1. Longest Common Prefix(LCP)

Το prefix ενός string $T= t_1...t_n$ είναι ένα string $T'= t_1...t_m$ όπου $m \leq n$. Ένα prefix δεν πρέπει να είναι το string το ίδιο δηλαδή $0 \leq m < n$.
Στο παρακάτω παράδειγμα ένα prefix του string “apple” είναι το string “ap”:

a p p l e
| |
a p

Το Longest Common Prefix δύο ή περισσοτέρων string είναι το μεγαλύτερο string το οποίο είναι κοινό prefix για όλα τα string. Εισήχθη το 1993 από τους Udi Manber και Gene Myers ώστε σε συνδυασμό με το suffix array να βελτιώσουν τον string search αλγόριθμό τους. Το 2003 οι Kärkkäinen και Sanders παρουσίασαν έναν από τους πρώτους γραμμικούς αλγορίθμους για τον υπολογισμό του suffix array και έδειξαν ότι τροποποιώντας τον αλγόριθμό τους μπορούσαν να υπολογίσουν το LCP. Ο Fischer το 2001 περιέγραψε τον γρηγορότερο μέχρι σήμερα αλγόριθμο γραμμικού χρόνου για υπολογισμό του LCP, ο οποίος βασίστηκε σε έναν από τους γρηγορότερους αλγορίθμους γραμμικού χρόνου για κατασκευή suffix array των Nong, Zhang και Chan (2009).
Ο πρώτος γραμμικός αλγόριθμος υπολογισμού LCP περιγράφηκε από τον Kasai (2001), ενώ αργότερα η μέθοδος του Kärkkäinen για υπολογισμό του LCP από τον υπολογισμό του Permuted Longest Common Prefix (PLCP), αποδείχθηκε πιο αποδοτική σε σχέση με του Kasai. Επίσης οι Gog και Ohlebusch παρουσίασαν έναν αλγόριθμο που δέχεται ως είσοδο τον μετασχηματισμό Burrows-Wheeler και υπολογίζει το LCP μειώνοντας τις απαιτήσεις σε μνήμη.
Σχετικά με παράλληλους αλγορίθμους υπάρχει ο αλγόριθμος των Kärkkäinen και Sanders που αναφέρθηκε παραπάνω και υπολογίζει το suffix array εκτός από το LCP, ενώ οι Deo και Kelly παρουσίασαν ένα παράλληλο αλγόριθμο για GPU’s που βασίζεται στην παραλληλοποίηση του αλγορίθμου του Kasai.
Το LCP σε συνδυασμό με το suffix array προτιμώνται σε σχέση με το suffix tree για ευρετήριο κειμένου λόγω των χαμηλών του απαιτήσεων σε μνήμη. Επίσης ο συνδυασμός αυτός χρησιμοποιείται για αποδοτικό pattern matching ενώ βρίσκει και πολλές άλλες εφαρμογές σε προβλήματα δέντρων, όπως η διάβαση από κάτω προς τα πάνω του πλήρους δέντρου επιθεμάτων ή η διάβαση του suffix δέντρου με χρήση suffix συνδέσμων, που με τη σειρά τους δίνουν λύσεις σε διάφορα προβλήματα επεξεργασίας string. Επίπροσθετα οι παρπάνω μέθοδοι διέλευσης δέντρων βρίσκουν εφαρμογές στην βιοπληροφορική καθιστώντας τον συνδυασμο δέντρου suffix-LCP μία από τις πιο σημαντικές δομές για την ανάλυση του γονιδιώματος και για την συγκριτική γονιδιωματική. Ο Kasai έδειξε ότι μια προσομοίωση της διέλευσης από κάτω προς τα πάνω του suffix tree είναι δυνατή χρησιμοποιώντας μόνο το suffix array και το LCP array. Οι Abouelhoda, Kurtz και Ohlebusch ενισχύνοντας την μέθοδο του Kasai με επιπλέον δομές δεδομένων περιέγραψαν πως το ενισχυμένο αυτό suffix array μπορεί να προσομοιώσει και τα τρία είδη διελεύσεων suffix δέντρων που αναφέρθηκαν πριν. Οι Fischer και Heun (2007) βελτίωσαν περαιτέρω το ενισχυμένο suffix δέντρο κάνοντας επεξεργασία του πίνακα των LCP και μείωσαν τις ανάγκες για χώρο μνήμης, δείχνοντας ότι κάθε πρόβλημα που λύνεται με αλγορίθμους suffix δέντρων μπορεί επίσης να λυθεί με χρήση του ενισχυμένου suffix array που κάνει χρήση και των LCP αλγορίθμων. Πολλές φορές το suffix array θα είναι διαθέσιμο οπότε είναι ωφέλιμο να υπάρχουν αποδοτικοί αλγόριθμοι για τον υπολογισμό αποκλειστικά του LCP. Επιπλέον με την μεγάλη αύξηση του μεγέθους των δεδομένων που θέλουμε να επεξεργαστούμε, η ύπαρξη γρήγορων αλγορίθμων υπολογισμού του LCP είναι πολύ σημαντική.

Ωστόσο υπάρχουν πολλές εφαρμογές του LCP χωρίς την ανάγκη για χρήση και του suffix array. Μπορεί να μας παρέχει με σημαντικές πληροφορίες σχετικά με την επαναληψιμότητα σε ένα δοσμένο string οπότε είναι μια χρήσιμη δομή πληροφορίας για την ανάλυση δεδομένων χαρακτήρων στη μοριακή βιολογία ή τον εντοπισμό πιθανών διαφοροποιήσεων σε ακολουθίες που μπορεί να είναι π.χ. αποτέλεσμα της αντιγραφής του DNA ή σφαλμάτων αλληλουχίας στο DNA.

Παρακάτω θα εξετάσουμε την απόδοση κάποιων αλγορίθμων στην επίλυση του προβλήματος εύρεσης του LCP, ανάμεσα σε δύο ή περισσότερες ακολουθίες DNA. Οι αλγόριθμοι που θα χρησιμοποιήσουμε είναι οι παρακάτω:

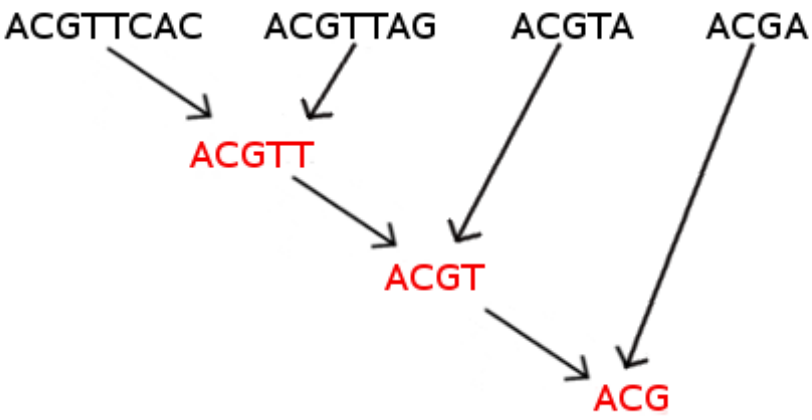
- Word by Word Matching
- Character by Character Matching
- Divide and Conquer
- Binary Search

1.1. Word by Word Matching

Ας υποθέσουμε ότι εχουμε τις παρακάτω ακολουθίες DNA, “ACGTTCAC” και “ACGTTAG”. Το μέγιστο κοινό prefix(LCP) αυτών των δύο ακολουθιών είναι η ακολουθία “ACGTT”. Έστω ότι εισάγουμε άλλη μία ακολουθία την “ACGTA”. Πλέον το LCP των τριών ακολουθιών είναι το “ACGT”.

Η διαδικασία που ακολουθήσαμε είναι η παρακάτω: Βρήκαμε το LCP για τις δύο πρώτες ακολουθίες και μετά βρήκαμε το LCP ανάμεσα στο LCP που βρήκαμε πριν και το νέο string. Η σχέση που προκύπτει είναι η εξής:

LCP(s1, s2, s3) = LCP(LCP(s1,s2), s3)



Σχήμα 1.1: Σχηματική απεικόνιση της διαδικασίας που ακολουθείται στην εκτέλεση του αλγορίθμου word-by-word

Γενικεύοντας αυτή την ιδέα ο αλγόριθμος θα δουλεύει ως εξής: θα διατρέχει τα strings $[s_1...s_n]$ βρίσκοντας σε κάθε διέλευση i το LCP των string $[s_1...s_i]$. Αν το LCP$(s_1...s_i)$ είναι κενό, τότε ο αλγόριθμος σταματάει. Διαφορετικά μετά από n διελεύσεις ο αλγόριθμος επιστρέφει το LCP$(s_1...s_n)$. Η σχέση που χρησιμοποιήσαμε είναι:

LCP$(s_1...s_n)$=LCP(LCP(LCP(s_1,s_2),s_3),...s_n)

Χρησιμοποιώντας αυτόν τον αλγόριθμο μπορούμε να βρούμε το LCP των δοσμένων ακολουθιών. Κάθε φορά θα υπολογίζεται το LCP της νέας ακολουθίας συγκρίνοντας το με το LCP που έχει προκύψει μέχρι εκείνη τη στιγμή. Το τελικό LCP θα είναι το LCP όλων των ακολουθιών.

Πολυπλοκότητα χρόνου: Αν n ο αριθμός των string και m το μήκος του μεγαλύτερου string η χρονική πολυπλοκότητα του αλγορίθομου θα είναι O(n*m), καθώς διατρέχουμε όλα τα string και όλους τους χαρακτήρες για το καθένα.

1.2. Character by Character Matching

Έστω ότι το τελευταίο string από αυτά που θέλουμε εξετάσουμε βρίσκεται στο τέλος του συνόλου των strings. Η προηγούμενη προσέγγιση θα εξετάσει όλα τα string βρίσκοντας πιθανώς πολλές φορές LCPs ανάμεσα στα strigs με μέγεθος μεγαλύτερο του τελευταίου, ενώ είναι προφανές ότι το τελικό LCP πρέπει να είναι ίσο ή μικρότερο του μικρότερου string. Για την αντιμετώπιση αυτή της περίπτωσης όπου θα εκτελεστούν αχρείαστοι υπολογισμοί μία μέθοδος είναι η “κάθετη” εξέταση των strings. Σε αυτόν τον αλγόριθμο αντί να εξετάσουμε κάθε ακολουθία ξεχωριστά, θα εξετάσουμε τους χαρακτήρες των ακολουθιών έναν προς έναν για μία στήλη, πριν προχωρήσουμε στην επόμενη. Χρησιμοποιώντας τις ίδιες ακολουθίες με πριν δηλαδή: “ACGTTCAC”, “ACGTTAG”, “ACGTA”, “ACGA θα ξεκινήσουμε από τον χαρακτήρα στην πρώτη θέση όλων των string οπότε:

A	C	G	T	T	C	A	C
A	C	G	T	T	A	G	
A	C	G	T	A			
A	C	G	A				

Πίνακας 1.1

Όλοι οι χαρακτήρες στο πρώτο πέρασμα είναι “A” άρα το “A” μπαίνει στο prefix που θέλουμε.
‘Επειτα:

A	C	G	T	T	C	A	C
A	C	G	T	T	A	G	
A	C	G	T	A			
A	C	G	A				

Πίνακας 1.2

Στο δεύτερο πέρασμα ο χαρακτήρας για τις ακολουθίες στην δεύτερη θέση είναι ξανά ο ίδιος και συγκεκριμένα το “C”. Άρα το “C” μπαίνει και αυτό στο prefix, το οποίο πλέον είναι: “AC”.
Η ίδια διαδικασία ξανά:

A	C	G	T	T	C	A	C
A	C	G	T	T	A	G	
A	C	G	T	A			
A	C	G	A				

Πίνακας 1.3

Στο τρίτο πέρασμα ο χαρακτήρας είναι πάλι κοινός για όλες τις ακολουθίες άρα θα αποτελέσει και αυτός μέρος του prefix. Το prefix μέχρι τώρα είναι το “ACG”. Προχωράμε στην επόμενη θέση:

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

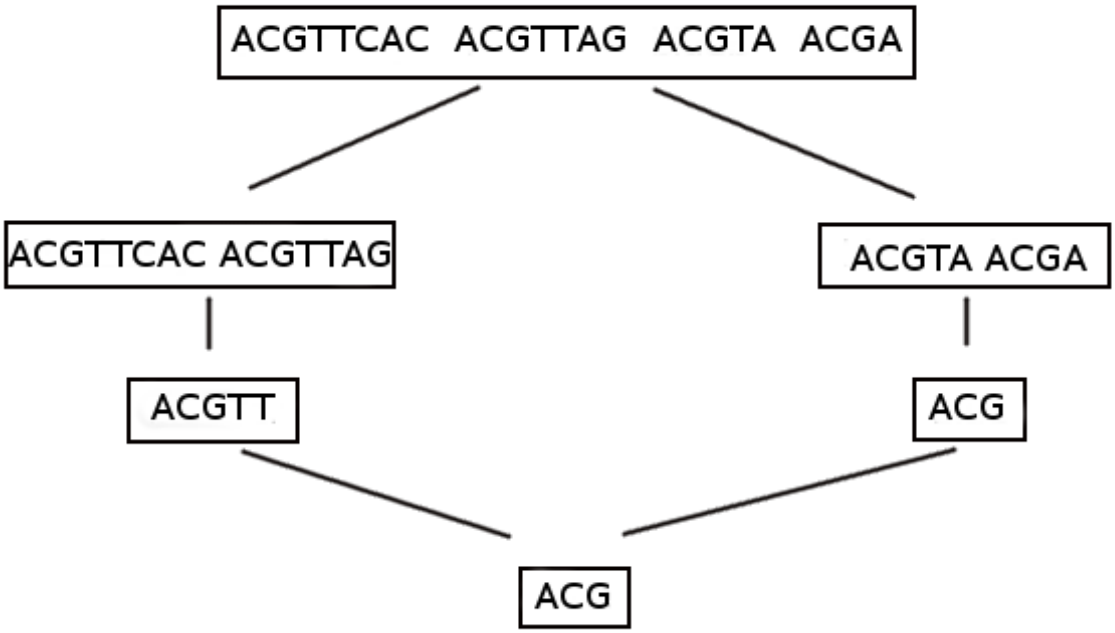
Πίνακας 1.4

Αυτή την φορά παρατηρούμε ότι ο χαρακτήρας δεν είναι ο ίδιος και για τις τρεις ακολουθίες καθώς ενώ για τις τρεις πρώτες είναι το “T”, για την τελευταία είναι ο χαρακτήρας “A”. Άρα ως prefix παραμένει η μέχρι τώρα ακολουθία δηλαδή το “ACG”.

Πολυπλοκότητα χρόνου: Η πολυπλοκότητα θα είναι και εδώ O(n*m) αφού διατρέχουμε όλα τα string και όλους τους χαρακτήρες. Όμως αυτό ο αλγόριθμος προσφέρει βελτίωση σε σχέση με τον “word by word”, καθώς στην περίπτωση που δεν υπάρχει κοινό prefix ανάμεσα στα string στον προηγούμενο αλγόριθμο έπρεπε να ψάξουμε όλα τα strings για να το διαπιστώσουμε. Αντίθετα σε αυτόν εδώ τον αλγόριθμο στο πρώτο πέρασμα για τον πρώτο χαρακτήρα του κάθε string, μόλις φτάσουμε στο string το οποίο δεν έχει κοινό χαρακτήρα με τα άλλα string θα ξέρουμε ότι δεν υπάρχει κοινό prefix και δεν θα χρειαστεί να συνεχίσουμε την αναζήτηση. Αυτό είναι ένα μεγάλο πλεονέκτημα όταν τα string είναι πολλά και κάνει πολύ πιο εύκολη την εύρεση του prefix.

1.3. Divide and conquer

Μια διαφορετική προσέγγιση είναι η χρήση του αλγορίθμου divide and conquer για τον υπολογισμό του prefix. Στην επιστήμη της πληροφορικής ο αλγόριθμος αυτός είναι μια γενικότερη προσέγγιση στην οποία βασίζονται διάφοροι χρήσιμοι αλγόριθμοι όπως ο αλγόριθμος binary search που θα αναλυθεί αργότερα. Η ιδέα του αλγορίθμου προέρχεται από την προσεταιριστική ιδιότητα της λειτουργίας του LCP. Παρατηρούμε ότι: $LCP(s_1...s_n) = LCP(LCP(s_1...s_k, LCP(s_{k+1}...s_n))$, όπου $1<k<n$.
Για να εφαρμόσουμε αυτή την παρατήρηση θα χρησιμοποιήσουμε την τεχνική του διαίρει και βασίλευε. Ο αλγόριθμος ξεκινάει χωρίζοντας το σύνολο των strings που δίνουμε σαν είσοδο σε δύο μέρη, δηλαδή από το αρχικό πρόβλημα LCP(si...sj) θα προκύψουν δύο υποπροβλήματα LCP(si...sm) και LCP(sm...sj), όπου m= (i+j)/2. Έπειτα κάνουμε το ίδιο για τα δύο επιμέρους τμήματα. Η ίδια διαδικασία επαναλαμβάνεται μέχρι όλα τα σύνολα να έχουν μήκος 1. Αφού έγινε η διαίρεση θα αρχίσει η “κατάκτηση” επιστρέφοντας κάθε φορά το κοινό prefix των strings του αριστερού και των strings του δεξιού μέρους. Από τις λύσεις του αριστερού και του δεξιού μέρους θα προκύψει η το τελική λύση $LCP(s_i...s_j)$, αρκεί να υπολογιστεί το κοινό prefix των δύο μερών



Σχήμα 1.2: Σχηματική απεικόνιση των βημάτων που ακολουθούνται κατά την εκτέλεση του αλγορίθμου divide and conquer

Πολυπλοκότητα χρόνου: Και εδώ εξετάζουμε όλα τα strings και όλους τους χαρακτήρες τους οπότε η πολυπλοκότητα θα είναι O(N*M). Ο αλγόριθμος αυτός είναι γρηγορότερος από τον word by word αλλά χειρότερος από τον character by character καθώς θα εξετάσει όλους τους χαρακτήρες των string μέχρι να βρεθεί το prefix, ειδικά στην περίπτωση που

δεν υπάρχει prefix η διαφορά στην ταχύτητα είναι μεγαλύτερη για τον ίδιο λόγο που αναφέρθηκε και πριν στην ανάλυση του character by character.

1.4. Binary Search

Στον αλγόριθμο αυτό θα γίνει εφαρμογή της μεθόδου δυαδικής αναζήτησης για την εύρεση του LCP. Στην δυαδική αναζήτηση ο αλγόριθμος συγκρίνει το στοιχείο που δίνεται ως είσοδος με την τιμή του μεσαίου στοιχείου της διάταξης. Αν δεν είναι ίσα το μισό στο οποίο η είσοδος δεν μπορεί να βρίσκεται αφαιρείται και η αναζήτηση συνεχίζεται στο άλλο μισό επαναλαμβάνοντας την ίδια διαδικασία, μέχρι να βρεθεί η επιθυμητή διάταξη. Σε αυτή την εφαρμογή κάθε φορά ο χώρος αναζήτησης ο οποίος έχει το μήκος του μικρότερου string(μπορούμε να επιλέξουμε να είναι το μέρος οποιουδήποτε string), θα χωρίζεται στην μέση και θα απορρίπτεται το μέρος που δεν περιέχει την τελική λύση ώστε να εξεταστεί μόνο το άλλο. Έστω το string που θα επιλέξουμε ότι είναι το s_1 τότε ο χώρος αναζήτησης θα είναι το $s_1[0...minlen]$ και οι δύο περιπτώσεις θα είναι: το $s_1[0...mid]$ να είναι κοινό string ανάμεσα στα strings του συνόλου. Αυτό σημαίνει ότι για οποιοδήποτε $i \leq mid$ το $s_1[0...i]$ είναι κοινό string οπότε το αφαιρούμε από χώρο αναζήτησης και αρχίζουμε να ψάχνουμε για μεγαλύτερο LCP στο δεύτερο μέρος. Η δεύτερη περίπτωση είναι το $s_1[0...mid]$ να μην είναι κοινό string οπότε για οποιοδήποτε $i>mid$ το $s_1[0...i]$ δεν είναι κοινό string οπότε αφαιρούμε το δεύτερο μέρος από τον χώρο αναζήτησης. Στην υλοποίηση που πραγματοποιήθηκε, στην πρώτη περίπτωση η αναζήτηση στο δεύτερο μέρος πραγματοποιείται με τον character by character, αλγόριθμο που περιγράφηκε πιο πριν, καθώς δίνει καλύτερα αποτελέσματα από να συνεχίζαμε την αναζήτηση με binary search.

Η εκτέλεση του αλγορίθμου σε βήματα:

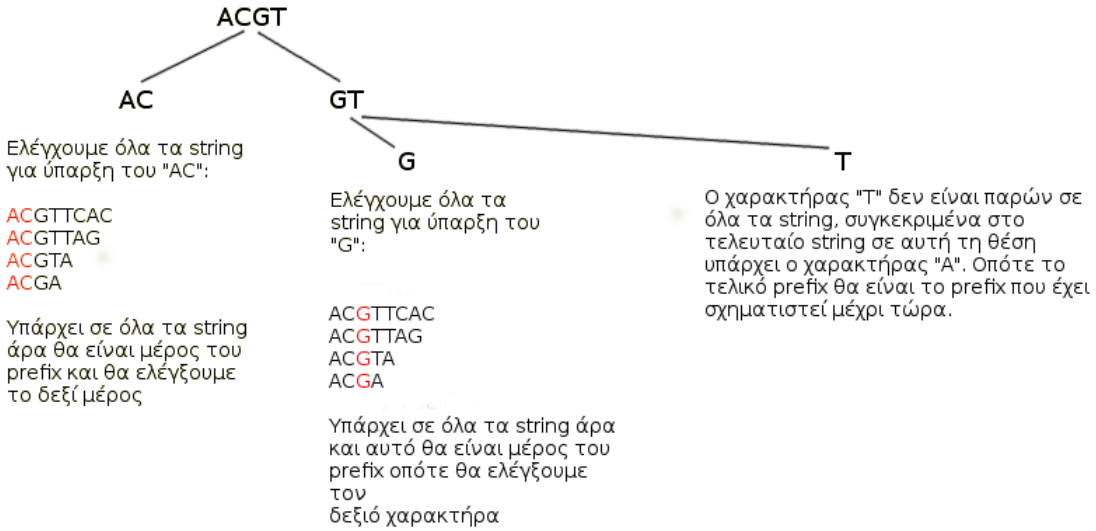
1. Βρίσκουμε το string με το ελάχιστο μέγεθος. Η τιμή του περνάει σε μια μεταβλητή L.

2. Εκτελείται δυαδική αναζήτηση σε οποιοδήποτε από τα strings του συνόλου(στην υλοποίηση επιλέγουμε το πρώτο string) για τις θέσεις από 0 έως L-1.

3. Αρχικά παίρνουμε το low=0 και το high=L-1 και χωρίζουμε το string σε δύο μέρη, το αριστερό(low έως mid) και το δεξιό(mid+1 έως high)

4. Ελέγχουμε αν όλοι οι χαρακτήρες στο αριστερό μισό είναι ίδιοι για τις ίδιες θέσεις για όλα τα υπόλοιπα strings του συνόλου. Εάν είναι τότε το αριστερό μέρος θα είναι μέρος του prefix που θέλουμε και θα συνεχίσουμε τον έλεγχο στο δεξιό μέρος για να διαπιστώσουμε αν υπάρχει μεγαλύτερο prefix.

5. Εάν δεν είναι, τότε δεν χρειάζεται να γίνει έλεγχος του δεξιού μέρους αφού υπάρχουν χαρακτήρες στο αριστερό μέρος που δεν είναι μέρος του prefix.



Σχήμα 1.3: Στο παραπάνω σχήμα απεικονίζονται τα βήματα που ακολουθούνται κατά την εκτέλεση του αλγορίθμου Binary Search

Πολυπλοκότητα χρόνου: Προκύπτει από την επαναληπτική σχέση T(m) = T(m/2) + O(m*n) όπου m το μήκος του μεγαλύτερου string. Ο αλγόριθμος αυτός είναι ο γρηγορότερος από όσους αναφέρθηκαν καθώς εξετάζει τους χαρακτήρες που βρίσκονται μέσα στα όρια που καθορίζονται από την αρχή του string μέχρι το μήκος του μικρότερου string. Επίσης αφού διαιρέσει περαιτέρω αυτό το substring στην μέση, αν το πρώτο μισό δεν είναι ολόκληρο μέρος του prefix να μην χρειαστεί να κάνει αναζήτηση στο άλλο μισό.

2. Μέγιστη Κοινή Υποσυμβολοσειρά

Το πρόβλημα εξαγωγής μέγιστης κοινής υποσυμβολοσειράς(Longest Common Substring-LCS problem) είναι το κλασικό πρόβλημα εύρεσης της μέγιστης υποσυμβολοσειράς η οποία είναι κοινή για όλα τα string σε ένα σύνολο από strings, συχνά πρόκειται για δύο strings. Δεν πρέπει να συγχέεται με το longest common subsequence πρόβλημα(θα αναφερθούμε σε αυτό σε επόμενο κεφάλαιο) όπου ψάχνουμε την μέγιστη κοινή υποακολουθία ανάμεσα σε κάποια string, η οποία δεν απαιτείται να καταλαμβάνει συνεχόμενες θέσεις στα αρχικά string. Η κατανόηση του πόσο όμοιες είναι δύο συμβολοσειρές και τι έχουν κοινό είναι ένα βασικό πρόβλημα στην μελέτη των string.
Το LCS πρόβλημα έχει μελετηθεί και αναλυθεί εκτενώς από τον Gusfield(1997).
Το 1970 ο Don Knuth έκανε την εικασία ότι είναι αδύνατο να κατασκευαστεί ένας αλγόριθμος γραμμικού χρόνου για την επίλυση του προβλήματος. Ωστόσο κάποια χρόνια αργότερα διατυπώθηκε ο πρώτος αλγόριθμος γραμμικου χρόνου με την χρήση ενός γενικευμένου suffix tree με βάση τις προτάσεις των Gusfield, Hui και Weiner. Η κλασική λύση επίλυσης του LCS βασίζεται σε δύο παρατηρήσεις. Η μία είναι ότι το LCS δύο string x και y είναι το LCP μερικών suffix του x και μερικών suffix του y. Η δεύτερη παρατήρηση είναι ότι το μέγιστο μήκος του LCP μεταξύ ενός suffix του x και suffixes των του y προσεγγίζεται στα δύο suffixes του y τα οποία είναι πιο κοντά στο x σύμφωνα με την λεξικογραφική σειρά.
Στην βιολογία το πρόβλημα εύρεσης substrings κοινά σε δοσμένα strings(DNA, RNA ή πρωτεΐνες) εμφανίζεται σε πολλές διαφορετικές περιπτώσεις. Ειδικά σε τομείς της μοριακής βιολογίας είναι ένα πρόβλημα που απασχολεί έντονα. Η αιτία του προβλήματος είναι ότι οι μεταλλάξεις που συμβαίνουν στο DNA όταν δύο είδη διαφοροποιούνται αλλάζουν πιο γρήγορα τα μέρη του DNA ή των πρωτεϊνών τα οποία είναι λιγότερο σημαντικά για την λειτουργία τους. Αντιθέτως τα μέρη του DNA ή των πρωτεϊνων τα οποία είναι κρίσιμα για την λειτουργία ενός μορίου διατηρούνται περισσότερο αναλοίωτα, καθώς οι μεταλλάξεις που συμβαίνουν σε αυτές τις περιοχές έχουν περισσότερες πιθανότητες να είναι θανατηφόρες. Μία άλλη εμφάνιση του προβλήματος είναι όταν εμφανίζεται σαν υποπρόβλημα πολλών ευρετικών μεθόδων που αναπτύσσονται στην βιβλιογραφία της βιολογίας για την ευθυγράμμιση string, το πρόβλημα δηλαδή της πολλαπλής ευθυγράμμισης ή για την σύγκριση βιολογικών ακολουθιών. Επιπλέον μία παραλλαγή της προσέγγισης που χρησιμοποιούμε για το LCS πρόβλημα είναι η προσέγγιση για την επίλυση του προβλήματος της μόλυνσης του DNA στο εργαστήριο. Η μόλυνση του DNA είναι ένα σύνηθες πρόβλημα στην βιοτεχνολογία όπου πρέπει να διαπιστωθεί αν σε ένα δείγμα DNA έχει μολυνθεί από ανεπιθύμητο DNA αναζητώντας για LCS ανάμεσα στο μολυσμένο δείγμα και τους πιθανούς μολυντές. Μία ακόμη σημαντική εφαρμογή του είναι η εύρεση ιδιαίτερα σημαντικών περιοχών σε μια DNA ακολουθία, περιοχές που περιέχουν γονίδια σημαντικά για την επιβίωση του οργανισμού. Σε αυτή την περίπτωση πρώτα εξετάζεται το DNA διάφορων συγγενών οργανισμών με σκοπό να διαπιστωθεί αν ταυτίζονται ή μοιάζουν ιδιαίτερα στις περιοχές που είναι απαραίτητες για την επιβίωσή τους και στις οποίες συμβαίνουν σημαντικά λιγότερες μεταλλάξεις.
Κάποιες παραλλαγές και γενικεύσεις του προβλήματος που επίσης βρίσκουν πολλές πρακτικές εφαρμογές σε διάφορους τομείς, συμπεραλαμβανομένου και της βιοπληροφορικής, είναι: το LCS with k-mismatches, το Longest Common Increasing Substring, το k-Common Repeated Substring problem και αλλες ευρετικές με βάση το LCS.
Παρακάτω θα εξεταστούν διάφοροι αλγόριθμοι επίλυσης του προβλήματος και η απόδοσή τους. Οι αλγόριθμοι αυτοί εκτελούν αναζήτηση για LCS ανάμεσα σε δύο strings, ως είσοδος δόθηκαν βιολογικά δεδομένα και είναι οι εξής:

- Naive Search
- Dynamic
- Suffix Tree
- Suffix Array

2.1. Naive Search

Έχοντας ως είσοδο δύο strings x και y, αρχικά βρίσκουμε όλα τα δυνατά substrings του μικρότερου από τα δύο και ψάχνουμε με την χρήση ενός αλγορίθμου pattern searching διαδοχικά για κάθε substring αν είναι και substring του άλλου string. Ο pattern searching αλγόριθμος που χρησιμοποιήθηκε εδώ ήταν ο Knuth Morris Pratt. Ο KMP για δοσμένο κείμενο txt[0...n-1] και pattern pat[0...m-1], όπου n>m, βρίσκει τις εμφανίσεις του pat στο txt και αποδίδει καλύτερα όταν το αλφάβητο των strings είναι μικρό, όπως και στην περίπτωσή μας όπου έχουμε ως αλφάβητο τις τέσσερις βάσεις A,C,G,T. Κάθε φορά ελέγχεται το μέγεθος του νέου substring που προκύπτει. Αν είναι μεγαλύτερο από το προηγούμενο θα παίρνει την θέση του LCS, μετά ελέγχεται το επόμενο κοκ ώστε από τα κοινά substrings να επιλεχθεί το μεγαλύτερο για να έχουμε το επιθυμητό αποτέλεσμα.

π.χ. Το LCS των string “ACGTTCAC” και “GATCGTG” με βάση τον παραπάνω αλγόριθμο θα βρεθεί ως εξής:

- Βρίσκουμε τα substrings του string “ACGTTCAC” δηλαδή “A”, “AC”, “ACG”, “ACGT”, “ACGTT”, “ACGTTC” κτλ
- Αναζητούμε τα substrings στο “GATCGTG” με χρήση του αλγορίθμου KMP
- Κάθε φορά που κάποιο από τα substrings του “ACGTTCAC” εντοπίζεται στο “GATCGTG”, αν το μέγεθός του είναι μεγαλύτερο από το προηγούμενο γίνεται αυτό το LCS
- Ελέγχοντας όλα τα substrings το αποτέλεσμα του αλγορίθμου είναι το “CGT” που είναι το LCS των δύο string

Πολυπλοκότητα χρόνου: Για να βρούμε όλα τα πιθανά substrings του ενός string θα χρειαστεί πολυπλοκότητα χρόνου $O(m^2)$ και για την αναζήτηση των substrings μέσα στο άλλο string με χρήση του KMP αλγορίθμου επιπλέον O(n). Οπότε συνολικά η πολυπλοκότητα θα είναι $O(n*m^2)$.

2.2. Dynamic Programming

Ο δυναμικός προγραμματισμός είναι τεχνική για την επίλυση αναδρομικών προβλημάτων με πιο αποδοτικό τρόπο. Στα αναδρομικά προβλήματα χρειάζεται συχνά η επανειλημμένη επίλυση υποπροβλημάτων. Για να αποφύγουμε να λύνουμε ξανά τα ίδια προβλήματα, στον δυναμικό προγραμματισμό αποθηκεύουμε τις λύσεις των υποπροβλημάτων ώστε να χρησιμοποιηθούν αργότερα. Με λίγα λόγια κάνουμε χρήση μνήμης. Με χρήση αυτών των δύο συνιστωσών του δυναμικού προγραμματισμού, της μνήμης και της αναδρομής. μπορούμε να λύσουμε τα περισσότερα προβήματα.
Για τον αλγόριθμο αυτό θα ξεκινήσουμε δημιουργώντας ένα πίνακα ο οποίος θα περιέχει τα μήκη των μέγιστων κοινών suffixes των substrings των strings που δίνουμε σαν είσοδο. Η προσέγγιση που θα ακολουθήσουμε θα είναι από κάτω προς τα πάνω. Θα ξεκινήσουμε να λύνουμε το πρόβλημα για τις μικρότερες δυνατές θέσεις στα string και θα

αποθηκεύουμε τις λύσεις για αργότερα. Καθώς θα εκτελούνται υπολογισμοί για όλο και μεγαλύτερες θέσεις, θα γίνεται χρήση των αποθηκευμένων λύσεων. Οπότε σε δύο μεταβλητές, μία για τη γραμμή και μία για την στήλη, θα αποθηκεύσουμε την θέση του κελιού του πίνακα που περιέχει τη μέγιστη τιμή. Όταν οι χαρακτήρες των δύο strings είναι ίδιοι στην αντίστοιχη θέση του πίνακα θα μπαίνει το άθροισμα: 1 για αυτό το ταίριασμα συν την τιμή που βρίσκεται στο πάνω αριστερό κελί του πίνακα.
Το πρόβλημα του LCS μπορεί να λυθεί με αυτό τον αλγόριθμο καθώς διαθέτει δύο ιδιότητες: 1. Επικαλυπτόμενα υποπροβλήματα: στην αναδρομή λύνουμε αυτά τα προβλήματα ξανά και ξανά ενώ στον δυναμικό μόνο μία και η λύση αποθηκεύεται για μελλοντική χρήση. Εδώ τα υποπροβλήματα είναι τα suffixes των substrings των strings 2. Βέλτιστη υποδομή: υπάρχει σε ένα πρόβλημα όταν η τελική λύση μπορεί να προκύψει από τις λύσεις των υποπροβλημάτων. Εδώ το επιθυμητό αποτέλεσμα μπορεί να προκύψει από τις λύσεις των υποπροβλημάτων.

Αν το μήκος του string x είναι m και του y είναι n το μέγιστο κοινό επίθεμα έχει την παρακάτω βέλτιστη ιδιότητα υποδομής:

LCSuffix(x, y, m, n) = LCSuffix(x, y, m-1, n-1) +1 όταν x[m-1] = y[n-1]
0 σε κάθε άλλη περίπτωση

Το μέγιστου μήκους LCSuffix θα είναι το LCS δηλαδή:

LCS(x, y, m, n) = Max(LCSuffix(x, y, i, j) για 1 ≤ i ≤ m
και 1 ≤ j ≤ n

Όταν ολοκληρωθεί ο έλεγχος και η συμπλήρωση του πίνακα θα διαπεράσουμε διαγώνια προς τα πίσω τον πίνακα ξεκινώντας από το κελί που δείχνουν οι δύο μεταβλητές γραμμής και στήλης, μειώνοντας κάθε φορά την τιμή της σειράς και της στήλης κατά 1. Έτσι θα πάρουμε το μέγιστο κοινό substring.

	G	A	T	C	G	T	G
A	0	1	0	0	0	0	0
C	0	0	0	1	0	0	0
G	1	0	0	0	2	0	1
T	0	0	1	0	0	3	0
T	0	0	1	0	0	1	0
C	0	0	0	2	0	0	0
A	0	1	0	0	0	0	0
C	0	0	0	1	0	0	0

Σχήμα 1.8: Εφαρμογή δυναμικού προγραμματισμού για την εύρεση LCS

Πολυπλοκότητα χρόνου: Για την εύρεση των μέγιστων κοινών suffixes όλων των substrings, την αποθήκευση τους στον πίνακα και εξαγωγή μετά από τον πίνακα του LCS θα χρειαστεί χρόνος πολυπλοκότητας O(n*m), όπου n,m τα μήκη των string της εισόδου.

2.3. Suffix Array

Ο ορισμός του suffix Array είναι πρόμοιος με τον ορισμό ενός suffix Tree το οποίο είναι συμπιεσμένο δέντρο όλων των suffixes ενός δοσμένου κειμένου. Έτσι και το suffix array είναι μία ταξινομημένη διάταξη όλων των suffixes ενός δοσμένου string. Οποιοσδήποτε suffix tree αλγόριθμος μπορεί να αντικατασταθεί με έναν αλγόριθμο που χρησιμοποιεί ένα suffix array ενισχυμένο με επιπλέον πληροφορίες και λύνει το ίδιο πρόβλημα με ίδια πολυπλοκότητα χρόνου. Ένα suffix array μπορεί να κατασκευαστεί από ένα suffix tree αν κάνουμε μία αναζήτηση κατά βάθος(DFS) σε αυτό. Επίσης ένα suffix tree μπορεί να κατασκευαστεί από ένα suffix array σε γραμμικό χρόνο.
Κάποια πλεονεκτήματα του suffix array σε σχέση με το suffix tree είναι βελτιωμένες απαιτήσεις χώρου, απλοί γραμμικοί αλγόριθμοι κατασκευής και βελτιωμένη cache τοπικότητα.
Είναι μια δομή δεδομένων που χρησιμοποιείται μεταξύ άλλων σε αλγορίθμους συμπίεσης δεδομένων, σε ευρετήρια πλήρους κειμένου και στο πεδίο της βιβλιομετρίας.

Ένα απλός τρόπος κατασκευής ενός suffix array είναι η κατασκευή ενός array όλων των suffixes και έπειτα η ταξινόμησή τους.

Στον αλγόριθμο LCS με χρήση suffix array αρχικά συνδυάζουμε τα string που δίνονται στην είσοδο σε ένα νέο string ως εξής:

$s=x\$_1y\$_2$

όπου x,y τα δύο string εισόδου, s το νέο string που προκύπτει και $\$_1$,$\$_2$ δύο διαφορετικά σύμβολα που δεν εντοπίζονται μέσα στα προηγούμενα string. Στη συνέχεια δημιουργείται μία διάταξη με όλα τα πιθανά suffixes που προκύπτουν για το s και μετά αυτά θα ταξινομηθούν με λεξικογραφική σειρά. Τέλος με χρήση ενός αλγορίθμου LCP(κάνουμε χρήση του αλγορίθμου binary search από αυτούς που παρουσιάστηκαν όντας ο πιο γρήγορος) θα βρούμε το LCP ανάμεσα σε κάθε τιμή και την επόμενή της. Το μεγαλύτερο από αυτά τα LCP θα είναι το ζητούμενο LCS των δύο string x,y.

Έστω string x = ”ACGTT” και y = “GATCG”, ο συνδυασμός τους για $\$_1$=# και $\$_2$=$ θα είναι το s=”ACGTT#GATCG$” οπότε:

	suffixes	SA	sorted suffixes	LCP
1	ACGTT#GATCG$	6	#GATCG$	0
2	CGTT#GATCG$	12	$	0
3	GTT#GATCG$	1	ACGTT#GATCG$	1
4	TT#GATCG$	8	ATCG$	0
5	T#GATCG$	10	CG$	2
6	#GATCG$	2	CGTT#GATCG$	0
7	GATCG$	11	G$	1
8	ATCG$	7	GATCG$	1
9	TCG$	3	GTT#GATCG$	0
10	CG$	5	T#GATCG$	1
11	G$	9	TCG$	1
12	$	4	TT#GATCG$	

Σχήμα 1.9: Εφαρμογή του αλγορίθμου για εύρεση του LCS με χρήση suffix array

Από το παραπάνω σχήμα είναι φανερό ότι το μέγιστο LCP που προκύπτει είναι το “CG” το οποίο αποτελεί και το επιθυμητό LCS του παραδείγματος.

Πολυπλοκότητα χρόνου:

2.4. Suffix Tree

Τα Suffix Trees βρίσκουν πολλές εφαρμογές στην βιολογία όπως στο πρόβλημα της γονιδιακής ρύθμισης, στην επιλογή υπογραφής των γονιδιωματικών αλληλουχιών και στο μέγιστο κοινό substring μεταξύ ακολουθιών γονιδιωμάτων ώστε να επιτευχθεί η γονιδιωματική ευθυγράμμιση.
Ο αλγόριθμος ξεκινάει φτιάχνοντας το Generalized Suffix Tree των string που δίνονται στην είσοδο με χρήση του αλγορίθμου McCreigth. Το Generalized Suffix Tree κατασκευάζετε από ένα σύνολο από strings. Έστω ότι έχουμε δύο strings το x και το y. Για αυτά τα δύο strings θα κατασκευάσουμε ένα νέο string έστω s όπου s=x#y$ όπου τα # και $ είναι κάποια από τα τερματικά σύμβολα που μπορούμε να χρησιμοποιήσουμε. Έπειτα θα χτίσουμε το suffix tree για το s το οποίο θα αποτελεί το Generalized Suffix Tree για τα x και y. Με χρήση του αλγορίθμου McCreight προκύπτουν οι όροι που θα είναι οι ακμές του δέντρου. Ο αλγόριθμος McCreigth ουσιαστικά είναι μία τροποποίηση του αλγορίθμου brute force που υπολογίζει τα suffix links χρησιμοποιώντας τα σαν συντομότερες διαδρομές. Ξεκινάει με ένα δέντρο T_1 και για i = 1...n θα χτίσει δέντρο T_{i+1} για το οποίο θα ισχύει ότι α) το T_{i+1} είναι συμπιεσμένο δέντρο για το s[j...n], j≤i+1, β) όλοι οι μη τερματικοί κόμβοι έχουν ένα suffix link έστω L(-). Σε κάθε διέλευση θα προστίθεται κόμβος i+1, string έστω h(i) πριν τον κόμβο το οποίο είναι το LCP του x[i...n] και x[j...n] για κάθε j<1, string μετά τον κόμβο έστω t(i) τέτοιο ώστε x[i...n] = h(i)t(i) και suffix link όπου h(i)->L(h(i)). Τα φύλλα του δέντρου περιέχουν τα suffixes των strings. Συγκεκριμένα περιέχουν είτε suffixes του x είτε suffixes του y είτε κοινά suffixes. Ψάχνοντας από την ρίζα προς κάποιο εσωτερικό κόμβο βρίσκουμε τα κοινά substrings του x και y, το μονοπάτι από τη ρίζα προς τον “βαθύτερο” κόμβο θα μας δώσει το επιθυμητό LCS.

Πολυπλοκότητα χρόνου: Για είσοδο δύο string x και y με μήκη n και m αντίστοιχα, η δημιουργία του suffix tree χρειάζεται O(n+m) χρόνο και η εύρεση του LCS επίσης O(n+m).

3. Longest Common Subsequence

Το LCSub πρόβλημα είναι το πρόβλημα εύρεσης της της μεγαλύτερης υποακολουθίας που είναι κοινή για δύο δοσμένες ακολουθίες. Η υποακολουθία αυτή εμφανίζεται με την ίδια σειρά στις δύο ακολουθίες αλλά αντίθετα με το Longest Common Substring(LCS) πρόβλημα δεν είναι απαραίτητα συνεχόμενη. Μία ακολουθία μήκους n έχει 2^n διαφορετικές πιθανές υποακολουθίες. Αποτελεί ένα κλασικό πρόβλημα της επιστήμης των υπολογιστών, είναι η βάση του diff(ένα πρόγραμμα σύγκρισης αρχείων που δίνει ως έξοδο τις διαφορές ανάμεσά τους) και βρίσκει εφαρμογές στην βιοπληροφορική. Χρησιμοποιείται επίσης ευρέως από συστήματα ελέγχου αλλαγών όπως είναι το Git για την αποδοχή των πολλαπλών αλλαγών που έγιναν σε μια συλλογή αρχείων στην οποία αναζητούνται αλλαγές.
Το LCSub πρόβλημα αρχικά μελετήθηκε από μοριακούς βιολόγους για την χρήση του στην μελέτη παρόμοιων αμινοξέων. Η κλασική λύση με χρήση δυναμικού προγραμματισμού διατυπώθηκε από τους Wagner και Fischer(1974) και έχει χρόνο $O(n^2)$ στη χειρότερη περίπτωση. Οι Masek και Paterson βελτίωσαν αυτό τον αλγόριθμο χρησιμοποιώντας την τεχνική των “Four-Russians” ώστε να μειώσει τον χρόνο της χειρότερης περίπτωσης σε $O(n^2/\log n)$. Ο Aho(1976) χρησιμοποιώντας ένα μοντέλο δέντρου αποφάσεων έδωσε ένα κάτω όριο χρόνου επίλυσης O(m*n) ενώ και ο Gotoh(1982) έδειξε ότι το πρόβλημα μπορεί να λυθεί σε χρόνο O(m*n) με χρήση δυναμικού προγραμματιμσού. Αρκετοί παράλληλοι αλγόριθμοι έχουν προταθεί για την περαιτέρω μείωση του χρόνου υπολογισμού με διαφορετικά υπολογιστικά μοντέλα(Y. Panet(1998), Jean Frédéric Myoupo(1999), L. Bergroth(2000), A. Aggarwal(1988)) και με συστολικές συστοιχίες(K. Nandan Babu(1997), V. Freschi(2000)). Επίσης υπάρχουν διάφοροι αλγόριθμοι που η πολυπλοκότητά τους βασίζεται σε άλλες παραμέτρους. Οι Myers και Nakatsu, για παράδειγμα, παρουσίασαν έναν O(n*D) αλγόριθμο όπου η παράμετρος D είναι η απλή απόσταση Levenshtein ανάμεσα σε δύο δοσμένα strings. Ο αλγόριθμος των Hunt και Szymanski λύνει το πρόβλημα σε χρόνο O((R+n)*logn), όπου R μία παράμετρος η οποία είναι ο συνολικός αριθμός διατεταγμένων ζευγαριών θέσεων στις οποίες τα δύο string είναι ίδια. Οι Rahman και Iliopoulos βελτιώνοντας έναν αλγόριθμο τους παρουσίασαν έναν νέο αλγόριθμο χρόνου O(R*loglogn), όπου σε εφαρμογή σε προβλήματα όπως εύρεση της η μεγαλύτερης σε μήκος ανερχόμενης υποακολουθίας μίας μεταβολής των ακεραίων από 1 έως n ή εύρεση ενός μέγιστου πληθάριθμου γραμμικά υποδεικνυόμενου υποσυνόλου κάποιας πεπερασμένης συλλογής διανυσμάτων σε δισδιάστατο χώρο, το R προσεγγίζει το n. Σε αυτές τις περιπτώσεις ο αλγόριθμος παρουσιάζει μία σχεδόν γραμμική O(n*loglogn) συμπεριφορά.
Το LCSub βρίσκει χρήση σε διάφορα προβλήματα όπως στη γενετική και στη μοριακή βιολογία, στη συμπίεση δεδομένων και στην εύρεση pattern όπου το LCSub δύο strings χρησιμοποιείται σαν μέτρο ομοιότητας των strings και επομένως των αντικειμένων που αντιπροσωπεύουν. Στην μοριακή βιολογία θέλουμε να συγκρίνουμε DNA ή πρωτεϊνικές ακολουθίες για να διαπιστώσουμε πόσο όμοιες είναι. Διαφοροποιήσεις του LCSub προβλήματος χρησιμοποιούνται για την μελέτη ομοιότητας δομών RNA. Οι Bereg και Zhu παρουσίασαν ένα νέο μοντέλο για δομική ευθυγράμμιση πολλαπλών RNA ακολουθιών. Μία άλλη πρακτική εφαρμογή στην βιοπληροφορική είναι για την βελτίωση του χρόνου υπολογισμού και της ευαισθησίας φίλτρων που χρησιμοποιούνται για την εξαγωγή μεγάλων πολλαπλών επαναλήψεων σε DNA ακολουθίες. Τα φίλτρα αυτά εφαρμόζουν μια συνθήκη την οποία πρέπει να ικανοποιούν οι ακολουθίες για να είναι μέρος των επαναλήψεων.
Κάποιες διαφοροποιήσεις και γενικεύσεις του LCSub προβλήματος είναι τα Constrained Longest Common Subsequence(CLCS), K-substring LCS, gapped LCS και Longest Common Increasing Subsequence(LCIS). Στενά συγγενεύοντα προβλήματα σχετίζονται με

τον υπολογισμό της string-to-string απόστασης(string editing), tree-to-tree απόσταση, το πρόβλημα διόρθωσης κυκλικών συμβολοσειρών, συγχώνευση strings και εύρεση των μικρότερων κοινών υπερακολουθιών.

Παρακάτω εξετάζονται κάποιοι αλγόριθμοι επίλυσης του προβλήματος και η απόδοσή τους. Στους αλγορίθμους αυτούς εκτελείται αναζήτηση για LCSub ανάμεσα σε δύο strings, όπου ως είσοδος χρησιμοποιούνται βιολογικά δεδομένα και είναι οι εξής:

- Naive approach
- Dynamic Programming
- Longest Increasing Subsequence

3.1. Naive

Η naive μέθοδος για την λύση του προβλήματος είναι να δημιουργηθούν όλες τις πιθανές υποακολουθίες και για τις δύο ακολουθίες και να βρεθεί η κοινή με το μεγαλύτερο μήκος. Η λύση αυτή είναι εκθετική όσο αναφορά την χρονική πολυπλοκότητα. Ο αλγόριθμος αυτός βασίζεται στην παρατήρηση ότι το LCSub πρόβλημα έχει μία βέλτιστη υποδομή(optimal substructure) που σημαίνει ότι μπορούμε να σπάσουμε το πρόβλημα σε άλλα μικρότερα και πιο απλά προβλήματα, αυτά σε άλλα μικρότερα και ακόμα πιο απλά κοκ. Οπότε για δύο ακολουθίες x[0..n-1] και y[0...m-1] όπου n, m τα μήκη των ακολουθιών αντίστοιχα τα βήματα που θα ακολουθήσουμε είναι τα εξής:

- Αν οι τελευταίοι χαρακτήρες και των δύο χαρακτήρων είναι ίδιοι δηλαδή x[n-1] == y[m-1] το LCSub θα είναι το LCSub ανάμεσα στις δύο αρχικές ακολουθίες έχοντας αφαιρέσει όμως τον τελευταίο χαρακτήρα τους ο οποίος θα είναι μέρος του LCSub δηλαδή: Lcsub(x,y) = Lcsub(x[0:n-1], y[0:m-1]) + x[n-1]
- Αν οι τελευταίοι χαρακτήρες των δύο χαρακτήρων είναι διαφορετικοί δηλαδή x[n-1] != y[m-1] τότε ro LCSub θα είναι το μέγιστο ανάμεσα στο LCSub της x με την y μείον τον τελευταίο της χαρακτήρα και στο LCSub της y με την x μείον τον τελευταίο χαρακτήρας της, δηλαδή Lcsub(x,y) = max(Lcsub(x, y[0:m-1]), Lcsub(x[0:n-1], y))

Αν για παράδειγμα δώσουμε ως είσοδο στον παραπάνω αλγόριθμο τα strings “GATAGAC”, “AGACTC” τότε αφού οι τελευταίοι χαρακτήρες τους είναι ίδιοι θα έχουμε: Lcsub(“GATAGAC”, “AGACTC”) = Lcsub(“GATAGA”, “AGACT”) + “C” και συνεχίζουμε την επίλυση λύνοντας τα υποπροβλήματα που προκύπτουν με τον ίδιο τρόπο.

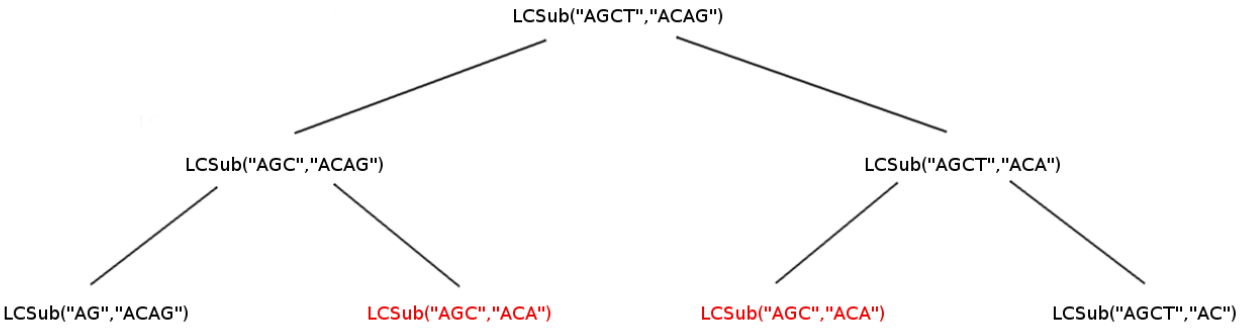
	G	A	T	A	G	A	C
A	-	-	-	-	-	-	-
G	x	-	-	-	-	-	-
A	-	x	-	-	-	-	-
C	-	-	-	-	-	-	-
T	-	-	x	-	-	-	-
C	-	-	-	-	-	-	x

Σχήμα 4.1: Πίνακας απεικόνισης της εφαρμογής της naive μεθόδου για δύο strings

Πολυπλοκότητα χρόνου: Όπως αναφέρθηκε και πιο πριν η πολυπλοκότητα του χρόνου για την naive μέθοδο είναι εκθετική και στην χειρότερη περίπτωση είναι $O(2^n)$. Η χειρότερη περίπτωση συμβαίνει όταν όλοι οι χαρακτήρες των δύο δοσμένων ακολουθιών είναι διαφορετικοί, οπότε δεν υπάρχει LCSub.

3.2. Dynamic Programming

Στον naive αλγόριθμο που περιγράφηκε πριν ελέγχονται όλοι οι συνδυασμοί των δύο ακολουθιών, μειώνοντας έτσι την ταχύτητά του. Επεκτείνοντάς τον θα βάλουμε όλες τις πιθανές θέσεις σε ένα δισδιάστατο πίνακα αποθηκεύοντας τα προσωρινά αποτελέσματα. Χρειάζεται μόνο να αποφύγουμε να διαβάσουμε αυτή την μνήμη πριν να έχουμε γράψει σε αυτήν. Σε αυτή την προσέγγιση αρχικά “χτίζεται” ένας δισδιάστατος πίνακας που θα χρησιμοποιηθεί σαν memoization cache. Έπειτα όταν ο πίνακας αυτός έχει φτιαχτεί τότε εντοπίζεται και επιστρέφεται από την πληροφορία που περιέχει ο πίνακας το επιθυμητό string.
Η διαδικασία που ακολουθούμε είναι παρόμοια με πριν όμως εδώ βασιζόμαστε στην παρατήρηση ότι το LCSub πρόβλημα παρουσιάζει επικαλυπτόμενα υποπροβλήματα. Για παράδειγμα για δύο strings έστω “AGCT”, “ACAG” ένα μέρος του δέντρου με βάση την naive μέθοδο θα είναι:



Σχήμα 4.2: Μερικό δέντρο για την εύρεση του LCSub με είσοδο δύο strings

Στο παραπάνω σχήμα το LCSub(“AGC”, “ACA”) θα πρέπει να λυθεί δύο φορές. Αν σχεδιάζαμε ολόκληρο το δέντρο θα βλέπαμε ότι τέτοιες περιπτώσεις υποπροβλημάτων που λύνονται ξανά και ξανά είναι πολλές. Οπότε το πρόβλημα αυτό έχει την ιδιότητα επικαλυπτόμενης υποδομής και όπως δείξαμε στην προηγούμενη μέθοδο έχει και βέλτιστη υποδομή άρα μπορεί να λυθεί με δυναμικό προγραμματισμό, όπου οι λύσεις των υποπροβλημάτων αποθηκεύονται προσωρινά για μελλοντική χρήση. Αναλυτικά τα βήματα που ακολουθούμε είναι τα εξής:

- Αρχικά ακολουθώντας τα βήματα της naive μεθόδου αποθηκεύουμε τις τιμές που προκύπτουν σε ένα πίνακα L διαστάσεων (n+1)*(m+1), όπου n, m τα μήκη των strings x, y στην είσοδο αντίστοιχα

- Η τιμή L[n][m] περιέχει το μήκος του LCSub. Δημιουργείται μία νέα κενή ακολουθία στην οποία θα αποθηκευτεί το LCSub
- Ξεκινώντας από την θέση L[n][m] του πίνακα και για κάθε κελί L[i][j]:
  - Αν οι χαρακτήρες των strings x, y στις θέσεις L[i][j] είναι ίδιες τότε αυτός ο κοινός χαρακτήρας είναι μέρος του LCSub
  - Διαφορετικά γίνεται σύγκριση μεταξύ των τιμών των θέσεων L[i-1][j] και L[i][j-1] και επιλέγεται η κατεύθυνση με την μεγαλύτερη τιμή

Αν εφαρμόσουμε τα προηγούμενα βήματα στα string “AGCT”, “ACAG”, ο πίνακας L που προκύπτει θα είναι:

		0	1	2	3	4
		Ø	A	C	A	G
0	Ø	0	0	0	0	0
1	A	0	1	1	1	1
2	G	0	1	1	1	2
3	C	0	1	2	2	2
4	T	0	1	2	2	2

Σχήμα 4.3: Ο πίνακας που δείχνει τα βήματα(πράσινο χρώμα) που ακολουθεί ο αλγόριθμος

Με πράσινο χρώμα είναι τα βήματα που ακολουθεί ο αλγόριθμος ξεκινώντας από την θέση (4,4). Όταν τα σύμβολα των x, y είναι ίδια πηγαίνουμε πάνω και αριστερά ενώ όταν είναι διαφορετικά, πηγαίνουμε αριστερά ή πάνω ανάλογα με το αν θα επιλέξουμε το LCSub(x[1...i-1], y[1...j]) ή το LCSub(x[1...i], y[1...j-1]). Ακολουθώντας αυτή την διαδρομή του αλγορίθμου βρίσκουμε ότι το LCSub που ψάχνουμε είναι το “AC”.
Πολυπλοκότητα χρόνου: Αν n το μήκος της ακολουθίας x και m της y τότε η χρονική πολυπλοκότητα του αλγορίθμου είναι O(n*m) και είναι σαφώς γρηγορότερος από την naive μέθοδο.

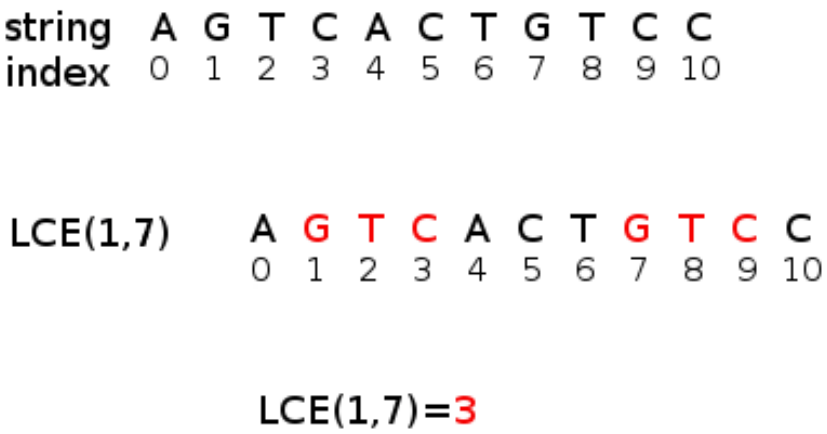
4. Longest Common Extension (LCE) πρόβλημα

Το LCE πρόβλημα για ένα string έστω x και για ένα ζευγάρι τιμών i,j είναι το πρόβλημα εύρεσης του μέγιστου κοινού substring στο x, που ξεκινά στις τιμές i και j. Ουσιαστικά πρόκειται για το μέγιστο κοινό prefix που ξεκινάει σε αυτές τις δύο τιμές. Εμφανίζεται σαν υποπρόβλημα σε διάφορα θεμελιώδη προβλήματα με strings όπως στο k-mismatch πρόβλημα, στη k-Difference Global Alignment, στο approximate string searching, στον υπολογισμός των διαδοχικών επαναλήψεων(ακριβείς ή κατά προσέγγιση), στον υπολογισμό παλίνδρομων και στο ταίριασμά τους με χρήση wildcards.
Οι Ilie, Navarro και Tinta αναζήτησαν απλούς και αποδοτικούς αλγορίθμους για το LCE πρόβλημα παρατηρώντας το μέγεθος των τιμών για διάφορα αλφάβητα. Ως αποτέλεσμα οι αλγόριθμοί τους είναι οι καλύτεροι όταν εφαρμοστούν σε πρακτικές εφαρμογές, τόσο σε χρόνο όσο και σε χώρο. Οι Landau-Vishkin χρησιμοποιούν το LCE σαν μια υπορουτίνα για επίλυση του προβλήματος του k-mismatch. Υπολογίζοντας το LCE που χρειάζεται στον κατά προσέγγιση string searching αλγόριθμό τους με τον απλούστερο από τους αλγόριθμούς των Ilie, Navarro και Tinta, προέκυψε αλγόριθμος που τρέχει έως και 20 φορές γρηγορότερα.
Οι Blanchet-Sadri και Lazarow επέκτειναν τα suffix δέντρα σε μερικές λέξεις, δηλαδή strings τα οποία περιέχουν αδιάφορους ή άγνωστους όρους. Έπειτα υπολόγισαν το Longest Common Compatible Extension(LCCE) για κάθε ζευγάρι θέσεων που πρόκειται για τα μεγαλύτερα substrings που αρχίζουν από τις δύο αυτές θέσεις και είναι συμβατά.
Παρακάτω αρχικά παρουσιάζονται δύο απλοί αλγόριθμοι για υπολογισμό του LCE και του LCE with k-mismatches, οι οποίοι χρησιμοποιούνται στην παρουσίαση ενός αλγορίθμου που παρουσίασαν το 2017 οι Alamro H., Alzamel M., Iliopoulos C.S., Pissis S.P., Watts S., Sung WK. Ο αλγόριθμος αυτός διαπιστώνει αν το string που του δίνεται ως είσοδος είναι k-Closed String.

4.1. LCE - Naive Search

Ο αλγόριθμος αυτός μας δίνει το μέγεθος του μεγαλύτερου extension συγκρίνοντας τους χαρακτήρες με αφετηρία τις δύο δοσμένες θέσεις και για κάθε κοινό διαδοχικό χαρακτήρα αυξάνεται μια μεταβλητή κατά ένα. Ουσιαστικά είναι σαν να υπολογίζουμε το μέγιστο κοινό prefix ανάμεσα σε δύο θέσεις στο ίδιο string. Η συνάρτηση θα μας επιστρέψει αυτή την μεταβλητή που θα είναι και το μέγιστο substring.
Αναλυτικά τα βήματα που ακολουθεί:

- Αρχικοποίηση της μεταβλητής του μεγέθους με την τιμή 0
- Αρχίζει να συγκρίνει το κοινό prefix που ξεκινάει από το i και το j κατά ένα χαρακτήρα τη φορά
- Εάν οι χαρακτήρες είναι οι ίδιοι, τότε αυτός ο χαρακτήρας είναι μέρος του LCE οπότε αυξάνεται η μεταβλητή του μεγέθους κατά ένα
- Εάν δεν είναι ίδιοι τότε επιστρέφει την τιμή του μεγέθους μέχρι εκείνη την στιγμή
- Το μέγεθος του μέγιστου κοινού prefix που θα επιστρέψει είναι το μέγεθος του επιθυμητού LCE



Σχήμα 1.9: Εφαρμογή της Naive μεθόδου για εύρεση του LCE

Παρότι μη αναμενόμενο καθώς η naive μέθοδος έχει μεγαλύτερη ασυμπτωτική πολυπλοκότητα χρόνου σε σχέση με άλλες αποδοτικές μεθόδους, τις ξεπερνάει σε απόδοση σε πρακτικές εφαρμογές. Για παράδειγμα έχει αποδειχθεί ότι είναι 5-6 φορές πιο γρήγορη από την μέθοδο Segment Tree, παρότι η ασυμπτωτική πολυπλοκότητα χρόνου της δεύτερης είναι πολύ μικρότερη της πρώτης. Γενικά η naive μέθοδος είναι η βέλτιστη επιλογή για υπολογισμό LCE όταν πρόκειτα για απόδοση μεσαίας περίπτωσης. Αυτό είναι κάτι που δεν συμβαίνει συχνά στην επιστήμη της πληροφορικής δηλαδή ένας φαινομενικά γρηγορότερος αλγόριθμος να αποδίδει χειρότερα από έναν λιγότερο αποδοτικό, όταν

δοκιμάζονται σε πρακτικά προβλήματα. Τέτοιες περιπτώσεις αποδεικνύουν ότι παρόλο που η ασυμπτωτική ανάλυση είναι ένας από τους πιο αποτελεσματικούς τρόπους για να συγκριθούν δύο αλγόριθμοι θεωρητικά, σε πρακτικές εφαρμογές κάποιες φορές μπορεί να συμβεί το αντίθετο.

4.2. LCE with k Mismatches

Μία λίγο διαφορετική προσέγγιση αν γενικεύσουμε το LCE, είναι η εύρεση του LCE για δύο θέσεις του string όμως εδώ δεν χρειάζεται το prefix των suffixes να είναι εντελώς όμοιο αλλά μπορεί να διαφέρει σε έναν k αριθμό θέσεων τον οποίο θα τον καθορίσουμε εμείς δίνοντας τον ως είσοδο στον αλγόριθμο. Δηλαδή το κοινό prefix τους θα θεωρείται έγκυρο εάν ταιριάζουν με k ή λιγότερα λάθη όσον αφορά την Hamming απόστασή τους.

Σχήμα 1.9: Εύρεση του LCE για k-mismatches

Ο παραπάνω αλγόριθμος μοιάζει με τον προηγούμενο μόνο που εδώ όταν βρεθεί κάποιος χαρακτήρας που δεν είναι κοινός κατά την αναζήτηση που ξεκίνησε με αφετηρία τις δύο θέσεις του string που δόθηκαν ως είσοδος, ο αλγόριθμος δεν τερματίζει. Αντίθετα για κάθε διαφορετικό χαρακτήρα θα αυξάνεται μια μεταβλητή η οποία όταν φτάσει το μέγεθος του αριθμού των θέσεων που έχουμε εμείς ορίσει θα τερματίζεται ο αλγόριθμος, δίνοντας μας το μέγιστο extension.

Με βάση τα παραπάνω έγινε η μελέτη του αλγορίθμου που θα περιγραφεί παρακάτω και η υλοποίηση του σε python.

4.3. Efficient Identification of k-Closed Strings

Ένα closed string είναι ένα string το οποίο περιέχει ένα substring το οποίο είναι prefix και suffix ταυτόχρονα, αλλά δεν το συναντάμε πουθενά αλλού στο string. Τα closed strings έχουν μελετηθεί σε πιο πρακτικό επίπεδο σε σχέση με τα παλίνδρομα strings. Έχει παρατηρηθεί ότι ο αριθμός closed strings σε ένα string ελαχιστοποιείται όταν αυτά τα string είναι παλίνδρομα. Επιπλέον ισχύει ότι το πάνω όριο στον αριθμό των παλίνδρομων συμπίπτει με το κάτω όριο του αριθμού των closed strings.
Τα closed strings εισήγαγε το 2011 ο Fici σαν αντικείμενα συνδυαστικού ενδιαφέροντος. Οι Badkobeh, Fici και Liptak έχουν αποδείξει ότι υπάρχει ένα αυστηρό κατώτατο όριο για τον αριθμό των closed strings σε strings δοσμένουμεγέθους και αλφάβητου. Η Badkobeh είχε παρουσιάσει έναν αλγόριθμο για την μετατροπή ενός string μήκους n σε ακολουθία των μεγαλύτερων closed strings με πούπλοκότητα χώρου και χρόνου O(n) και έναν άλλο αλγόριθμο για υπολογισμό του μεγαλύτερου closed string για κάθε θέση του δοσμένου string σε χρόνο O(n*(logn/log(logn)) και χώρο O(n). Επίσης έχουν μελετηθεί σε σχέση με Sturmian και τραπεζοειδείς λέξεις. Οι Alamro H., Alzamel M., Iliopoulos C.S., Pissis S.P., Watts S., Sung WK. (2017) επέκτειναν τον ορισμό των closed strings σε k-closed strings στα οποία επιτρέπονται κάποιες διαφορές ανάμεσα στο prefix και suffix, ο αριθμός των οποίων δεν πρέπει να ξεπερνάει μια παράμετρο k που έχουμε ορίσει. Ο αριθμός αυτός είναι ο αριθμός σφαλμάτων της Hamming απόστασης ανάμεσα στα δύο strings, όπου Hamming απόσταση ανάμεσα σε δύο string x και y του ίδιου μήκους είναι ο αριθμός των θέσεων στα x και y στις οποίες έχουν διαφορετικούς χαρακτήρες. Με τον αλγόριθμο που παρουσίασαν μπορεί να διαπιστωθεί αν ένα δοσμένο string μήκους n για ένα ακέραιο αλφάβητο είναι k-closed σε χρόνο O(k*n) και O(n) χώρο, καθώς και το όριο για το οποίο το string είναι k-closed. Ο αλγόριθμος αυτός θα αναλυθεί και θα υλοποιηθεί παρακάτω. Αρχικά ο αλγόριθμος κάνει χρήση του LCE και της επέκτασής του LCE-k στα οποία αναφερθήκαμε σε προηγούμενα κεφάλαια. Επίσης κάνει χρήση του αλγορίθμου Kangaroo, μια μέθοδος που χρησιμοποιείται για εκτέλεση πολλαπλών LCE-k ερωτημάτων δοσμένο string χτίζοντας ένα suffix tree. Έπειτα ορίζει γενικά τα k-closed strings ως εξής:

Ορισμός 1: Ένα string x μήκους n καλείται k-closed εάν και μόνο αν n≤1 ή ικανοποιούνται οι παρακάτω προυποθέσεις για k’ όπου 0≤k’≤k:

- υπάρχουν κατάλληλα prefix u και suffix v του x, με μήκη |u|=|v| ώστε $\delta_H(u,v)\leq k'$
- εκτός από το u και v, δεν υπάρχει άλλος παράγοντας του x έστω w με μήκος |w|=|u| =|v| για τον οποίο να ισχύει $\delta^H(u,w)\leq k'$ ή $\delta^H(u,w)\leq k'$

Στον παραπάνω ορισμό τα u και v για το μικρότερο k’ καλούνται το k-closed όριο του x. Αν n≤1 ως k-closed όριο ορίζεται το ε.
Επιπλέον δίνονται κάποιοι επιπλέον ορισμοί, που είναι διαφοροποιήσεις του πρώτου και συγκεκριμένα για τα ασθενώς k-closed strings, για τα ισχυρώς k-closed strings και τα ψευδώς k-closed strings. Ως ασθενώς k-closed string ορίζεται:

Ορισμός 2: Ένα string έστω x μήκους n καλείται ασθενώς k-closed εάν και μόνο εάν n≤1 ή ικανοποιούνται οι παρακάτω προυποθέσεις:

- υπάρχουν κατάλληλα prefix u και suffix v του x, με μήκη |u|=|v| ώστε $\delta_H(u,v)\leq k'$
- τα u και v υπάρχουν μόνο ως prefix και suffix αντίστοιχα και δεν εμφανίζονται αλλού μέσα στο x

Τα u,v τότε αποκαλούνται ασθενώς k-closed όριο του x. Αν n≤1 ως ασθενώς k-closed όριο ορίζεται το ε.

Ορισμός 3: Ένα string έστω x μήκους n καλείται ισχυρώς k-closed εάν και μόνο εάν n≤1 ή ικανοποιούνται οι παρακάτω προυποθέσεις:

- υπάρχει κάποιο όριο έστω b του x
- εκτός από το prefix και το suffix δεν υπάρχει άλλος παράγοντας του x, έστω w, μήκους |w| = |b| ώστε $\delta_H(b,w) \leq k$

Το b ονομάζεται ισχυρώς k-closed όριο του x. Αν n≤1 ως ισχυρώς k-closed όριο ορίζεται το ε.

Για την κατασκευή του αλγορίθμου θα επικεντρωθούμε σε μια συγκεκριμένη υποκατηγορία των πιθανών LCE-k ερωτημάτων και θα δημιουργήσουμε δύο δομές στις οποίες θα αποθηκεύσουμε τις τιμές τους. Αυτές οι δομές είναι η Longest Prefix k-Match ακολουθία και η Longest Suffix k-Match ακολουθία ενός string x, τις οποίες από εδώ και στο εξής θα τις αποκαλούμε $LPM_k(x)$ και $LSM_k(x)$ αντίστοιχα.
Η $LPM_k(x)[j]$(αντίστοιχα $LSM_k(x)[j]$) είναι το μήκος της μεγαλύτερης ακολουθίας του x που ξεκινάει(τελειώνει) από την μεταβλητή j και η οποία είναι ίδια με το prefix(suffix) του x ίδιου μήκους με εξαίρεση k χαρακτήρες, με εξαίρεση το j στο ίδιο το prefix(suffix), για το οποίο ορίζουμε την τιμή -1. Σύμφωνα με τον ορισμό η ακολουθία LPM είναι ίση με την αντίστροφη της LSM για το αντίστροφο x, ενώ ισχύει και το αντίθετο, οπότε:

$$\mathbf{LPM_k(x)[j] = LSM_k(x^R)[n-1-j]}$$
$$\mathbf{LSM_k(x)[j] = LPM_k(x^R)[n-1-j]}$$

Με βάση αυτές τις δύο σχέσεις οι LPM και LSM εκφράζονται σε συνάρτηση με το LCE ώστε να μπορεί να εφαρμοστεί η μέθοδος Kangaroo για την κατασκευή τους:

$$\mathbf{LPM_k(x)[j] = LCE_k(0, j) \text{ του } x \text{ για } j \in 1, n-1]}$$
$$\mathbf{LSM_k(x)[j] = LCE_k(0, n-1-j) \text{ του } x^R \text{ για } j \in [0, n-2]}$$

Σε αυτές τις σχέσεις βασίζονται οι τρεις συνθήκες που πρέπει να ικανοποιεί ένα string x μήκους n ≥ 2 ώστε να είναι k-closed. Συγκεκριμένα πρέπει για έναν αριθμό k, 0 ≤ k ≤ n να υπάρχει ένα j ∈ {1, . . . , n − 1} και ένα k’ ∈ {0, . . . , k} ώστε να ισχύουν οι εξής συνθήκες:

1. $\mathbf{j + LPM_{k'}(x)[j] = n}$
2. $\mathbf{\forall i < j, LPM_{k'}(x)[i] < LPM_{k'}(x)[j]}$
3. $\mathbf{\forall i > n-1-j, LSM_{k'}(x)[i] < LSM_{k'}(x)[j]}$

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Abouelhoda, Mohamed Ibrahim; Kurtz, Stefan; Ohlebusch, Enno (2004). *"Replacing suffix trees with enhanced suffix arrays"*. Journal of Discrete Algorithms. 2: 53

2. Alamro, H, Alzamel, M, Iliopoulos, CS, Pissis, SP, Watts, S & Sung, W-K 2017, Efficient Identification of k-Closed Strings. in G Boracchi, L Iliadis, C Jayne & A Likas (eds), *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25--27, 2017, Proceedings.* vol. 744, Springer International Publishing Switzerland, Cham, pp. 583-595

3. Arnold, Michael & Ohlebusch, Enno. (2011). *Linear Time Algorithms for Generalizations of the Longest Common Substring Problem*. Algorithmica. 60. 806-818. 10.1007/s00453-009-9369-1.

4. Babenko, Maxim & Starikovskaya, Tatiana. (2008). *Computing Longest Common Substrings Via Suffix Arrays*. 64-75. 10.1007/978-3-540-79709-8_10.

5. Böckenhauer, Hans-Joachim & Bongartz, Dirk. (2007). *Algorithmic aspects of bioinformatics.* Translated from the German original. 10.1007/978-3-540-71913-7.

6. Golnaz Badkobeh, Hideo Bannai, Keisuke Goto, Tomohiro I, Costas S. Iliopoulos, Shunsuke Inenaga, Simon J. Puglisi, Shiho Sugimoto, Closed factorization, Discrete Applied Mathematics, Volume 212, 2016, Pages 23-29, ISSN 0166-218X

7. Gusfield, Dan (1999), *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press

8. Fischer J. (2011) *Inducing the LCP-Array*. In: Dehne F., Iacono J., Sack JR. (eds) Algorithms and Data Structures. WADS 2011. Lecture Notes in Computer Science, vol 6844. Springer, Berlin, Heidelberg

9. Kärkkäinen, Juha; Sanders, Peter(2003). *Simple linear work suffix array construction*. Proceedings of the 30th international conference on Automata, languages and programming

10. Liu, Wei & Chen, Lin. (2007). A Fast Longest Common Subsequence Algorithm for Biosequences Alignment. International Federation for Information Processing Digital Library; Computer And Computing Technologies In Agriculture, Volume I;. 258. 10.1007/978-0-387-77251-6_8.

11. Lucian Ilie, Gonzalo Navarro, Liviu Tinta, *The longest common extension problem revisited and applications to approximate string searching*, Journal of Discrete Algorithms, Volume 8, Issue 4, 2010, Pages 418-428, ISSN 1570-8667

12. Manber, U. & Myers, G. W. (1993), *'Suffix arrays: a new model for on-line string searches'*. SIAM J. Comput. 22 (5): 935–948.

13. Rahman, Mohammad Sohel and Costas S. Iliopoulos. “A New Efficient Algorithm for Computing the Longest Common Subsequence.” *Theory of Computing Systems* 45 (2007): 355-371.

14. Rahman, Mohammad & Iliopoulos, Costas. (2006). Algorithms for Computing Variants of the Longest Common Subsequence Problem. 399-408. 10.1007/11940128_41.

15. S. Gog and E. Ohlebusch, *“Fast and lightweight LCP-array construction algorithms”,* in Algorithm Engineering and Experiments (ALENEX), 2011, pp. 25–34.

16. Tomas Flouri, Emanuele Giaquinta, Kassian Kobert, Esko Ukkonen, *Longest common substrings with k mismatches*, Information Processing Letters, Volume 115, Issues 6–8, 2015, Pages 643-647, ISSN 0020-0190

17. Yusufu, Munina & Yusufu, Gulina. (2015). *Efficient Algorithm for Extracting Complete Repeats from Biological Sequences*. International Journal of Computer Applications. 128. 33-37. 10.5120/ijca2015906752.

18. W.J. Hsu, M.W. Du, *New algorithms for the LCS problem*, Journal of Computer and System Sciences, Volume 29, Issue 2, 1984, Pages 133-152, ISSN 0022-0000

ΠΑΡΑΡΤΗΜΑ

Παρακάτω παρατίθενται οι κώδικες που υλοποιήθηκαν

lcp_wbw.py

```
# LCP word by word

# lcp for 2 strings
def lcp(x,y):

    result = ""
    n=min(len(x),len(y))

    for j in range(n):
        if x[j]!=y[j]:
            break
        result += x[j]

    return result

def LCP(x):

    n=len(x)
    result=x[0]
    for i in xrange(1,n):
        result=lcp(result,x[i])

    return result
```

lcp_cbc.py

```
# LCP character by character

def LCP(x):

        result=''
        n=len(x)
        m=len(x[0])
        for i in range(n-1):
                m=min(m,len(x[i+1]))

        for j in range(m):
                y=x[0][j]
                for k in range(n):
                        if x[k][j]!=y:
                        break
```

```
            else:
                continue
            break

    result += x[0][0:j+1]

    return result
```

lcp_dac.py

```
# LCP divide and conquer

def lcp2(x,y):

    result = ""
    n=min(len(x),len(y))

    for j in range(n):
        if x[j]!=y[j]:
            break
        result += x[j]

    return result

def LCP(x,low,high):

    if low==high:
        return x[low]

    if high>low:
        mid=low+(high-low)/2

        str1=LCP(x,low,mid)
        str2=LCP(x,mid+1,high)

        result=lcp2(str1,str2)

        return result
```

lcp_bs.py

```
# LCP binary search

def LCP(x):

    n=len(x)
```

```
    prefix=""
    m=len(x[0])
    for i in range(n-1):
        m=min(m,len(x[i+1]))

    low=0
    high=m
    mid=low+(high-low)/2

    for i in range(1,n):
        for j in range(mid):
            if x[i][j]!=x[0][j]:
                prefix=x[0][0:j]
                break
        else:
            prefix=x[0][0:mid]

            for w in range(1,n):
                for v in range(mid,high):
                    if x[w][v]!=x[0][v]:
                        prefix=x[0][0:v]
                        break
                else:
                    prefix=x[0][0:high]
                    continue
                break
            continue
        break

    return prefix
```

lcs_naive.py

```
def lcs_naive(x,y):

    length = len(x)
    z = []
    for i in range(length):
        for j in range(i,length):
            z.append(x[i:j+1])
    lcs=""
    lz=len(z)
    for i in range(lz-1):
        p=KMPSearch(z[i],y)
        if p>=0:
            if len(z[i])>len(lcs):
                lcs=z[i]
    return lcs
```

lcs_dynamic.py

```
def lcs(x,y):

    n=len(x)
    m=len(y)
    result=""
    l3n=0
    row=0
    col=0
    L=[[0 for v in range(m)]for w in range(n)]

    for i in range(n):
        for j in range(m):
            if x[i]==y[j]:
                L[i][j]=L[i-1][j-1]+1
                if l3n<L[i][j]:
                    l3n=L[i][j]
                    row=i
                    col=j
    if l3n==0:
        print "no common substring"

    while L[row][col]!=0:
        result=x[row]+result
        row-=1
        col-=1

    return result
```

lcs_sarray.py

```
# LCS with Suffix Array

def lcs(x,y):

    s=x+"#"+y+"$"
    n=len(s)
    r=[]
    lcs=""

    for i in range(n):
        r.append(s[i:n])

    # lexikographiki taksinomisi tis listas
    r=sorted(r)
```

```
    for j in range(n-1):
        l=LCP([r[j],r[j+1]])
        if len(l)>len(lcs):
            lcs=l

    return lcs
```

lcs_stree.py

```
import sys


class STree():
    # Class representing the suffix tree
    def __init__(self, input=''):
        self.root = _SNode()
        self.root.depth = 0
        self.root.idx = 0
        self.root.parent = self.root
        self.root._add_suffix_link(self.root)

        if not input == '':
            self._build_generalized(input)

    def _build(self, x):
            # Builds a Suffix tree
            self.word = x
        self._build_McCreight(x)

    def _build_McCreight(self, x):
            # Builds a Suffix tree using McCreight O(n) algorithm
        u = self.root
        d = 0
        for i in range(len(x)):
            while u.depth == d and u._has_transition(x[d+i]):
                u = u._get_transition_link(x[d+i])
                d = d + 1
                while d < u.depth and x[u.idx + d] == x[i + d]:
                    d = d + 1
            if d < u.depth:
                u = self._create_node(x, u, d)
            self._create_leaf(x, i, u, d)
            if not u._get_suffix_link():
                self._compute_slink(x, u)
            u = u._get_suffix_link()
            d = d - 1
            if d < 0:
                d = 0
```

```
def _create_node(self, x, u, d):
    i = u.idx
    p = u.parent
    v = _SNode(idx=i, depth=d)
    v._add_transition_link(u,x[i+d])
    u.parent = v
    p._add_transition_link(v, x[i+p.depth])
    v.parent = p
    return v

def _create_leaf(self, x, i, u, d):
    w = _SNode()
    w.idx = i
    w.depth = len(x) - i
    u._add_transition_link(w, x[i + d])
    w.parent = u
    return w

def _compute_slink(self, x, u):
    d = u.depth
    v = u.parent._get_suffix_link()
    while v.depth < d - 1:
        v = v._get_transition_link(x[u.idx + v.depth + 1])
    if v.depth > d - 1:
        v = self._create_node(x, v, d-1)
    u._add_suffix_link(v)

def _build_generalized(self, xs):
      # Builds a Generalized Suffix Tree (GST) from the input
      # array

    terminal_gen = self._terminalSymbolsGenerator()

    _xs = ''.join([x + next(terminal_gen) for x in xs])
    self.word = _xs
    self._generalized_word_starts(xs)
    self._build(_xs)
    self.root._traverse(self._label_generalized)

def _label_generalized(self, node):
      # Helper method that labels the nodes of GST with indexes
      # of strings
    if node.is_leaf():
        x = {self._get_word_start_index(node.idx)}
    else:
        x = {n for ns in node.transition_links for n in ns[0].generalized_idxs}
    node.generalized_idxs = x

def _get_word_start_index(self, idx):
      # Helper method that returns the index of the string based
      # on node's
```

```
        i = 0
        for _idx in self.word_starts[1:]:
            if idx < _idx:
                return i
            else:
                i+=1
        return i

    def lcs(self, stringIdxs=-1):
            # Returns the Largest Common Substring of stringIdxs.
        if stringIdxs == -1 or not isinstance(stringIdxs, list):
            stringIdxs = set(range(len(self.word_starts)))
        else:
            stringIdxs = set(stringIdxs)

        deepestNode = self._find_lcs(self.root, stringIdxs)
        start = deepestNode.idx
        end = deepestNode.idx + deepestNode.depth
        return self.word[start:end]

    def _find_lcs(self, node, stringIdxs):
            # Helper method that finds LCS by traversing the labeled
            # GSD
        nodes = [self._find_lcs(n, stringIdxs)
            for (n,_) in node.transition_links
            if n.generalized_idxs.issuperset(stringIdxs)]

        if nodes == []:
            return node

        deepestNode = max(nodes, key=lambda n: n.depth)
        return deepestNode

    def _generalized_word_starts(self, xs):
            # Helper method returns the starting indexes of strings in
            # GST
        self.word_starts = []
        i = 0
        for n in range(len(xs)):
            self.word_starts.append(i)
            i += len(xs[n]) + 1

    def _terminalSymbolsGenerator(self):
        # Generator of unique terminal symbols used for building
        # the Generalized Suffix Tree.

        py2 = sys.version[0] < '3'
        UPPAs = list(list(range(0xE000,0xF8FF+1)) + list(range(0xF0000,0xFFFFD+1)) +
list(range(0x100000, 0x10FFFD+1)))
        for i in UPPAs:
            if py2:
                yield(unichr(i))
```

```
            else:
                yield(chr(i))
        raise ValueError("Too many input strings.")


class _SNode():
    # Class representing a Node in the Suffix tree
    def __init__(self, idx=-1, parentNode=None, depth=-1):
        # Links
        self._suffix_link = None
        self.transition_links = []
        # Properties
        self.idx = idx
        self.depth = depth
        self.parent = parentNode
        self.generalized_idxs = {}

    def __str__(self):
        return("SNode: idx:"+ str(self.idx) + " depth:"+str(self.depth) +
            " transitons:" + str(self.transition_links))

    def _add_suffix_link(self, snode):
        self._suffix_link = snode

    def _get_suffix_link(self):
        if self._suffix_link != None:
            return self._suffix_link
        else:
            return False

    def _get_transition_link(self, suffix):
        for node,_suffix in self.transition_links:
            if _suffix == '__@__' or suffix == _suffix:
                return node
        return False

    def _add_transition_link(self, snode, suffix=''):
        tl = self._get_transition_link(suffix)
        if tl:
            self.transition_links.remove((tl,suffix))
        self.transition_links.append((snode,suffix))

    def _has_transition(self, suffix):
        for node,_suffix in self.transition_links:
            if _suffix == '__@__' or suffix == _suffix:
                return True
        return False

    def is_leaf(self):
        return self.transition_links == []

    def _traverse(self, f):
```

```
        for (node,_) in self.transition_links:
            node._traverse(f)
        f(self)

    def _get_leaves(self):
        if self.is_leaf():
            return [self]
        else:
            return [x for (n,_) in self.transition_links for x in n._get_leaves()]
```

lce.py

```
def LCE(x,i,j):
n=len(x)
length=0
while x[i+length]==x[j+length]:
          length+=1
          if j+length==n:
                    break
return length+1
```

lce_k.py

```
def LCE_k(x,i,j,k):
    p=0
    n=len(x)
    length=0
    while True:
        if x[i+length]==x[j+length]:
              length+=1
        else:
            p+=1
            if p==k+1:
                break
            length+=1

        if j+length==n:
            break

    return length
```

getBorder.py

```
# k_closed
```

```
import numpy as np

def LCE_k(x,i,j,k):
    p=0
    n=len(x)
    length=0
    while True:
        if x[i+length]==x[j+length]:
            length+=1
        else:
            p+=1
            if p==k+1:
                break
            length+=1

        if j+length==n:
            break

    return length

def Reverse(x):
    x_l=list(x)
    x_l.reverse()
    x_rl=x_l
    x_r=''.join(x_rl)

    return x_r

def LPM(x,k):
    l=len(x)

    lpm=np.arange(l)
    for i in range(l):
        lpm[i]=-1

    for j in range(l):
        if j==0:
            lpm[j]=-1
        else: lpm[j]=LCE_k(x,0,j,k)

    return lpm

def LSM(x,k):

    l=len(x)

    lsm=np.arange(l)
    for i in range(l):
        lsm[i]=-1

    x=Reverse(x)
```

```
    for j in range(l):
        if j==l-1:
            lsm[j]=-1
        else: lsm[j]=LCE_k(x,0,l-1-j,k)

    return lsm

def getBorder(x,k):

    n=len(x)
    m=0
    c=0
    con1=np.arange(n)
    con2=np.arange(n)
    con3=np.arange(n)
    con=np.arange(n)

    for i in range(n):
        con1[i]=0
        con2[i]=0
        con3[i]=0
        con[i]=0

    lpm=LPM(x,k)
    lsm=LSM(x,k)

    for j in range(n):
        r=0
        p=0
        # first condition
        if j+lpm[j]==n:
            con1[j]=1
        # second condition
        if j==0:
            con2[j]=1
        else:
            for m in range(j):
                if lpm[j]>lpm[m]:
                    r+=1
                    if r==j:
                        con2[j]=1
        # third condition
        if j==0:
            con3[j]=1
        else:
            for c in range(j):
                if lsm[n-j-1]>lsm[n-c-1]:
                    p+=1
                    if p==j:
                        con3[j]=1

    for s in range(n-1):
```

```
        con[s]=con1[s]*con2[s]*con3[s]

    return con
```

lcsub_naive.py

```
#LCSub Naive

def lcs_n(x,y):

    n=len(x)
    m=len(y)

    if n==0 or m==0:
        return ""

    if x[n-1]==y[m-1]:
        return lcs_n(x[0:n-1],y[0:m-1])+x[n-1]

    l1=lcs_n(x,y[0:m-1])
    l2=lcs_n(x[0:n-1],y)

    if len(l1)>len(l2):
        return l1
    else:
        return l2
```

lcsub_dynamic.py

```
# LCSub_dynamic

def lcs(x,y):

    n=len(x)
    m=len(y)
    L=[[0 for v in range(m+1)]for w in range(n+1)]
    #print L
    for i in range(n+1):
        for j in range(m+1):
            if i==0 or j==0:
                L[i][j]=0

            elif x[i-1]==y[j-1]:
                L[i][j]=L[i-1][j-1]+1

            else:
                L[i][j]=max(L[i-1][j], L[i][j-1])
```

```
lcs=[""]
i=n
j=m
while i>0 and j>0:
    if x[i-1]==y[j-1]:
        lcs.append(x[i-1])
        i-=1
        j-=1
    elif L[i-1][j]>L[i][j-1]:
        i-=1
    else:
        j-=1

return "".join(reversed(lcs))
```