Longest common subsequence problem

From Wikipedia, the free encyclopedia

Not to be confused with longest common substring problem.

The **longest common subsequence** (**LCS**) **problem** is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). It differs from problems of finding common substrings: unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences. The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the `diff` utility, and has applications in bioinformatics. It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files.

Complexity [edit]

For the general case of an arbitrary number of input sequences, the problem is NP-hard.[1] When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming (see *Solution* below). Assume you have N sequences of lengths $n_1, ..., n_N$. A naive search would test each of the 2^{n_1} subsequences of the first sequence to determine whether they are also subsequences of the remaining sequences; each subsequence may be tested in time linear in the lengths of the remaining sequences, so the time for this algorithm would be

$$O\left(2^{n_1}\sum_{i>1} n_i\right).$$

For the case of two sequences of *n* and *m* elements, the running time of the dynamic programming approach is $O(n \times m)$. For an arbitrary number of input sequences, the dynamic programming approach gives a solution in

$$O\left(N\prod_{i=1}^{N} n_i\right).$$

Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Беларуская
Español
Esperanto
فارسی
Français
한국어
Italiano
日本語
Polski
Русский
Српски / srpski
தமிழ்
中文
Edit links

There exist methods with lower complexity,[2] which often depend on the length of the LCS, the size of the alphabet, or both.

Notice that the LCS is not necessarily unique; for example the LCS of "ABC" and "ACB" is both "AB" and "AC". Indeed the LCS problem is often defined to be finding *all* common subsequences of a maximum length. This problem inherently has higher complexity, as the number of such subsequences is exponential in the worst case,[3] even for only two input strings.

Solution for two sequences [edit]

The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to high-level subproblems often reuse lower level subproblems. Problems with these two properties—optimal substructure and overlapping subproblems—can be approached by a problem-solving technique called dynamic programming, in which subproblem solutions are memoized rather than computed over and over. The procedure requires memoization—saving the solutions to one level of subproblem in a table (analogous to writing them to a *memo*, hence the name) so that the solutions are available to the next level of subproblems. This method is illustrated here.

Prefixes [edit]

The subproblems become simpler as the sequences become shorter. Shorter sequences are conveniently described using the term *prefix*. A prefix of a sequence is the sequence with the end cut off. Let S be the sequence (AGCA). Then, the sequence (AG) is one of the prefixes of S. Prefixes are denoted with the name of the sequence, followed by a subscript to indicate how many characters the prefix contains.[4] The prefix (AG) is denoted S_2, since it contains the first 2 elements of S. The possible prefixes of S are

S_1 = (A)
S_2 = (AG)
S_3 = (AGC)
S_4 = (AGCA).

The solution to the LCS problem for two arbitrary sequences, X and Y, amounts to construct some function, $LCS(X, Y)$, that gives the longest subsequences common to X and Y. That function relies on the following two properties.

First property [edit]

Suppose that two sequences both end in the same element. To find their LCS, shorten each sequence by removing the last element, find the LCS of the shortened sequences, and to that LCS append the removed element.

> For example, here are two sequences having the same last element: (BANANA) and (ATANA).
> Remove the same last element. Repeat the procedure until you find no common last element. The removed sequence will be (ANA).
> The sequences now under consideration: (BAN) and (AT)
> The LCS of these last two sequences is, by inspection, (A).
> Append the removed element, (ANA), giving (AANA), which, by inspection, is the LCS of the original sequences.

In general, for any sequences X and Y of length n and m, if we denote their elements x_1 to x_n and y_1 to y_m and their prefixes X_1 to X_{n-1} and Y_1 to Y_{m-1}, then we can say this:

> If: $x_n = y_m$
> then: $LCS(X_n, Y_m) = LCS(X_{n-1}, Y_{m-1})$ ^ x_n

where the caret ^ indicates that the following element, x_n, is appended to the sequence. Note that the LCS for X_n and Y_m involves determining the LCS of the shorter sequences, X_{n-1} and Y_{m-1}.

Second property [edit]

Suppose that the two sequences X and Y do not end in the same symbol. Then the LCS of X and Y is the longer of the two sequences $LCS(X_n, Y_{m-1})$ and $LCS(X_{n-1}, Y_m)$.

To understand this property, consider the two following sequences :

sequence X: ABCDEFG (n elements)

sequence Y: BCDGK (m elements)

The LCS of these two sequences either ends with a G (the last element of sequence X) or does not.

Case 1: the LCS ends with a G
Then it cannot end with a K. Thus it does not hurt to remove the K from sequence Y: if K were in the LCS, it would be its last character; as a consequence K is not in the LCS. We can then write: $LCS(X_n,Y_m) = LCS(X_n, Y_{m-1})$.

Case 2: the LCS does not end with a G
Then it does not hurt to remove the G from the sequence X (for the same reason as above). And then we can write: $LCS(X_n,Y_m) = LCS(X_{n-1}, Y_m)$.

In any case, the LCS we are looking for is one of $LCS(X_n, Y_{m-1})$ or $LCS(X_{n-1}, Y_m)$. Those two last LCS are both common subsequences to X and Y. LCS(X,Y) is the longest. Thus its value is the longest sequence of $LCS(X_n, Y_{m-1})$ and $LCS(X_{n-1}, Y_m)$.

LCS function defined [edit]

Let two sequences be defined as follows: $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$. The prefixes of X are $X_{1, 2, \ldots m}$; the prefixes of Y are $Y_{1, 2, \ldots n}$. Let $LCS(X_i, Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j. This set of sequences is given by the following.

$$LCS\left(X_i, Y_j\right) = \begin{cases} \emptyset & \text{if } \ i = 0 \text{ or } j = 0 \\ LCS\left(X_{i-1}, Y_{j-1}\right) \frown x_i & \text{if } x_i = y_j \\ \text{longest}\left(LCS\left(X_i, Y_{j-1}\right), LCS\left(X_{i-1}, Y_j\right)\right) & \text{if } x_i \neq y_j \end{cases}$$

To find the longest subsequences common to X_i and Y_j, compare the elements x_i and y_j. If they are equal, then the sequence $LCS(X_{i-1}, Y_{j-1})$ is extended by that element, x_i. If they are not equal, then the longer of the two sequences, $LCS(X_i, Y_{j-1})$, and $LCS(X_{i-1}, Y_j)$, is retained. (If they are both the same length, but not identical, then both are retained.) Notice that the subscripts are reduced by 1 in these formulas. That can result in a subscript of 0. Since the sequence elements are defined to start at 1, it was necessary to add the requirement that the LCS is empty when a subscript is zero.

Worked example [edit]

The longest subsequence common to C = (AGCAT), and R = (GAC) will be found. Because the *LCS* function uses a "zeroth" element, it is convenient to define zero prefixes that are empty for these sequences: $C_0 = \emptyset$; and $R_0 = \emptyset$. All the prefixes are placed in a table with C in the first row (making it a column header) and R in the first column (making it a row header).

LCS Strings

	Ø	**A**	**G**	**C**	**A**	**T**
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø					
A	Ø					
C	Ø					

This table is used to store the LCS sequence for each step of the calculation. The second column and second row have been filled in with Ø, because when an empty sequence is compared with a non-empty sequence, the longest common subsequence is always an empty sequence.

$LCS(R_1, C_1)$ is determined by comparing the first elements in each sequence. G and A are not the same, so this LCS gets (using the "second property") the longest of the two sequences, $LCS(R_1, C_0)$ and $LCS(R_0, C_1)$. According to the table, both of these are empty, so $LCS(R_1, C_1)$ is also empty, as shown in the table below. The arrows indicate that the sequence comes from both the cell above, $LCS(R_0, C_1)$ and the cell on the left, $LCS(R_1, C_0)$.

$LCS(R_1, C_2)$ is determined by comparing G and G. They match, so G is appended to the upper left sequence, $LCS(R_0, C_1)$, which is (Ø), giving (ØG), which is (G).

For $LCS(R_1, C_3)$, G and C do not match. The sequence above is empty; the one to the left contains one element, G. Selecting the longest of these, $LCS(R_1, C_3)$ is (G). The arrow points to the left, since that is the longest of the two sequences.

$LCS(R_1, C_4)$, likewise, is (G).

$LCS(R_1, C_5)$, likewise, is (G).

"G" Row Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø					
C	Ø					

For $LCS(R_2, C_1)$, A is compared with A. The two elements match, so A is appended to Ø, giving (A).

For $LCS(R_2, C_2)$, A and G do not match, so the longest of $LCS(R_1, C_2)$, which is (G), and $LCS(R_2, C_1)$, which is (A), is used. In this case, they each contain one element, so this LCS is given two subsequences: (A) and (G).

For $LCS(R_2, C_3)$, A does not match C. $LCS(R_2, C_2)$ contains sequences (A) and (G); $LCS(R_1, C_3)$ is (G), which is already contained in $LCS(R_2, C_2)$. The result is that $LCS(R_2, C_3)$ also contains the two subsequences, (A) and (G).

For $LCS(R_2, C_4)$, A matches A, which is appended to the upper left cell, giving (GA).

For $LCS(R_2, C_5)$, A does not match T. Comparing the two sequences, (GA) and (G), the longest is (GA), so $LCS(R_2, C_5)$ is (GA).

"G" & "A" Rows Completed

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	←↑(A) & (G)	←↑(A) & (G)	↖(GA)	←(GA)
C	Ø					

For $LCS(R_3, C_1)$, C and A do not match, so $LCS(R_3, C_1)$ gets the longest of the two sequences, (A).

For $LCS(R_3, C_2)$, C and G do not match. Both $LCS(R_3, C_1)$ and $LCS(R_2, C_2)$ have one element. The result is that $LCS(R_3, C_2)$ contains the two subsequences, (A) and (G).

For $LCS(R_3, C_3)$, C and C match, so C is appended to $LCS(R_2, C_2)$, which contains the two subsequences, (A) and (G), giving (AC) and (GC).

For $LCS(R_3, C_4)$, C and A do not match. Combining $LCS(R_3, C_3)$, which contains (AC) and (GC), and $LCS(R_2, C_4)$, which contains (GA), gives a total of three sequences: (AC), (GC), and (GA).

Finally, for $LCS(R_3, C_5)$, C and T do not match. The result is that $LCS(R_3, C_5)$ also contains the three sequences, (AC), (GC), and (GA).

Completed LCS Table

	Ø	A	G	C	A	T
Ø	Ø	Ø	Ø	Ø	Ø	Ø
G	Ø	←↑Ø	↖(G)	←(G)	←(G)	←(G)
A	Ø	↖(A)	←↑(A) & (G)	←↑(A) & (G)	↖(GA)	←(GA)
C	Ø	↑(A)	←↑(A) & (G)	↖(AC) & (GC)	←↑(AC) & (GC) & (GA)	←↑(AC) & (GC) & (GA)

The final result is that the last cell contains all the longest subsequences common to (AGCAT) and (GAC); these are (AC), (GC), and (GA). The table also shows the longest common subsequences for every possible pair of prefixes. For example, for (AGC) and (GA), the longest common subsequence are (A) and (G).

Traceback approach [edit]

Calculating the LCS of a row of the LCS table requires only the solutions to the current row and the previous row. Still, for long sequences, these sequences can get numerous and long, requiring a lot of storage space. Storage space can be saved by saving not the actual subsequences, but the length of the subsequence and the direction of the arrows, as in the table below.

Storing length, rather than sequences

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	←↑0	↖1	←1	←1	←1
A	0	↖1	←↑1	←↑1	↖2	←2
C	0	↑1	←↑1	↖2	←↑2	←↑2

The actual subsequences are deduced in a "traceback" procedure that follows the arrows backwards, starting from the last cell in the table. When the length decreases, the sequences must have had a common element. Several paths are possible when two arrows are shown in a cell. Below is the table for such an analysis, with numbers colored in cells where the length is about to decrease. The bold numbers trace out the sequence, (GA).[5]

Traceback example

	Ø	A	G	C	A	T
Ø	0	**0**	0	0	0	0
G	0	←↑0	↖1	←**1**	←1	←1
A	0	↖1	←↑1	←↑1	↖2	←**2**
C	0	↑1	←↑1	↖2	←↑2	←↑**2**

Relation to other problems [edit]

For two strings $X_{1\ldots m}$ and $Y_{1\ldots n}$, the length of the shortest common supersequence is related to the length of the LCS by[2]

$$|SCS(X,Y)| = n + m - |LCS(X,Y)|.$$

The edit distance when only insertion and deletion is allowed (no substitution), or when the cost of the substitution is the double of the cost of an insertion or deletion, is:

$$d'(X,Y) = n + m - 2 \cdot |LCS(X,Y)|.$$

Code for the dynamic programming solution [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(March 2013)*

Computing the length of the LCS [edit]

The function below takes as input sequences `X[1..m]` and `Y[1..n]` computes the LCS between `X[1..i]` and `Y[1..j]` for all `1 ≤ i ≤ m` and `1 ≤ j ≤ n`, and stores it in `C[i,j]`. `C[m,n]` will contain the length of the LCS of `X` and `Y`.

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
       C[i,0] = 0
    for j := 0..n
       C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

Alternatively, memoization could be used.

Reading out an LCS [edit]

The following function backtracks the choices taken when computing the `C` table. If the last characters in the prefixes are equal, they must be in an LCS. If not, check what gave the largest LCS of keeping x_i and y_j, and make the same choice. Just choose one if they were equally long. Call the function with `i=m` and `j=n`.

```
function backtrack(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i = 0 or j = 0
        return ""
    else if  X[i] = Y[j]
        return backtrack(C, X, Y, i-1, j-1) + X[i]
    else
        if C[i,j-1] > C[i-1,j]
            return backtrack(C, X, Y, i, j-1)
        else
            return backtrack(C, X, Y, i-1, j)
```

Reading out all LCSs [edit]

If choosing x_i and y_j would give an equally long result, read out both resulting subsequences. This is returned as a set by this function. Notice that this function is not polynomial, as it might branch in almost every step if the strings are similar.

```
function backtrackAll(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i = 0 or j = 0
        return {""}
    else if X[i] = Y[j]
        return {Z + X[i] for all Z in backtrackAll(C, X, Y, i-1, j-1)}
    else
        R := {}
        if C[i,j-1] ≥ C[i-1,j]
            R := R ∪ backtrackAll(C, X, Y, i, j-1)
        if C[i-1,j] ≥ C[i,j-1]
            R := R ∪ backtrackAll(C, X, Y, i-1, j)
        return R
```

Print the diff [edit]

This function will backtrack through the C matrix, and print the diff between the two sequences. Notice that you will get a different answer if you exchange `≥` and `<`, with `>` and `≤` below.

```
function printDiff(C[0..m,0..n], X[1..m], Y[1..n], i, j)
    if i > 0 and j > 0 and X[i] = Y[j]
        printDiff(C, X, Y, i-1, j-1)
        print "  " + X[i]
    else if j > 0 and (i = 0 or C[i,j-1] ≥ C[i-1,j])
        printDiff(C, X, Y, i, j-1)
        print "+ " + Y[j]
    else if i > 0 and (j = 0 or C[i,j-1] < C[i-1,j])
        printDiff(C, X, Y, i-1, j)
        print "- " + X[i]
    else
        print ""
```

Example [edit]

Let X be "`XMJYAUZ`" and Y be "`MZJAWXU`". The longest common subsequence between X and Y is "`MJAU`". The table `C` shown below, which is generated by the function `LCSLength`, shows the lengths of the longest common subsequences between prefixes of X and Y. The ith row and jth column shows the length of the LCS between $X_{1..i}$ and $Y_{1..j}$.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U

0	**Ø**	**0**	0	0	0	0	0	0	0
1	**X**	0	0	0	0	0	0	1	1
2	**M**	0	**1**	1	1	1	1	1	1
3	**J**	0	1	1	**2**	2	2	2	2
4	**Y**	0	1	1	2	2	2	2	2
5	**A**	0	1	1	2	**3**	3	3	3
6	**U**	0	1	1	2	3	3	3	**4**
7	**Z**	0	1	2	2	3	3	3	4

The highlighted numbers show the path the function `backtrack` would follow from the bottom right to the top left corner, when reading out an LCS. If the current symbols in X and Y are equal, they are part of the LCS, and we go both up and left (shown in **bold**). If not, we go up or left, depending on which cell has a higher number. This corresponds to either taking the LCS between $X_{1..i-1}$ and $Y_{1..j}$, or $X_{1..i}$ and $Y_{1..j-1}$.

Code optimization [edit]

Several optimizations can be made to the algorithm above to speed it up for real-world cases.

Reduce the problem set [edit]

The C matrix in the naive algorithm grows quadratically with the lengths of the sequences. For two 100-item sequences, a 10,000-item matrix would be needed, and 10,000 comparisons would need to be done. In most real-world cases, especially source code diffs and patches, the beginnings and ends of files rarely change, and almost certainly not both at the same time. If only a few items have changed in the middle of the sequence, the beginning and end can be eliminated. This reduces not only the memory requirements for the matrix, but also the number of comparisons that must be done.

```
function LCS(X[1..m], Y[1..n])
    start := 1
    m_end := m
    n_end := n
    trim off the matching items at the beginning
    while start ≤ m_end and start ≤ n_end and X[start] = Y[start]
        start := start + 1
    trim off the matching items at the end
    while start ≤ m_end and start ≤ n_end and X[m_end] = Y[n_end]
        m_end := m_end - 1
        n_end := n_end - 1
    C = array(start-1..m_end, start-1..n_end)
    only loop over the items that have changed
    for i := start..m_end
        for j := start..n_end
            the algorithm continues as before ...
```

In the best-case scenario, a sequence with no changes, this optimization would completely eliminate the need for the C matrix. In the worst-case scenario, a change to the very first and last items in the sequence, only two additional comparisons are performed.

Reduce the comparison time [edit]

Most of the time taken by the naive algorithm is spent performing comparisons between items in the sequences. For textual sequences such as source code, you want to view lines as the sequence elements instead of single characters. This can mean comparisons of relatively long strings for each step in the algorithm. Two optimizations can be made that can help to reduce the time these comparisons consume.

Reduce strings to hashes [edit]

A hash function or checksum can be used to reduce the size of the strings in the sequences. That is, for source code where the average line is 60 or more characters long, the hash or checksum for that line might be only 8 to 40 characters long. Additionally, the randomized nature of hashes and checksums would guarantee that comparisons would short-circuit faster, as lines of source code will rarely be changed at the beginning.

There are three primary drawbacks to this optimization. First, an amount of time needs to be spent beforehand

to precompute the hashes for the two sequences. Second, additional memory needs to be allocated for the new hashed sequences. However, in comparison to the naive algorithm used here, both of these drawbacks are relatively minimal.

The third drawback is that of collisions. Since the checksum or hash is not guaranteed to be unique, there is a small chance that two different items could be reduced to the same hash. This is unlikely in source code, but it is possible. A cryptographic hash would therefore be far better suited for this optimization, as its entropy is going to be significantly greater than that of a simple checksum. However, the benefits may not be worth the setup and computational requirements of a cryptographic hash for small sequence lengths.

Reduce the required space [edit]

If only the length of the LCS is required, the matrix can be reduced to a $2 \times \min(n, m)$ matrix with ease, or to a $\min(m, n) + 1$ vector (smarter) as the dynamic programming approach only needs the current and previous columns of the matrix. Hirschberg's algorithm allows the construction of the optimal sequence itself in the same quadratic time and linear space bounds.[6]

Further optimized algorithms [edit]

Several algorithms exist that are worst-case faster than the presented dynamic programming approach.[7] For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.[8] For r (and $n > m$), the number of matches between the two sequences, there is an algorithm that performs in $O((n + r)\log(n))$ time.[9]

Behavior on random strings [edit]

Main article: Chvátal–Sankoff constants

Beginning with Chvátal & Sankoff (1975),[10] a number of researchers have investigated the behavior of the longest common subsequence length when the two given strings are drawn randomly from the same alphabet. When the alphabet size is constant, the expected length of the LCS is proportional to the length of the two strings, and the constants of proportionality (depending on alphabet size) are known as the Chvátal–Sankoff constants. Their exact values are not known, but upper and lower bounds on their values have been proven,[11] and it is known that they grow inversely proportionally to the square root of the alphabet size.[12] Simplified mathematical models of the longest common subsequence problem have been shown to be controlled by the Tracy–Widom distribution.[13]

See also [edit]

- Longest increasing subsequence
- Longest alternating subsequence
- Levenshtein distance

References [edit]

1. ^ David Maier (1978). "The Complexity of Some Problems on Subsequences and Supersequences". *J. ACM* (ACM Press) **25** (2): 322–336. doi:10.1145/322063.322075.
2. ^ [a] [b] L. Bergroth and H. Hakonen and T. Raita (2000). "A Survey of Longest Common Subsequence Algorithms". *SPIRE* (IEEE Computer Society) **00**: 39–48. doi:10.1109/SPIRE.2000.878178. ISBN 0-7695-0746-8.
3. ^ Ronald I. Greenberg (2003-08-06). "Bounds on the Number of Longest Common Subsequences". arXiv:cs.DM/0301030.
4. ^ Xia, Xuhua (2007). *Bioinformatics and the Cell: Modern Computational Approaches in Genomics, Proteomics and Transcriptomics*. New York: Springer. p. 24. ISBN 0-387-71336-0.
5. ^ Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2001). "15.4". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. pp. 350–355. ISBN 0-262-53196-8.
6. ^ Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences". *Communications of the ACM* **18** (6): 341–343. doi:10.1145/360825.360861.
7. ^ http://books.google.com/books?id=mFd_grFyiT4C&pg=PA132&lpg=PA132&dq=hunt+szymanski+algorithm&source=bl&ots=sMc-HtvNTQ&sig=FtrZ_b5JdJ25lghwc1-XOfysaf8&hl=en&sa=X&ei=-BU9VPK7OpS7ggT0gYEQ&ved=0CDsQ6AEwAw#v=onepage&q&f=false
8. ^ Masek, William J.; Paterson, Michael S. (1980), "A faster algorithm computing string edit distances", *Journal of Computer and System Sciences* **20** (1): 18–31, doi:10.1016/0022-0000(80)90002-1, MR 566639.
9. ^ http://www.cs.bgu.ac.il/~dpaa111/wiki.files/HuntSzymanski.pdf
10. ^ Chvatal, Václav; Sankoff, David (1975), "Longest common subsequences of two random sequences", *Journal of*

Applied Probability **12**: 306–315, doi:10.2307/3212444, MR 0405531.

11. ^ Lueker, George S. (2009), "Improved bounds on the average length of longest common subsequences", *Journal of the ACM* **56** (3), A17, doi:10.1145/1516512.1516519, MR 2536132.
12. ^ Kiwi, Marcos; Loebl, Martin; Matoušek, Jiří (2005), "Expected length of the longest common subsequence for large alphabets", *Advances in Mathematics* **197** (2): 480–498, doi:10.1016/j.aim.2004.10.012, MR 2173842.
13. ^ Majumdar, Satya N.; Nechaev, Sergei (2005), "Exact asymptotic results for the Bernoulli matching model of sequence alignment", *Physical Review E* **72** (2): 020901, 4, doi:10.1103/PhysRevE.72.020901, MR 2177365.

External links [edit]

- Dictionary of Algorithms and Data Structures: longest common subsequence
- A collection of implementations of the longest common subsequence in many programming languages

The Wikibook *Algorithm implementation* has a page on the topic of: ***Longest common subsequence***

Categories: Problems on strings | Combinatorics | Dynamic programming | Polynomial-time problems | NP-complete problems