequence P between positions ℓ and r of the suffix array A of sequence T, it suffices to keep track of the length llcp of the longest common prefix between P and the suffix $T[A[\ell],\dots,n]$ as well as the length rlcp of the longest common prefix between P and the suffix $T[A[r],\dots,n]$, because the shortest of these two longest common prefixes will be a prefix common to all the suffixes starting at positions $A[\ell]$ through $A[r]$ of sequence T.

That is, if $P[1,\dots,llcp] = T[A[\ell],\dots,A[\ell]+llcp-1]$ and $P[1,\dots,rlcp] = T[A[r],\dots,A[r]+rlcp-1]$, then $P[1,\dots,h] = T[A[k],\dots,A[k]+h-1]$ for all $\ell \leqslant k \leqslant r$, where h is the smallest of llcp and rlcp.

Algorithm

The length of the longest common prefix between $P[i,\ldots,m]$ and $T[j,\ldots,n]$ can be obtained by traversing the two sequences and counting the number c of common elements in their prefixes until $P[i+c] \neq T[j+c]$.

```
function lcp($P[i,\ldots,m], T[j,\ldots,n]$)
    $c \leftarrow 0$
    while $i+c \leqslant m$ and $j+c \leqslant n$ and $P[i+c] = T[j+c]$ do
        $c \leftarrow c+1$
    return $c$
```

Algorithm

The occurrences of sequence $P[1,\dots,m]$ in sequence $T[1,\dots,n]$, where $m \leqslant n$, are obtained by binary search on the portion $A[\ell,\dots,r]$ of the suffix array of T, while keeping track of the length llcp of the longest common prefix between P and the suffix $T[A[\ell],\dots,n]$ as well as the length rlcp of the longest common prefix between P and the suffix $T[A[r],\dots,n]$.

Then the lexicographical comparison of $P[1,\dots,m]$ with $T[A[i],\dots,A[i]+m-1]$ is replaced by the computation of the length c of the longest common prefix between $P[h+1,\dots,m]$ and $T[A[i]+h,\dots,n]$, where h is the smallest of llcp and rlcp, and $P[1,\dots,m] = T[A[i],\dots,A[i]+m-1]$ if $h+c=m$.

Otherwise, the positions of the $h+c$ common elements are skipped, and $P[h+c+1]$ and $T[A[i]+h+c]$ are compared next.

Algorithm

```
**function** occurrences($P, T, A$)
    $m \leftarrow \text{length}(P)$
    $n \leftarrow \text{length}(T)$
    $\ell \leftarrow 1$
    $r \leftarrow n$
    $llcp \leftarrow rlcp \leftarrow 0$
    **while** $\ell \leqslant r$ **do**
        $i \leftarrow (\ell + r)/2$
        $h \leftarrow \min\{llcp, rclp\}$
        $c \leftarrow \text{lcp}(P[h+1, \ldots, m], T[A[i]+h, \ldots, n])$
```

Algorithm

```
        if $h + c = m$ then
            $j \leftarrow i$
            while $i > 1$ and $P = T[A[i-1], \ldots, A[i-1] + m - 1]$ do
                $i \leftarrow i - 1$
            while $j < n$ and $P = T[A[j+1], \ldots, A[j+1] + m - 1]$ do
                $j \leftarrow j + 1$
            return $(i, j)$
        else if $P[h + c + 1] < T[A[i] + h + c]$ then
            $r \leftarrow i - 1$
            $rlcp \leftarrow h + c$
        else
            $\ell \leftarrow i + 1$
            $llcp \leftarrow h + c$
    return $(-1, -1)$
```

Definition

Subsequences shared by two sequences reveal information common to the two sequences.

As there are several ways in which a sequence can be contained in another sequence, common subsequences can be common prefixes, suffixes, suffixes of prefixes, or prefixes of suffixes.

Further, in order to reveal the most of their shared information, it is interesting to find common subsequences of largest size between two given sequences.

Example

The following fragments of DNA sequence

```
TGCTTCTGACTATAATAG
GCTTCCGGCTCGTATAATGTGTGG
```

contain the Pribnow box TATAAT as their longest common subsequence, as shown in the two alignments below.

```
          TATAAT                       TATAAT
TGCTTCTGACTATAATAG      GCTTCCGGCTCGTATAATGTGTGG
```

Algorithm

The common occurrences of a pattern as a subsequence of two given sequences can be found by traversing the two sequences, starting at each possible initial position in turn where the pattern sequence could occur.

The starting positions of the occurrences of a pattern sequence P of length m in each of the two text sequences T_1 of length n_1 and T_2 of length n_2, with $m \leqslant \min\{n_1, n_2\}$, are collected in a list L.

Algorithm

```
**function** common_occurrences($P, T_1, T_2$)
    $m \leftarrow$ length($P$)
    $n_1 \leftarrow$ length($T_1$)
    $n_2 \leftarrow$ length($T_2$)
    $L \leftarrow \emptyset$
    for $i \leftarrow 1$ to $n_1 - m + 1$ do
        if $P[1,\ldots,m] = T_1[i,\ldots,i+m-1]$ then
            for $j \leftarrow 1$ to $n_2 - m + 1$ do
                if $P[1,\ldots,m] = T_2[j\ldots,j+m-1]$ then
                    $L \leftarrow L \cup \{(i,j)\}$
    return $L$
```

Algorithm

On the other hand, the longest common subsequences of two given sequences can be found by traversing the two sequences, starting at each possible initial position of each sequence in turn where a common subsequence could occur.

Given two sequences S_1 and S_2, the subsequences $S_1[i,\ldots,j]$, of length $j-i+1$, and $S_2[k,\ldots,k+j-i]$, also of length $k+j-i-k+1=j-i+1$, are compared to each other, and the longest common subsequences, of length ℓ, are collected in a list L.

Algorithm

```
**function** longest_common_subsequences($S_1, S_2$)
    $L \leftarrow \emptyset$
    $\ell \leftarrow 0$
    **for** $i \leftarrow 1$ **to** $length(S_1)$ **do**
        **for** $j \leftarrow i$ **to** $length(S_1)$ **do**
            $X \leftarrow S_1[i, \ldots, j]$
            **for** $k \leftarrow 1$ **to** $length(S_2) - j + i$ **do**
                $Y \leftarrow S_2[k, \ldots, k + j - i]$
                **if** $X = Y$ **then**
                    **if** $length(X) = \ell$ **then**
                        $L \leftarrow L \cup \{X\}$
                    **else if** $length(X) > \ell$ **then**
                        $L \leftarrow \{X\}$
                        $\ell \leftarrow length(X)$
    return $L$
```

Remark

The algorithm for finding the longest common subsequences of two sequences S_1 and S_2 by traversing part of the sequences once for each possible starting position of a common subsequence has the disadvantage that it can be slow in practice for large sequences, because a subsequence $S_1[i, \ldots, j]$ is compared to a subsequence $S_2[k, \ldots, k+j-i]$ even if the comparison of a prefix of the former to a prefix of the latter has already failed and, thus, they cannot be common subsequences.

Finding the longest common subsequences of two sequences is indeed another classic pattern matching problem, for which faster algorithms are known.

Some of them are based on dynamic programming, while others are based on storing all the suffixes of the two sequences in a compact representation called a generalized suffix array.

Definition

The longest common subsequences of two sequences can be obtained by finding the longest common suffixes between each pair of prefixes of the sequences, keeping all the longest ones.

In general, the length $LCS(S_1[1,\dots,i],S_2[1,\dots,j])$ of a longest common suffix between two prefixes $S_1[1,\dots,i]$ and $S_2[1,\dots,j]$ of the sequences S_1 and S_2 is given by the recurrence

$$LCS(S_1[1,\dots,i],S_2[1,\dots,j]) = \begin{cases} LCS(S_1[1,\dots,i-1],S_2[1,\dots,j-1])+1 \\ \qquad \text{if } S_1[i]=S_2[j] \\ 0 \qquad \text{otherwise} \end{cases}$$

Computation of this recurrence by dynamic programming involves the use of a dynamic programming table to store each $LCS(S_1[1,\dots,i],S_2[1,\dots,j])$, for $1 \leqslant i \leqslant n_1$ and $1 \leqslant j \leqslant n_2$, where n_1 is the length of S_1 and n_2 is the length of S_2.

Example

The longest common subsequences of the two sequences TGCTTCTGACTATAATAG and GCTTCCGGCTCGTATAATGTGTGG have length 6, as shown in entry $(16, 18)$ of the following dynamic programming table.

Prefix TGCTTCTGACTATAAT of the first sequence and prefix GCTTCCGGCTCGTATAAT of the second sequence share TATAAT as a common suffix.

Example

		G	C	T	T	C	C	G	G	C	T	C	G	T	A	T	A	A	T	G	T	G	T	G	G
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	1	0	0	1	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0
G	2	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	2	0	2	0	2	1
C	3	0	2	0	0	1	1	0	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
T	4	0	0	3	1	0	0	0	0	0	3	0	0	1	0	1	0	0	1	0	1	0	1	0	0
T	5	0	0	1	4	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0
C	6	0	1	0	0	5	1	0	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
T	7	0	0	2	1	0	0	0	0	0	2	0	0	1	0	1	0	0	1	0	1	0	1	0	0
G	8	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	2	0	2	0	2	1
A	9	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0
C	10	0	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
T	11	0	0	2	1	0	0	0	0	0	2	0	0	1	0	1	0	0	1	0	1	0	1	0	0
A	12	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	1	0	0	0	0	0	0	0
T	13	0	0	1	1	0	0	0	0	0	1	0	0	1	0	3	0	0	2	0	1	0	1	0	0
A	14	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	4	1	0	0	0	0	0	0	0
A	15	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	5	0	0	0	0	0	0	0
T	16	0	0	1	1	0	0	0	0	0	1	0	0	1	0	2	0	0	6	0	1	0	1	0	0
A	17	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	1	0	0	0	0	0	0	0
G	18	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1

Algorithm

The dynamic programming table *LCS* is filled in for each $1 \leqslant i \leqslant n_1$ and $1 \leqslant j \leqslant n_2$ while keeping track of the length ℓ of a longest common suffix between all pairs of prefixes considered so far, and all common suffixes of prefixes of length $i-(i-\ell+1)+1=\ell$ are collected in a list L.

Algorithm

```
function longest_common_subsequences($S_1, S_2$)
    $L \leftarrow \emptyset$
    $\ell \leftarrow 0$
```

Algorithm

```
for $i \leftarrow 1$ to $length(S_1)$ do
    for $j \leftarrow i$ to $length(S_2)$ do
        $LCS[i,j] \leftarrow 0$
        if $S_1[i] = S_2[j]$ then
            if $i = 1$ or $j = 1$ then
                $LCS[i,j] \leftarrow 1$
            else
                $LCS[i,j] \leftarrow LCS[i-1,j-1]+1$
            if $LCS[i,j] > \ell$ then
                $\ell \leftarrow LCS[i,j]$
                $L \leftarrow \emptyset$
            if $LCS[i,j] = \ell$ then
                $L \leftarrow L \cup \{S_1[i-\ell+1,\ldots,i]\}$
return $L$
```

Definition

The generalized suffix array of two sequences is a permutation of all starting positions of the suffixes of the sequences, lexicographically sorted, where the starting position of each suffix is tagged as coming from the first or the second sequence.

Despite its simplicity, the generalized suffix array of two sequences is very useful for finding common subsequences, because all the occurrences of a sequence within both of the sequences appear together in the generalized suffix array, as prefixes of suffixes of the sequences.

Algorithm

As in the case of the suffix array of a sequence, the generalized suffix array of two sequences can also be obtained by various methods.

The simplest way to obtain the generalized suffix array of two sequences consists in first tagging the sequences and then sorting an array of positions by comparing the corresponding suffixes of the tagged sequences.

The generalized suffix array for a sequence S_1 of length n_1 and a sequence S_2 of length n_2 is built by sorting an array A of positions 1 through n_1 tagged by the sequence identifier 1 and positions 1 through n_2 tagged by the sequence identifier 2, where the comparison of an array entry $A[i]$ tagged by, say, 1 and an array entry $A[j]$ tagged by, say, 2 is based on the comparison of suffixes $S_1[A[i],\ldots,n_1]$ and $S_2[A[j],\ldots,n_2]$.

Algorithm

```
procedure generalized_suffix_array($S_1, S_2, A$)
    $n_1 \leftarrow \text{length}(S_1)$
    $n_2 \leftarrow \text{length}(S_2)$
    for $i \leftarrow 1$ to $n_1$ do
        $A[i] \leftarrow (i, 1)$
    for $i \leftarrow 1$ to $n_2$ do
        $A[n_1 + i] \leftarrow (i, 2)$
    sort $A$ by comparing $S_1[A[i], \ldots, n_1]$ with $S_2[A[j], \ldots, n_2]$
```

Algorithm

The common occurrences of a pattern as a subsequence of two given sequences can also be found by binary search on the generalized suffix array of the two sequences.

A common occurrence of a pattern sequence P of length m in a sequence S_1 of length n_1, with $m \leqslant n_1$, and a sequence S_2 of length n_2, with $m \leqslant n_2$, is obtained by binary search on the generalized suffix array $A[1, \dots, n_1 + n_2]$ of S_1 and S_2.

Starting with the index $i = (1 + n_1 + n_2)/2$, the pattern P is compared to the first m characters of the suffix of S_t starting at position k, where $(k, t) = A[i]$, and, as a result, either the search continues in $A[1, \dots, i-1]$ because $P < S_t[k, \dots, k+m-1]$ in lexicographical order, or it continues in $A[i+1, \dots, n]$ because $P > S_t[k, \dots, k+m-1]$ in lexicographical order, or it finishes because an occurrence of P as a prefix of the suffix of S_t starting at position k was found.

The portion of the generalized suffix array being searched is always $A[\ell, \dots, r]$.

Algorithm

Once a common occurrence, at position k, of pattern sequence P in sequence S_t has been found, where $(k,t) = A[i]$, the remaining occurrences (if any) are obtained by extending the interval (i,i) of generalized suffix array indices as much as possible, profiting from the fact that all occurrences appear together in A.

The lower index i of the interval is decreased by one for each further occurrence towards the beginning of the generalized suffix array, and the upper index $j = i$ is increased by one for each further occurrence towards the end of the generalized suffix array.

The resulting interval (i,j) comprises the starting positions in A of all the common occurrences of sequence P in sequences S_1 and S_2.

Algorithm

```
function common_occurrences($P, S_1, S_2, A$)
    $m \leftarrow \text{length}(P)$
    $n_1 \leftarrow \text{length}(S_1)$
    $n_2 \leftarrow \text{length}(S_2)$
    $\ell \leftarrow 1$
    $r \leftarrow n_1 + n_2$
```

Algorithm

```
while $\ell \leqslant r$ do
    $i \leftarrow (\ell + r)/2$
    $(k, t) \leftarrow A[i]$
    if $P < S_t[k, \ldots, k + m - 1]$ then
        $r \leftarrow i - 1$
    else if $P > S_t[k, \ldots, k + m - 1]$ then
        $\ell \leftarrow i + 1$
    else
        $j \leftarrow i$
        while $i > 1$, $(k, t) \leftarrow A[i - 1]$ and $P = S_t[k, \ldots, k + m - 1]$ do
            $i \leftarrow i - 1$
        while $j < n$, $(k, t) \leftarrow A[j + 1]$ and $P = S_t[k, \ldots, k + m - 1]$ do
            $j \leftarrow j + 1$
        return $(i, j)$
return $(-1, -1)$
```

Algorithm

As in the case of suffix arrays, the occurrences of a sequence common to two sequences can be found even more quickly when the suffixes of the sequences share long prefixes, because when comparing $P[1,\dots,m]$ with $S_t[k,\dots,k+m-1]$ in the previous algorithm, the length of the longest common prefix between them can be obtained as a by-product of their comparison, and it can be used in later iterations to shorten the suffixes that remain to be compared.

In fact, when searching for the occurrences of sequence P between positions ℓ and r of the generalized suffix array A of sequences S_1 and S_2, it suffices to keep track of the length llcp of the longest common prefix between P and the suffix $S_t[k,\dots,n_t]$, where $A[\ell]=(k,t)$, as well as the length rlcp of the longest common prefix between P and the suffix $S_t[k\dots,n_t]$, where $A[r]=(k,t)$, because the shortest of these two longest common prefixes will be a prefix common to all the suffixes starting at positions $A[\ell]$ through $A[r]$ of sequences S_1 or S_2.

Algorithm

A common occurrence of a pattern sequence P of length m in a sequence S_1 of length n_1, with $m \leqslant n_1$, and a sequence S_2 of length n_2, with $m \leqslant n_2$, is obtained by binary search on the portion $A[\ell, \dots, r]$ of the generalized suffix array of S_1 and S_2, while keeping track of the length llcp of the longest common prefix between P and the suffix $S_t[k, \dots, n_t]$, where $A[\ell] = (k, t)$, as well as the length rlcp of the longest common prefix between P and the suffix $S_t[k, \dots, n_t]$, where $A[r] = (k, t)$.

Then the lexicographical comparison of $P[1, \dots, m]$ with $S_t[k, \dots, k+m-1]$ is replaced by the computation of the length c of the longest common prefix between $P[h+1, \dots, m]$ and $S_t[k+h, \dots, n_t]$, where h is the smallest of llcp and rlcp, and $P[1, \dots, m] = S_t[k, \dots, k+m-1]$ if $h+c=m$.

Otherwise, the positions of the $h+c$ common elements are skipped, and $P[h+c+1]$ and $S_t[k+h+c]$ are compared next.

Algorithm

```
function common_occurrences($P, S_1, S_2, A$)
    $m \leftarrow$ length($P$)
    $n_1 \leftarrow$ length($S_1$)
    $n_2 \leftarrow$ length($S_2$)
    $\ell \leftarrow 1$
    $r \leftarrow n \leftarrow n_1 + n_2$
    $llcp \leftarrow rlcp \leftarrow 0$
    while $\ell \leqslant r$ do
        $i \leftarrow (\ell + r)/2$
        $(k, t) \leftarrow A[i]$
        $h \leftarrow \min\{llcp, rclp\}$
        $c \leftarrow \mathrm{lcp}(P[h+1, \ldots, m], S_t[k+h, \ldots, n_t])$
```

Algorithm

```
    if $h + c = m$ then
        $j \leftarrow i$
        while $i > 1$, $(k,t) \leftarrow A[i-1]$ and $P = S_t[k,\dots,k+m-1]$ do
            $i \leftarrow i - 1$
        while $j < n$, $(k,t) \leftarrow A[j+1]$ and $P = S_t[k,\dots,k+m-1]$ do
            $j \leftarrow j + 1$
        return $(i,j)$
    else if $P[h+c+1] < S_t[k+h+c]$ then
        $r \leftarrow i - 1$
        $rlcp \leftarrow h + c$
    else
        $\ell \leftarrow i + 1$
        $llcp \leftarrow h + c$
return $(-1,-1)$
```

Algorithm

On the other hand, the longest common subsequences of two given sequences can be found by traversing the generalized suffix array of the two sequences and computing the longest common prefix of each pair of consecutive entries if they correspond to different sequences.

The length p of the longest common prefix between suffixes F_1 and F_2 in each pair of consecutive entries of the generalized suffix array A is computed, and the largest length ℓ of the longest common prefixes found so far is kept together with a list L of all such longest common subsequences.

Algorithm

```
function longest_common_subsequences($S_1, S_2, A$)
    $L \leftarrow \emptyset$
    $\ell \leftarrow 0$
    $(k, t) \leftarrow A[1]$
    $F_2 \leftarrow S_t[k, \ldots, n_t]$
```

Algorithm

```
for $i \leftarrow 2$ to $n_1 + n_2$ do
    $F_1 \leftarrow F_2$
    $(k, t) \leftarrow A[i]$
    $F_2 \leftarrow S_t[k, \ldots, n_t]$
    $(k', t') \leftarrow A[i-1]$
    if $t \neq t'$ then
        $p \leftarrow lcp(F_1, F_2)$
        if $p > \ell$ then
            $\ell \leftarrow p$
            $L \leftarrow \emptyset$
        if $p = \ell$ then
            $L \leftarrow L \cup \{S_t[1, \ldots, p]\}$
return $L$
```

Algorithm

On the other hand, the longest common subsequences of a given set of $k \geqslant 2$ sequences can be found by traversing the generalized suffix array of the sequences and computing the longest common prefix of each set of k consecutive entries if they correspond to different sequences.

Example

The DNA sequences

```
GATTACA
TAGACCA
ATACA
```

have three longest common subsequences of length two.

```
GATTACA-
-TAGACCA
--ATACA-
```

```
GATTACA
TAGACCA
--ATACA
```

```
GATTACA---
---TAGACCA
--ATACA---
```

Example

```
[7, 1] A
[7, 2] A
[5, 3] A
[5, 1] ACA
[3, 3] ACA
[4, 2] ACCA
[2, 2] AGACCA
[1, 3] ATACA
[2, 1] ATTACA
[6, 1] CA
[6, 2] CA
[4, 3] CA
[5, 2] CCA
[3, 2] GACCA
[1, 1] GATTACA
[4, 1] TACA
[2, 3] TACA
[1, 2] TAGACCA
[3, 1] TTACA
```