
Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Υπολογιστών και Πληροφορικής

# Αλγόριθμοι Ανάλυσης Δεδομένων Next Generation Sequencing

Διπλωματική Εργασία

Ζάκκας Κώστας

**Επιβλέπων:** Μακρής Χρήστος
Αναπληρωτής Καθηγητής
**Συνεπιβλέπων:** Καναβός Ανδρέας
Μεταδιδακτορικός Ερευνητής

Πάτρα, Οκτώβριος 2018


University of Patras
Polytechnic School
Department of Computer Engineering and Informatics

# Next Generation Sequencing Data Analysis Algorithms

Diploma Thesis

Zakkas Kostas

**Supervisor:** Makris Christos
Associate Professor
**Co-supervisor:** Kanavos Andreas
Postdoctoral researcher

Patras, October 2018

# Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον αναπληρωτή καθηγητή κ. Μακρή Χρήστο και τον μεταδιδακτορικό ερευνητή κ. Καναβό Ανδρέα για την καθοδήγηση και την συνεχή βοήθεια τους στη συγγραφή της παρούσας διπλωματικής.

# Πίνακας Περιεχομένων

# Περίληψη

Η μεγάλη αύξηση των βιολογικών δεδομένων τις τελευταίες δύο δεκαετίες έχει κάνει ακόμα πιο επιτακτική την ανάγκη μελέτης αποτελεσματικών αλγορίθμων για την ανάλυση και εξόρυξη πληροφορίας από τα δεδομένα αυτά. Η απομόνωση του DNA από το βιολογικό περιβάλλον του, μας επιτρέπει να το αντιμετωπίσουμε ως ένα μονοδιάστατο string χαρακτήρων και να εφαρμόσουμε έτσι διάφορους αλγορίθμους ανάλυσης string. Η μελέτη και ανάπτυξη αποτελεσματικών αλγορίθμων για την εύρεση ομοιότητας ανάμεσα σε αλληλουχίες DNA αποτελεί ένα σημαντικό στόχο της βιοπληροφορικής. Συνήθως τα προβλήματα με strings στην βιοπληροφορική είναι ένα από τα δομικά στοιχεία για τον υπολογισμό άλλων προβλημάτων. Στην παρούσα διπλωματική εξετάζονται διάφοροι αλγόριθμοι για την επίλυση προβλημάτων με strings βιολογικών δεδομένων στη μηχανή επεξεργασίας δεδομένων μεγάλης κλίμακας Apache Spark και σε γλώσσα προγραμματισμού Python. Αρχικά μελετώνται αλγόριθμοι επίλυσης του Longest Common Prefix (LCP) προβλήματος που προσφέρει χρήσιμες πληροφορίες για την ανάλυση δεδομένων χαρακτήρων στη μοριακή βιολογία και τον εντοπισμό διαφοροποιήσεων σε ακολουθίες που μπορεί να είναι αποτέλεσμα της αντιγραφής του DNA ή σφαλμάτων ακολουθίας DNA. Έπειτα εξετάζονται αλγόριθμοι επίλυσης του Longest Common Substring (LCS) προβλήματος το οποίο είναι βασικό πρόβλημα στην μελέτη των strings και εμφανίζεται σε πολλές διαφορετικές περιπτώσεις στην βιολογία. Στη συνέχεια γίνεται μελέτη αλγορίθμων για την επίλυση του Longest Common Subsequence (LCSub) προβλήματος το οποίο βρίσκει εφαρμογή σε διάφορα προβλήματα στη γενετική και στην μοριακή βιολογία ενώ χρησιμοποιείται σαν μέτρο ομοιότητας των strings, οπότε και των βιολογικών ακολουθιών που αντιπροσωπεύουν. Τέλος αναλύονται αλγόριθμοι επίλυσης του Longest Common Extension (LCE) προβλήματος το οποίο εμφανίζεται σαν υποπρόβλημα σε διάφορα θεμελιώδη προβλήματα με strings όπως το k-Difference Global Alignment για την κατασκευή εργαλείων στοίχισης στην βιοπληροφορική.

**Λέξεις-κλειδιά:** string, βιοπληροφορική, αλγόριθμοι, LCP, LCS, LCSub, LCE

# Abstract

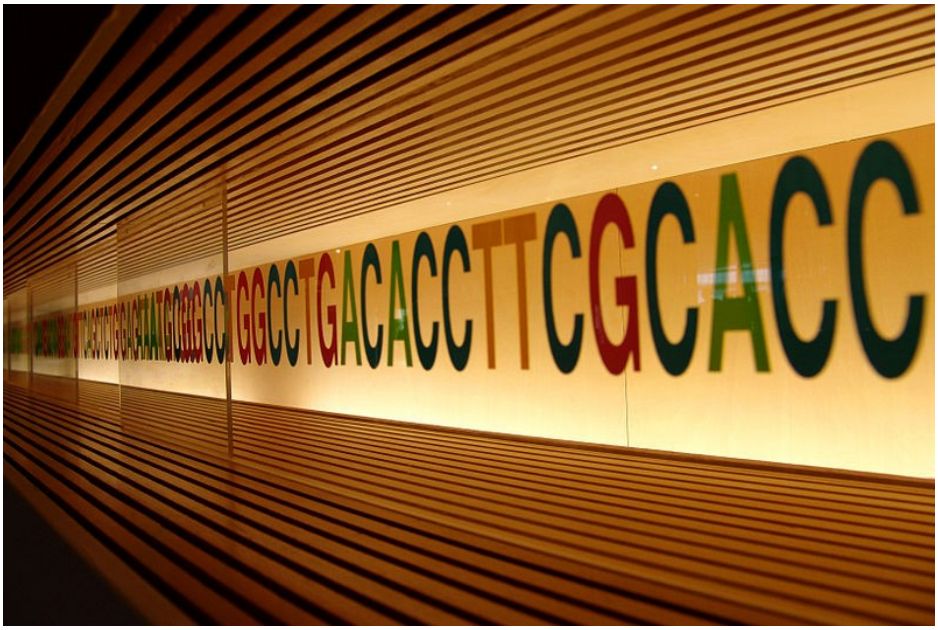
The large increase in biological data over the past two decades has made the need to study efficient algorithms for analyzing and extracting information from these data even more necessary. The isolation of DNA from its biological environment allows us to treat it as a one-dimensional string characters and implement different string analysis algorithms. The study and developing of effective algorithms that find similarity between DNA sequences are an important objective of bioinformatics. Usually, problems with strings in bioinformatics are one of the building blocks for calculating other problems. In the present diploma thesis various algorithms for solving problems with biological data strings are examined in the Apache Spark large-scale data processing engine and in Python programming language. Initially, we study solving algorithms for the Longest Common Prefix (LCP) problem which provides useful information for character data analysis in molecular biology and identification of sequence variations that may be the result of DNA replication or DNA sequence errors. Then we examine algorithms that solve the Longest Common Substring (LCS) problem which is a major problem in the study of strings and it occurs in many different cases in biology. Algorithms are then studied to solve the Longest Common Subsequence (LCSub) problem that addresses various problems in genetics and molecular biology while being used as a measure of similarity between the strings and the biological sequences they represent. Finally, we analyze Longest Common Extension (LCE) problem solving algorithms, a problem that appears as a sub-problem in several fundamental problems with strings such as the k-Difference Global Alignment for the construction of alignment tools in bioinformatics.

**Key-words:** string, bioinformatics, algorithms, LCP, LCS, LCSub, LCE

# 1. Εισαγωγή

## 1.1. DNA και πληροφορική

Το DNA είναι νουκλεικό οξύ με μορφή διπλής έλικας, το οποίο αποτελείται από χαρακτηριστικά τμήματα του γενετικού υλικού που είναι μοναδικά σε κάθε άτομο. Αποτελεί τη γενετική ταυτότητα κάθε ατόμου και η κατανόηση της λειτουργίας του επιτρέπει στους επιστήμονες να κατανοήσουν καλύτερα τη γενετική της ζωής και την κληρονόμηση ορισμένων χαρακτηριστικών και νόσων. Όταν οι βιολόγοι βρίσκουν μια νέα ακολουθία DNA θέλουν να διαπιστώσουν με ποιες άλλες γνωστές ακολουθίες έχει ομοιότητες. Η σύγκριση ακολουθιών έχει χρησιμοποιηθεί επιτυχώς για την ανάπτυξη σύνδεσης ανάμεσα σε γονίδια τα οποία είναι υπεύθυνα για την εμφάνιση καρκίνου και γονίδια που είναι σχετικά με την φυσιολογική ανάπτυξη.




**Εικόνα 1.1:** Η ακολουθία DNA αναπαρίσταται σαν μια ακολουθία χαρακτήρων (A, C, G, T)

Πολλά χρήσιμα συμπεράσματα για την επιστήμη της βιολογίας μπορούν να προκύψουν αν απομονώσουμε το DNA από το βιολογικό περιβάλλον του, όπου λειτουργεί ως ένα μόριο τριών διαστάσεων σε αλληλεπίδραση με το RNA και με πρωτείνες και το αντιμετωπίσουμε ως ένα μονοδιάστατο string χαρακτήρων. Κάνοντας αυτή την <<υπόθεση>> και αφού πλέον η βιολογική ακολουθία έχει αναπαρασταθεί σαν μια ακολουθία χαρακτήρων (A-Adenine (Αδενίνη), C-Cytosine (Κυτοσίνη), G-Guanine (Γουανίνη), T-Thymine (Θυμίνη)), δίνεται η δυνατότητα σε μη βιολόγους να ασχοληθούν με θέματα που σχετίζονται με το

DNA και συγκεκριμένα από την σκοπιά της πληροφορικής ανοίγει ένα πολύ ενδιαφέρον και σημαντικό πεδίο έρευνας.

## 1.2. Βιοπληροφορική

Η βιοπληροφορική είναι μια υβριδική επιστήμη που συνδέει βιολογικά δεδομένα με τεχνικές αποθήκευσης, διανομής και ανάλυσης πληροφοριών για την υποστήριξη διαφόρων τομέων επιστημονικής έρευνας όπως π.χ. της βιοϊατρικής. Η βιοπληροφορική τροφοδοτείται με δεδομένα που προκύπτουν από πειράματα σχετικά με προσδιορισμούς γονιδιωματικής αλληλουχίας και μετρήσεις μοτίβων γονιδιακής έκφρασης. Η εξόρυξη αυτών των δεδομένων οδηγεί σε επιστημονικές ανακαλύψεις και στον εντοπισμό νέων κλινικών εφαρμογών όπως για παράδειγμα εντοπισμός συσχετίσεων ανάμεσα σε αλληλουχίες γονιδίων και ασθένειες, πρόβλεψη πρωτεϊνικών δομών από αλληλουχίες αμινοξέων, διευκόλυνση του σχεδιασμού νέων φαρμάκων και προσαρμογή των θεραπειών σε διαφορετικούς ασθενείς με βάση τις αλληλουχίες του DNA τους (φαρμακογονιδιωματική).


**Εικόνα 1.2:** Το abstract word cloud της βιοπληροφορικής

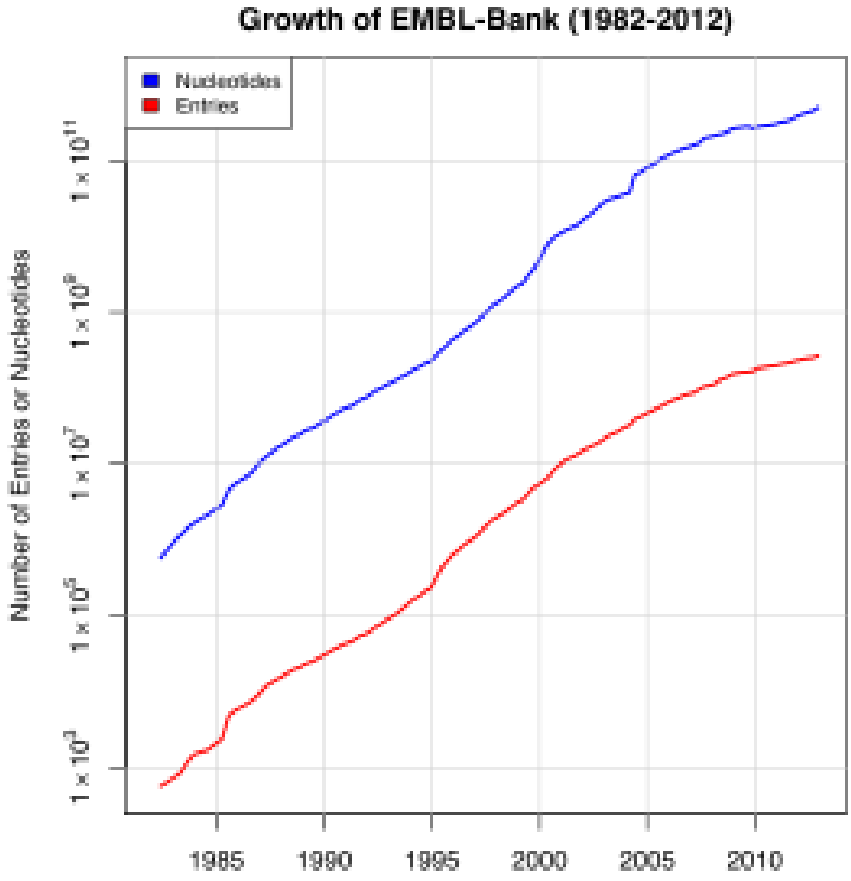
Τα δεδομένα που χρησιμοποιούνται κλασικά στην βιοπληροφορική περιλαμβάνουν αλληλουχίες DNA από γονίδια ή ολόκληρα γονιδιώματα, αλληλουχίες αμινοξέων σε πρωτεΐνες και τρισδιάστατες δομές πρωτεϊνών, νουκλεικών οξέων και συμπλεγμάτων πρωτεΐνο-νουκλεικών οξέων. Επιπλέον ροές δεδομένων περιλαμβάνουν: μεταγραφική μικρογραφία, πρότυπο σύνθεσης RNA από DNA, πρωτεϊνωματική, η κατανομή των

πρωτεϊνών στα κύτταρα, αλληλεπιδρωματική, πρότυπα αλληλεπιδράσεων ανάμεσα σε πρωτεϊνες και νουκλεϊκά οξέα, μεταβολιμική, δηλαδή η φύση και τα πρότυπα κίνησης των μετασχηματισμών μικρών μορίων από τις βιοχημικές οδούς οι οποίες είναι ενεργές στα κύτταρα. Σε όλες τις περιπτώσεις το ενδιαφέρον επικεντρώνεται στην λήψη περιεκτικών και ακριβών δεδομένων για συγκεκριμένους τύπους κυττάρων και στον εντοπισμό προτύπων παραλλαγών μέσα στα δεδομένα.

## 1.3. Παράγοντες εξελίξεων στην βιοπληροφορική και κατευθύνσεις που την οδήγησαν

Η βιοπληροφορική έχει καθοριστεί από την μεγάλη αύξηση παραγωγής δεδομένων στην βιολογία. Οι μέθοδοι επεξεργασίας αλληλουχιών γονιδιώματος έχουν παίξει μεγάλο ρόλο σε αυτό.Την δεκαετία του 1950 η ανάπτυξη της βιοπολυμερούς αλληλούχισης και ο ψηφιακός υπολογιστής ξεκίνησαν μια ψηφιακή επανάσταση στις βιοεπιστήμες. Έπειτα στα τέλη της δεκαετίας του 1970 η εμφάνιση των προσωπικών υπολογιστών (PC) και η αλληλούχιση του Sanger οδήγησε σε μια σημαντική ποσότητα δεδομένων αλληλουχίας η οποία αποθηκεύτηκε σε βάσεις δεδομένων και ενώθηκε εννοιολογικά μέσα σε ένα υπολογιστικό πλαίσιο. Στη δεκαετία του 1990 η άνοδος του διαδικτύου διευκόλυνε την αύξηση διαμοιρασμού των δεδομένων ενώ πλέον οι τεχνικές ανάλυσης άρχισαν να στρέφονται σε προγράμματα που φιλοξενούνται σε ιστότοπους. Στα μέσα της δεκαετίας του 2000 μεγάλη αλλαγή επήλθε με την εμφάνιση του cloud computing και του Next Generation Sequencing (NGS). Η αλλαγή αυτή οδήγησε σε δραματική αύξηση της κλίμακας των συνόλων δεδομένων. Το 1999 τα αρχεία αλληλουχιών νουκλεϊνικών οξέων περιείχαν συνολικά 3.5 δισεκατομμύρια νουκλεοτίδια, μόνο λίγο μεγαλύτερος αριθμός από το μήκος του ανθρώπινου γονιδιώματος. Μετά από 10 χρόνια ο αριθμός αυτός αυξήθηκε σε 283 δισεκατομμύρια νουκλεοτίδια δηλαδή το μήκος 95 ανθρώπινων γονιδιωμάτων.

Η μεγάλη αυτή αύξηση δεδομένων δημιούργησε την ανάγκη για αλλαγές στις υποδομές αποθήκευσης. Οι βάσεις δεδομένων όπως το Ευρωπαϊκό Αρχείο Νουκλεοτιδίων (ENA) και το Αρχείο Ανάγνωσης Ακολουθιών (SRA) δημιουργήθηκαν για την αποτελεσματική οργάνωση και αποθήκευση δεδομένων αλληλουχίας. Αυτά τα σύνολα δεδομένων δημιουργούν νέες προκλήσεις καθώς είναι υπερβολικά μεγάλα για τις παλιές τεχνικές ανάλυσης και διαμοιρασμού, όμως οι πρόσφατες εξελίξεις στις υπολογιστικές τεχνολογίες και προσεγγίσεις, ειδικά η άνοδος του cloud computing, παρέχουν δυνατότητες διαχείρισης των τεράστιων ποσοτήτων δεδομένων αλληλούχισης που παράγονται διαρκώς. Εργαλεία στοίχισης έχουν αναπτυχθεί παράλληλα με την τεχνολογία αλληλούχισης για τις ανάγκες επεξεργασίας δεδομένων αλληλουχίας. Η μείωση στον χρόνο εκτέλεσης τους ακολουθεί κατά προσέγγιση τον νόμο του Moore.

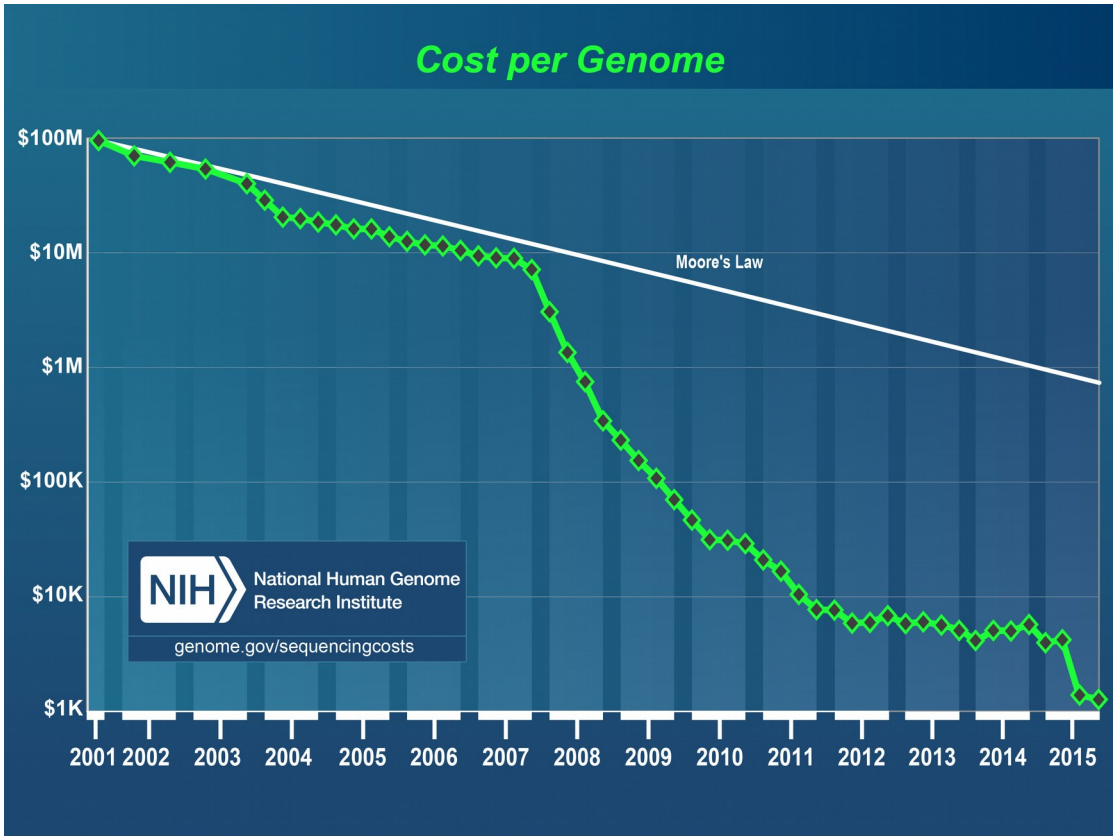



**Εικόνα 1.3:** Αύξηση της βάσης δεδομένων αλληλουχιών νουκλεοτιδίων του EMBL

Η βελτιωμένη απόδοση καθοδηγείται από την πρόοδο που επήλθε σε μια σειρά διακριτών αλγορίθμων. Αρχικά στην εποχή της αλληλούχισης Sanger, οι αλγόριθμοι Smith-Waterman και Needleman-Wunsch βασίζονταν στον δυναμικό προγραμματισμό για να βρουν μια τοπική ή ολική βέλτιστη στοίχιση. Όμως η πολυπλοκότητα αυτών των προσεγγίσεων καθιστά αδύνατη τη χαρτογράφηση αλληλουχιών σε μεγάλα γονιδιώματα. Με βάση αυτόν τον περιορισμό πολλοί αλγόριθμοι αναπτύχθηκαν με βελτιστοποιημένες δομές δεδομένων, χρησιμοποιώντας πίνακες κατακερματισμού (για παράδειγμα Fasta, BLAST, BLAT (παρόμοιο με το BLAST), MAQ και Novoalign) ή suffix arrays με τον μετασχηματισμό Burrows-Wheeler (για παράδειγμα STAR (Spliced Transcripts Alignment to a Reference), BWA (Burrows-Wheeler Aligner) και Bowtie.

Παρά την τεράστια αύξηση των βιολογικών δεδομένων η πρόοδος στον τομέα της γονιδιωματικής οδήγησε σε σημαντική μείωση του κόστους της αλληλουχίας του γονιδιώματος. Τα Εθνικά Ινστιτούτα Υγείας των Η.Π.Α. έθεσαν ως στόχο στους ερευνητές να μειωθεί το κόστος της αλληλουχίας ενός ανθρώπινου γινιδιώματος σε 1000$. Κάτι τέτοιο θα έδινε την δυνατότητα η αλληλουχία DNA να είναι ένα προσιτό εργαλείο για νοσοκομεία και κλινικές ώστε να υπάρχει πιο αποτελεσματική διάγνωση ασθενειών. Ως αποτέλεσμα έχει δοθεί σημαντική προσοχή στα έξοδα για την δημιουργία αλληλουχίας

γονιδιώματος και στον τρόπο υπολογισμού τους. Με την αυξανόμενη κλίμακα μελετών ανθρώπινης γενετικής και τον αυξανόμενο αριθμό κλινικών εφαρμογών για την αλληλούχιση γονιδιώματος δίνεται μεγαλύτερη ακόμα προσοχή στην κατανόηση του κόστους δημιουργίας αλληλουχίας ανθρώπινου γονιδιώματος.




**Εικόνα 1.4:** Μείωση κόστους ανά γονιδίωμα σύμφωνα με το NHGRI

Η μείωση του κόστους αλληλούχισης και ο αυξανόμενος αριθμός των αλληλουχιών δημιουργούν μεγαλύτερες απαιτήσεις από τους υπολογιστικούς πόρους και τις γνώσεις που χρειάζονται για την διαχείριση των δεδομένων ακολουθίας. Είναι πολύ σημαντικό όσο το ποσό των δεδομένων αλληλουχίας συνεχίζει να αυξάνεται, τα δεδομένα αυτά να μην αποθηκεύονται απλά αλλά να οργανώνονται με τρόπο τόσο κλιμακωτό όσο και εύκολα προσβάσιμο για την ευρύτερη ερευνητική κοινότητα. Η βιοπληροφορική για να ανταποκριθεί στις αυξανόμενες απαιτήσεις ακολουθεί νέες κατευθύνσεις ώστε να προσαρμοστεί στις ραγδαίες αλλαγές. Μία κατεύθυνση που έρχεται ως απάντηση στα συνεχώς αυξανόμενα γονιδιώματα και σύνολα αλληλουχιών είναι η εξέλιξη των αλγορίθμων στοίχισης. Επίσης δημιουγείται η ανάγκη συμπίεσης για τον χειρισμό μεγάλου μεγέθους αρχείων. Ιδιαίτερα η ανάγκη συμπίεσης που εκμεταλλεύεται τις γνώσεις που υπάρχουν και είναι συγκεκριμένες για την ανάλυση των ακολουθιών δεδομένων, με στόχο την επίτευξη καλύτερων αποτελεσμάτων από αυτά που παρέχονται από τους πιο γενικούς αλγορίθμους συμπίεσης. Μία άλλη αλλαγή που προκαλείται περιλαμβάνει την ανάγκη για κατανεμημένο και παράλληλο υπολογισμό στο cloud ώστε να είναι ευκολότερη η διαχείριση

μεγάλων ποσοτήτων δεδομένων και ταυτόχρονα να αναλύονται αξιόπιστα. Επίσης καθώς μελλοντικά ένας μεγάλος όγκος δεδομένων αλληλουχίας θα είναι ιδιωτικά δεδομένα προσώπων που ταυτοποιούνται μέσω αυτών προκύπτει το θέμα της ασφάλειας. Συνεπώς υπάρχει ανάγκη να τεθούν σε εφαρμογή πρωτόκολλα για την ασφάλεια τέτοιων δεδομένων, ιδιαιτέρως σε περιβάλλον υπολογισμών στο cloud.

## 1.4. Στόχοι βιοπληροφορικής

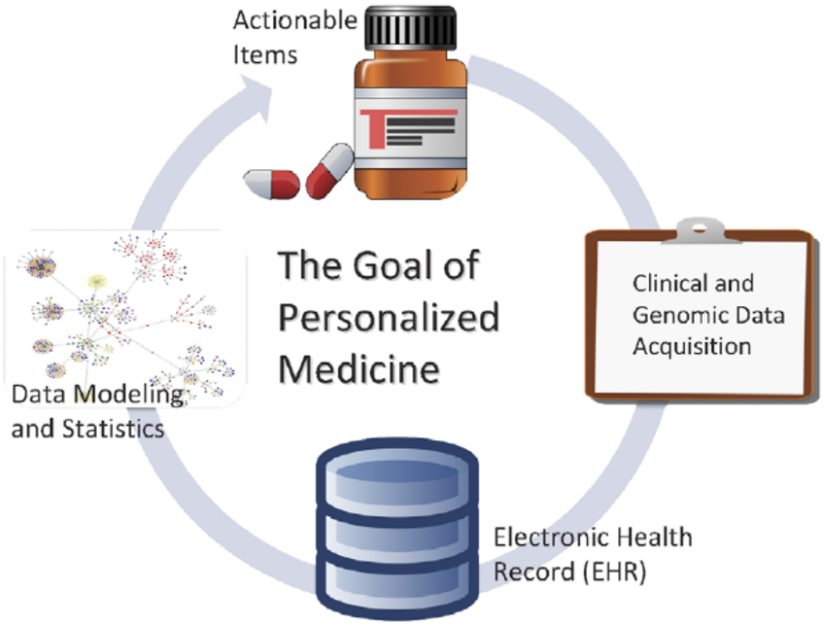
Αρχικά μεγάλο μέρος της έρευνας στην βιοπληροφορική ήταν στενά εστιασμένο και στόχευε κυρίως στην δημιουργία αλγορίθμων ανάλυσης συγκεκριμένων τύπων DNA όπως αλληλουχίες γονιδίων ή πρωτεϊνικές δομές. Πλέον όμως η βιοπληροφορική έχει ευρύτερους στόχους και αποσκοπούν στην εύρεση τρόπου με τον οποίο διαφορετικοί τύποι δεδομένων να συνδυαστούν ώστε να είναι δυνατή η κατανόηση φυσικών φαινομένων συμπεριλαμβανομένων των οργανισμών και των ασθενειών.

Ένας σημαντικός στόχος της βιοπληροφορικής είναι η ανάπτυξη αποτελεσματικών αλγορίθμων για μέτρηση της ομοιότητας μεταξύ αλληλουχιών. Ο αλγόριθμος Needleman-Wusch βασίζεται στον δυναμικό προγραμματισμό και εγγυάται την εύρεση της βέλτιστης στοίχισης των ζευγαριών ακολουθιών χωρίζοντας το πρόβλημα (ακολουθία) σε μικρότερα (τμήματα της ακολουθίας) και χρησιμοποιώντας τις λύσεις των μικρότερων προβλημάτων για κατασκευή της τελικής λύσης του αρχικού προβλήματος. Παρότι ο αλγόριθμος Needleman-Wunsch είναι αποτελεσματικός είναι πολύ αργός για χρήση με μεγάλη βάση δεδομένων ακολουθιών. Οπότε έχει δοθεί μεγάλη βάση στην εύρεση γρήγορων αλγορίθμων που μπορούν να είναι αποδοτικοί για τα τεράστια ποσά πληροφορίας που υπάρχουν πλέον στις βάσεις δεδομένων. Ένα παράδειγμα είναι το πρόγραμμα BLAST (Basic Local Alignment Search Tool) το οποίο εντοπίζει περιοχές τοπικής ομοιότητας ανάμεσα σε ακολουθίες συγκρίνοντας τις ακολουθίες νουκλεοτιδίων ή πρωτεϊνών με βάσεις δεδομένων ακολουθιών και υπολογίζει την στατιστική σημασία των ταιριασμάτων. Μία εξέλιξη του BLAST το PSI-(position specific iterated-) BLAST χρησιμοποιεί τα πρότυπα συντήρησης σε σχετικές ακολουθίες και συνδυάζει την υψηλή ταχύτητα του BLAST με πολύ υψηλή ευαισθησία για να βρει σχετικές ακολουθίες.

Παράλληλα στοχεύει και στην ανάπτυξη εργαλείων όπως το PSI-BLAST που αναφέρθηκε παραπάνω ή το FASTA καθώς και πόρων που βοηθούν στην ανάλυση δεδομένων. Η ανάπτυξη τέτοιων πόρων απαιτεί εμπειρία στην θεωρία υπολογιστών και πλήρη κατανόηση εννοιών της βιολογίας. Έπειτα τα αποτελέσματα της ανάλυσης πρέπει να ερμηνεύονται με τρόπο βιολογικά χρήσιμο. Παραδοσιακά οι βιολογικές μελέτες εξέταζαν με λεπτομέρεια μεμονωμένα συστήματα και τα σύγκριναν με άλλα με τα οποία σχετίζονταν. Πλέον η βιοπληροφορική προσπαθεί να επεκτείνει την ανάλυση σε παγκόσμιο επίπεδο συμπεριλαμβάνοντας όλα τα διαθέσιμα δεδομένα με στόχο να αποκαλυφθούν κοινές αρχές που ισχύουν για διάφορα συστήματα και να αναδειχθούν νέα χαρακτηριστικά.

Ένας άλλος πρωταρχικός στόχος της είναι η ανάπτυξη ιδεών και τεχνολογίας που απαιτούνται για την ικανοποιητική συλλογή πληροφοριών, οι οποίες χρησιμοποιούνται για

τον ποσοτικό προσδιορισμό της βιολογικής λειτουργίας και την ανάπτυξη λεπτομερών προγνωστικών βιοϊατρικών μοντέλων. Αυτά θα χρησιμοποιηθούν για να περιγράψουν με ακρίβεια την κανονική φυσιολογική λειτουργία, να γίνουν κατανοητοί οι μηχανισμοί πίσω από τις παθολογικές διεργασίες, να ανακαλυφθούν νέοι βιοδείκτες και θεραπευτικοί στόχοι για ασθένειες και να γίνει αξιολόγηση των ιατρικών και χειρουργικών παρεμβάσεων. Επίσης η βιοπληροφορική είναι υπεύθυνη για την οργάνωση αυτών των δεδομένων με τέτοιο τρόπο ώστε να είναι ευκολότερη για τους ερευνητές η πρόσβαση στην υπάρχουσα πληροφορία καθώς και η καταχώρηση νέων δεδομένων που παράγονται. Παρόλο που η επεξεργασία δεδομένων είναι βασικό καθήκον της, οι πληροφορίες που είναι αποθηκευμένες σε βάσεις δεδομένων είναι ουσιαστικά άχρηστες μέχρι να αναλυθούν. Έτσι ο σκοπός της βιοπληροφορικής επεκτείνεται πολύ περισσότερο. Επιπλέον η βιοπληροφορική, συγκεκριμένα η μεταφραστική βιοπληροφορική, είναι ένα πεδίο που μπορεί να συμβάλλει σημαντικά στην ανάπτυξη της εξατομικευμένης ιατρικής. Η εξατομικευμένη ιατρική είναι ένα μοντέλο υγειονομικής περίθαλψης που είναι προγνωστικό, εξατομικευμένο, προληπτικό και συμμετοχικό.




**Εικόνα 1.5:** Η βιοπληροφορική παίζει σημαντικό ρόλο στην ανάπτυξη της εξατομικευμένης ιατρικής

Μέσω της βιοπληροφορικής μπορούν να αντιμετωπιστούν οι προκλήσεις που θέτει η εξατομικευμένη ιατρική μέσω της ανάπτυξης μεθόδων αποθήκευσης, ανάλυσης και ερμήνευσης για βελτιστοποίηση του μετασχηματισμού των αυξανόμενων βιοϊατρικών δεδομένων σε προληπτική, προγνωστική και συμμετοχική υγειονομική φροντίδα.

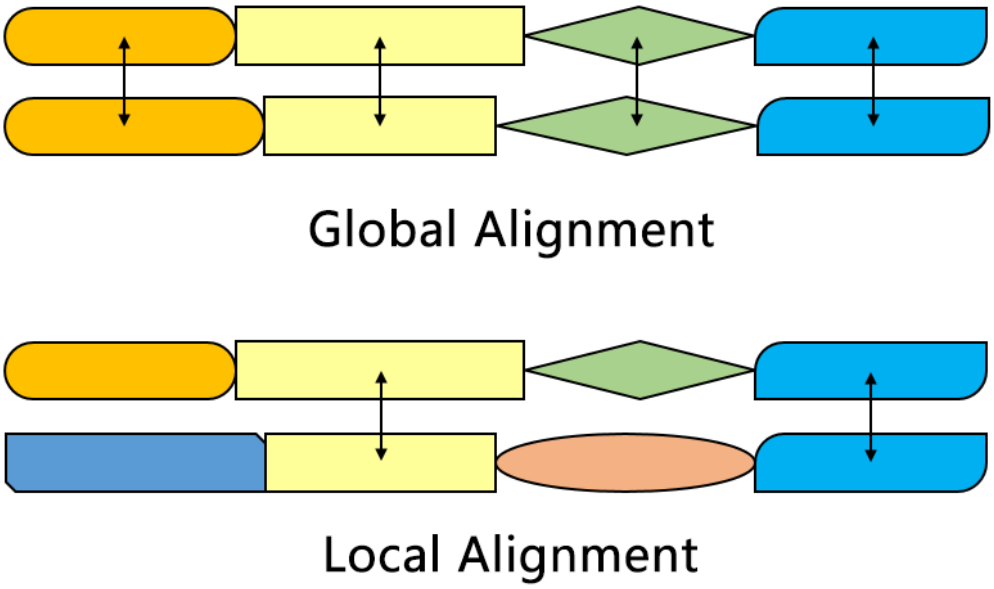
Ένας άλλος στόχος της βιοπληροφορικής είναι η επέκταση των πειραματικών δεδομένων από προβλέψεις. Η πρόβλεψη της πρωτεϊνικής δομής από μια αλληλουχία αμινοξέων είναι ένας θεμελιώδης στόχος της υπολογιστικής βιολογίας. Η αυθόρμητη αναδίπλωση

πρωτεϊνών δείχνει ότι κάτι τέτοιο είναι εφικτό. Η πρόοδος στην ανάπτυξη τεχνικών ώστε να προβλέπεται η αναδίπλωση πρωτεϊνών μετριέται με προγράμματα π.χ. CASP ανά διετία. Επιπλέον η βιοπληροφορική στοχεύει στη πρόβλεψη αλληλεπιδράσεων ανάμεσα σε πρωτεΐνες, με βάση ατομικές τους δομές, γνωστό ως πρόβλημα σύνδεσης. Τα συμπλέγματα ανάμεσα σε πρωτεΐνες έχουν καλή συμπληρωματικότητα στο σχήμα της επιφάνειας και στην πολικότητα και σταθεροποιούνται κυρίως μέσα από αδύναμες αλληλεπιδράσεις όπως για παράδειγμα δεσμοί υδρογόνου ή δυνάμεις van der Waals. Ο σκοπός των προγραμμάτων είναι να δημιουργήσουν μια προσομοίωση αυτών των αλληλεπιδράσεων ώστε να προβλέψουν την βέλτιστη χωρική σχέση μεταξύ των μερών της αλληλεπίδρασης.

## 1.5. Προβλήματα με strings στην βιοπληροφορική

Τα προβλήματα με strings στην βιοπληροφορική συνήθως είναι ένα από τα πολλά δομικά στοιχεία για υπολογισμούς άλλων προβλημάτων. Τυπικά τα αποτελέσματά τους χρησιμοποιούνται σε διαδικασίες υψηλότερου επιπέδου όπως ταξινόμηση, πρόβλεψη δομής, φυλογενετικά δέντρα ή ευθυγραμμίσεις πολλαπλών ακολουθιών. Η περιοχή της κατά προσέγγιση αντιστοίχισης και της σύγκρισης αλληλουχιών είναι κεντρικό θέμα στην υπολογιστική μοριακή βιολογία τόσο εξαιτίας της παρουσίας σφαλμάτων στα μοριακά δεδομένα, όσο και λόγω της ύπαρξης ενεργών λειτουργιών που προκαλούν μεταλλάξεις. Οι μέθοδοι σύγκρισης ακολουθιών σκοπό έχουν να αποκαλύψουν και να μοντελοποιήσουν αυτές τις περιπτώσεις.

Η στοίχιση ακολουθιών έχει γίνει το κεντρικό εργαλείο για την σύγκριση αλληλουχίας στην μοριακή βιολογία. Συγκεκριμένα μεγάλο ενδιαφέρον παρουσιάζουν οι στοιχίσεις αλληλουχιών πρωτεϊνών, καθώς αυτές οι στοιχίσεις παρέχουν σημαντικές πληροφορίες της γονιδιακής και πρωτεϊνικής λειτουργίας. Υπάρχουν πολλοί άλλοι διαφορετικοί τύποι στοίχισης όπως ολικές στοιχίσεις σε ζεύγη πρωτεϊνών που σχετίζονται με την κοινή καταγωγή καθ’ όλην την έκταση του μήκους τους, τοπικές στοιχίσεις που περιλαμβάνουν σχετιζόμενα τμήματα πρωτεϊνών, πολλαπλές στοιχίσεις μελών πρωτεϊνικών οικογενειών και στοιχίσεις που έγιναν στην διάρκεια αναζητήσεων σε βάσεις δεδομένων για την ανίχνευση ομογενειών. Η σύγκριση αλληλουχιών, ειδικά όταν συνδυάζεται με τη συστηματική συλλογή, διόρθωση και αναζήτηση βάσεων δεδομένων που περιέχουν βιομοριακές αλληλουχίες, έχει γίνει αναπόσπαστο κομμάτι της σύγχρονης μοριακής βιολογίας. Ένα γεγονός που εξηγεί την σημασία των μοριακών δεδομένων ακολουθίας και της σύγκρισης αλληλουχιών στην βιολογία είναι ότι σε βιομοριακές αλληλουχίες (DNA, RNA ή αλληλουχίες αμινοξέων) η μεγάλη ομοιότητα μεταξύ αλληλουχιών συνήθως συνεπάγεται και μεγάλη ομοιότητα στη δομή και σε σημαντικές λειτουργίες. Στην εξελικτική διαδικασία οι επιτυχημένες δομές (πρωτεΐνες, ρυθμιστικές αλληλουχίες DNA, μορφολογικά χαρακτηριστικά κλπ) επαναχρησιμοποιούνται, αντιγράφονται, τροποποιούνται και χρησιμοποιούνται σαν δομικά υλικά κατασκευής άλλων δομών. Οι ίδιες και συναφείς μοριακές δομές και μηχανισμοί εμφανίζονται επανειλημμένα στο γονιδίωμα ενός μόνο είδους και σε ένα πολύ ευρύ φάσμα διαφορετικών ειδών.




**Εικόνα 1.6:** Ολική και τοπική στοίχιση ακολουθιών

Το πρόβλημα εύρεσης του Closest string αρχικά εισήχθη και μελετήθηκε στα πλαίσια της βιοπληροφορικής. Η λύση του βασίζεται στην εύρεση ενός κοινού μοτίβου σε πολλά, αλλά όχι σε όλα απαραίτητα, τα επιθυμητά strings και είναι σημαντική καθώς εμφανίζεται σε πολλές εφαρμογές στην βιοπληροφορική. Ένα παρόμοιο πρόβλημα είναι και το Farthest String Problem μόνο που αντί της αναζήτησης της μικρότερης Hamming απόστασης μεταξύ μιας τιμής και ενός συνόλου strings, τώρα θα αναζητηθεί η μεγαλύτερη. Η εύρεση ύπαρξης παρόμοιων περιοχών ανάμεσα σε πολλές DNA, RNA ή πρωτεϊνικές ακολουθίες παίζει σημαντικό ρόλο σε εφαρμογές όπως ο σχεδιασμός καθολικών εκκινητών PCR, σχεδιασμός γενετικού διερευνητή, σχεδιασμός αντινοηματικών φαρμάκων, εύρεση μεταγραφικού παράγοντα, αναγνώριση μοτίβου και προσδιορισμός οικογένειας πρωτεϊνών. Το closed string χρησιμοποιείται συχνά για να μοντελοποιήσει αυτές τις εφαρμογές. Τέλος ένα ακόμα πρόβλημα με εφαρμογές στην αλληλούχιση γονιδιώματος είναι το Minimum Common Partition Problem (MCSP). Για δύο αλληλουχίες DNA το MCSP αναζητά το μικρότερο σύνολο των κοινών δομικών στοιχείων των ακολουθιών, που είναι ένα πρόβλημα το οποίο σχετίζεται στενά με την σύγκριση και αναδιάταξη ακολουθιών γονιδιώματος. Άλλα προβλήματα με strings που εμφανίζονται συχνά στην βιοπληροφορική είναι η εύρεση Tandem Repeats, η πρόβλεψη της δομής βιομορίων, το median string πρόβλημα που σχετίζεται με το πρόβλημα της πολλαπλής στοίχισης, η ανίχνευση κατά προσέγγιση επικάλυψης για συναρμολόγηση ακολουθιών κλπ.

## 1.6. Περιεχόμενα διπλωματικής εργασίας

Σκοπός της παρούσας διπλωματικής είναι η θεωρητική και πρακτική μελέτη αλγορίθμων και δομών δεδομένων για την αποδοτική ανάλυση μεγάλου πλήθους ακολουθιών DNA στην κύρια μνήμη υπολογιστικών συστημάτων. Θα γίνει εστίαση στα προβλήματα της εξαγωγής του μέγιστου κοινού προθέματος-Longest Common Prefix (LCP), της μέγιστης κοινής υπο-συμβολοσειράς-Longest Common Substring (LCS), της μέγιστης κοινής υποακολουθίας-Longest Common Subsequence (LCSub) και εύρεση του μέγιστου κοινού extension-Longest Common Extension (LCE). Για την μελέτη των αλγορίθμων θα χρησιμοποιηθεί το περιβάλλον επεξεργασίας μεγάλου όγκου δεδομένων Apache Spark σε γλώσσα Python. Για την επαλήθευση της ορθής λειτουργίας των αλγορίθμων και των χρονικών πολυπλοκοτήτων τους, θα εκτελεστούν πειράματα με χρήση βιολογικών δεδομένων. Ο κώδικας που χρησιμοποιείται υλοποιήθηκε στα πλαίσια της εργασίας ενώ χρησιμιμοποιείται και μέρος κώδικα που προυπήρχε και τροποποιήθηκε κατάλληλα. Αρχικά εξετάζεται το πρόβλημα Longest Common Prefix, το οποίο χρησιμοποιείται στην ανάλυση δεδομένων χαρακτήρων στην μοριακή βιολογία και στον εντοπισμό διαφοροποιήσεων στο DNA ως αποτέλεσμα της αντιγραφής του. Οι αλγόριθμοι που θα αναλύσουμε και υλοποιήσουμε δέχονται στην είσοδο τους δύο ή περισσότερα strings και μας επιστρέφουν το LCP τους. Ειδικότερα θα αναφερθούμε στους εξής αλγόριθμους:

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

Στη συνέχεια ασχολούμαστε με το Longest Common String πρόβλημα, ένα βασικό πρόβλημα στη μελέτη των strings. Οι αλγόριθμοι τους οποίους θα μελετήσουμε δέχονται ως είσοδο δύο strings και μας επιστρέφουν το LCS τους.Συγκεκριμένα επικεντρωνόμαστε στους παρακάτω αλγορίθμους:

- Naive Search
- Dynamic
- Suffix Tree
- Suffix Array

Έπειτα ακολουθεί ένα ακόμα πρόβλημα με strings που βρίσκει πολλές εφαρμογές στην βιοπληροφορική, το Longest Common Subsequence πρόβλημα. Όπως και στο LCS έτσι και εδώ η είσοδος δέχεται δύο strings. Οι αλγόριθμοι που θα αναλυθούν είναι οι εξής:

- Naive approach
- Dynamic Programming
- Longest Increasing Subsequence

Το τελευταίο πρόβλημα με το οποίο θα ασχοληθούμε είναι το Longest Common Extension πρόβλημα, που εμφανίζεται σαν υποπρόβλημα σε διάφορα θεμελιώδη προβλήματα με strings. Συγκεκριμένα θα εστιάσουμε στους παρακάτω αλγορίθμους:

- Naive Search
- Naive Search with k-Mismatches
- k-Closed Border

Οι δύο πρώτοι αλγόριθμοι θα χρησιμοποιηθούν για την επίλυση του προβλήματος εύρεσης k-Closed Border σε ένα string. Τέλος θα εκτελέσουμε πειράματα με βιολογικά δεδομένα για όλους τους αλγορίθμους που αναφέραμε παραπάνω και θα καταγράψουμε τους χρόνους εκτέλεσης τους, ώστε να επιβεβαιώσουμε ότι η απόδοση τους σε πρακτικές χρήσεις είναι ανάλογη της θεωρητική τους απόδοσης.

# 2. Apache Spark

## 2.1. Ιστορία του Apache Spark

Το Apache Spark είναι μια ενοποιημένη ισχυρή μηχανή ανάλυσης για επεξεργασία δεδομένων μεγάλης κλίμακας και είναι σχεδιασμένη έτσι ώστε να προσφέρει μεγάλη ταχύτητα. Το Apache Spark επιτυγχάνει υψηλή απόδοση τόσο για πακέτα δεδομένων όσο και για δεδομένα συνεχούς ροής, χρησιμοποιώντας έναν προγραμματιστή DAG τελευταίας τεχνολογίας, έναν βελτιστοποιητή ερωτημάτων και έναν μηχανισμό φυσικής εκτέλεσης. Προσφέρει διάφορες υπολογιστικές λειτουργίες όπως διαλογικές αναζητήσεις, επεξεργασία ροής, επαναληπτικούς αλγορίθμους και πολλά άλλα που οδηγούν σε μείωση της επιβάρυνσης της διαχείρισης ξεχωριστών εργαλείων. Στο cluster μνήμης η υπολογιστική δύναμη που προσφέρεται από το Spark ενισχύει την ταχύτητα επεξεργασίας.


**Εικόνα 2.1**: Το λογότυπο του Spark

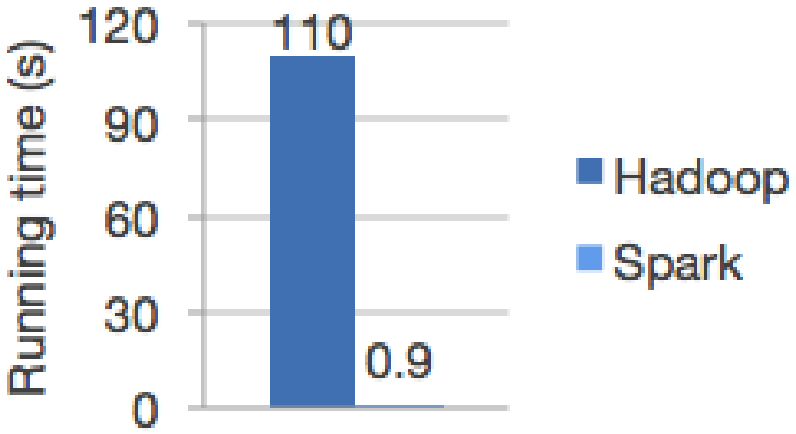
Μέσα στο Spark cluster υπάρχει ένα πρόγραμμα οδήγησης όπου αρχίζει η εκτέλεση λογικής εφαρμογής, με πολλές διεργασίες <<εργάτες>> που επεξεργάζονται δεδομένα παράλληλα. Κατά τη διάρκεια της εκτέλεσης, το πρόγραμμα οδήγησης περνάει κώδικα στο μηχάνημα του εργάτη όπου θα διεξαχθεί η επεξεργασία της αντίστοιχης κατάτμησης δεδομένων. Τα δεδομένα θα υποβληθούν σε διαφορετικά βήματα μετασχηματισμού ενώ παραμένουν στο ίδιο διαμέρισμα όσο το δυνατόν περισσότερο (για να αποφευχθεί η ανάμειξη δεδομένων μεταξύ μηχανών). Στο τέλος της εκτέλεσης, οι ενέργειες θα εκτελεστούν στον εργάτη και το αποτέλεσμα θα επιστραφεί στο πρόγραμμα οδήγησης. Πίσω από το cluster υπάρχει μια σημαντική δομή που ονομάζεται RDD (Resilient Distributed Dataset). Το Spark είχε ως θεμέλιο λίθο της αρχιτεκτονικής του το RDD, το οποίο είναι μια διαμοιρασμένη συλλογή αρχείων διαθέσιμη μόνο για ανάγνωση, ανεκτική σε σφάλματα. Δημιουργήθηκε για να ξεπεράσει περιορισμούς που έθεταν προηγούμενα πλαίσια υπολογιστικών συστάδων όπως το MapReduce ή το Dryad, που είχαν ευρέως υιοθετηθεί για ανάλυση δεδομένων σε μεγάλη κλίμακα. Αρχικά αναπτύχθηκε στο Πανεπιστήμιο της Καλιφόρνια Berkeley στο AMPLab από τον Matei Zaharia το 2009. Το

2010 ανακυρήχθηκε ως λογισμικό ανοιχτού κώδικα και από τον Φεβρουάριο του 2014 υιοθετήθηκε από την Apache Software, καθιστώντας το ένα κορυφαίου επιπέδου project της Apache.

## 2.2. Πλεονεκτήματα

Η δημοφιλία και η μεγαλύτερη προτίμηση που έχει το Spark σε σχέση με το Hadoop οφείλεται σε διάφορα χαρακτηριστικά του:

**Ταχύτητα** – Η υψηλή του ταχύτητα είναι ο κύριος λόγος της δημοφιλίας του και είναι σχεδιασμένο από την βάση προς τα πάνω με στόχο την αποδοτικότητα. Προσφέρει έως και 100 φορές γρηγορότερη επεξεργασία σε σχέση με το Hadoop αξιοποιώντας υπολογιστικές μνήμες και άλλες βελτιστοποιήσεις. Η υψηλή του ταχύτητα επιτυγχάνεται μέσω ελεγχόμενου διαμερισμού και είναι ωφέλιμη για την επεξεργασία δεδομένων μεγάλης κλίμακας. Τα δεδομένα χειρίζονται με χρήση διαμερισμού μέσω του οποίου μπορεί να εκτελεστεί παράλληλη κατανεμημένη επεξεργασία ακόμα και σε ελάχιστη κίνηση. Επιπλέον χρησιμοποιεί μόνο ένα μικρό αριθμό πόρων οπότε είναι αποδοτικό και από άποψη κόστους. Το Spark είναι επίσης γρήγορο όταν τα δεδομένα αποθηκεύονται στον δίσκο και μέχρι στιγμής κατέχει το παγκόσμιο ρεκόρ για ταξινόμηση σε μεγάλη κλίμακα σε δίσκο.



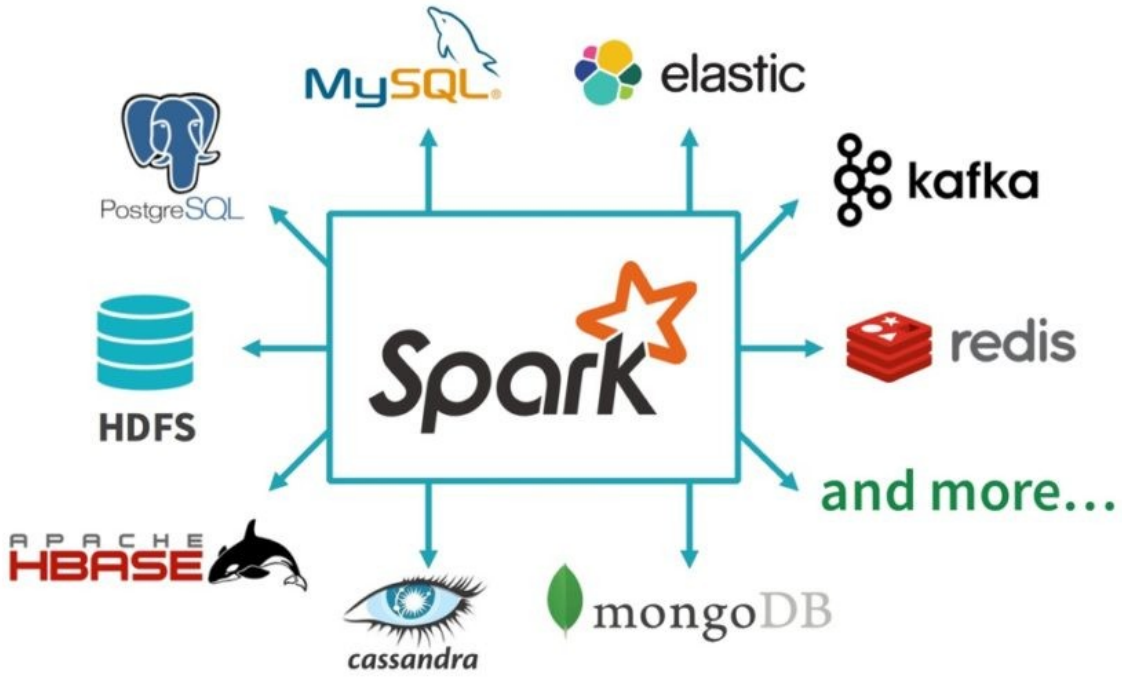

**Εικόνα 2.2**: Σύγκριση χρόνου εκτέλεσης του αλγορίθμου λογιστικής παλινδρόμησης σε Hadoop και Spark

**Ευκολία στη χρήση** – Το Spark παρέχει ευκολία στη χρήση APIs για λειτουργίες με μεγάλα σύνολα δεδομένων. Προσφέρει πάνω από 100 operators υψηλού επιπέδου που διευκολύνουν την κατασκευή παράλληλων εφαρμογών, για μετατροπή δεδομένων και για χειρισμό ημιδομημένων δεδομένων. Επίσης μπορεί να χρησιμοποιηθεί από τα shells των Scala, Python, R και SQL για γρήγορη δημιουργία εφαρμογών σε αυτές τις γλώσσες.

**Συμβατότητα** – Το Spark είναι συμβατό με τον διαχειριστή πόρων και τρέχει με το Hadoop όπως και το MapReduce. Άλλοι διαχειριστές πόρων συμβατοί με το Spark είναι οι YARN, EC2, Mesos, Kubernetes ενώ μπορεί να τρέχει και αυτόνομα στο cloud. Επίσης υποστηρίζει διάφορες πηγές δεδομένων όπως JSON, Cassandra, HDFS, Apache HBase και Hive, RDBMs πίνακες ή σε CSV μορφή. Επιπλέον παρέχεται από το Data Source API του Spark SQL ακόμα και συνδεόμενος μηχανισμός πρόσβασης στα δομημένα δεδομένα, ενώ διάφορες πηγές δεδομένων μπορεί να είναι μέρος της βάσης δεδομένων του Spark.

**Επεξεργασία σε πραγματικό χρόνο** – Άλλος ένας λόγος για την δημοφιλία του Spark είναι η επεξεργασία σε πραγματικό χρόνο πακέτων δεδομένων και η λειτουργία επεξεργασίας εντός μνήμης. Η διαδικασία υπολογισμού του Spark σε πραγματικό χρόνο έχει χαμηλή καθυστέρηση μεταφοράς στη φύση εξαιτίας της εκτέλεσης του υπολογισμού εντός μνήμης. Το Spark είναι σχεδιασμένο για μαζική κλιμάκωση και μπορεί να υποστηρίξει τους χρήστες που έχουν ομάδες παραγωγής με χιλιάδες κόμβους και διάφορα υπολογιστικά μοντέλα.

**Αργή αξιολόγηση –** Το Apache Spark συνήθως καθυστερεί την αξιολόγηση και γίνεται μόνο όταν αυτό είναι απαραίτητο. Εξαιτίας αυτού του λόγου αυξάνεται πολύ η ταχύτητά του. Έχει προστεθεί σε ένα DAG ή Direct Acyclic Graph για μετασχηματισμό, το οποίο εκτελείται όποτε απαιτούνται ορισμένα δεδομένα από τους οδηγούς.



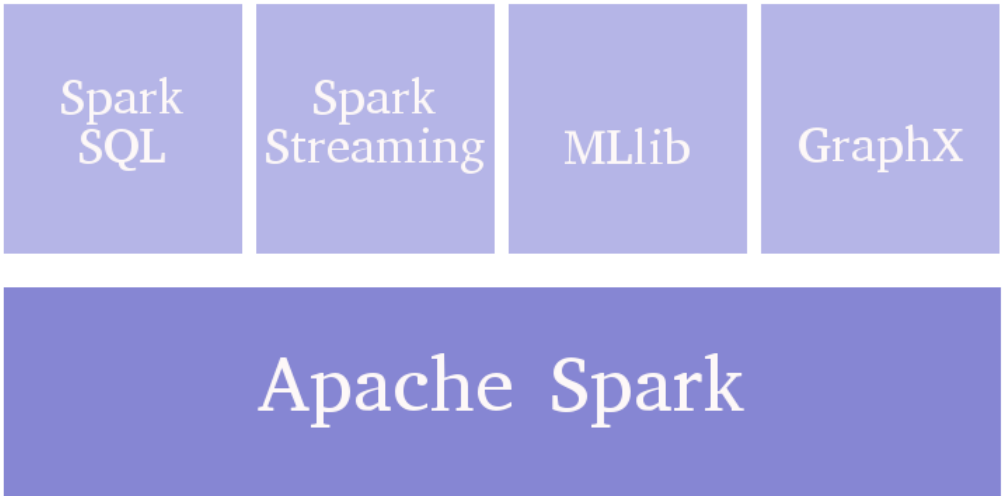

**Εικόνα 2.3:** Το Apache Spark είναι συμβατό με διάφορες πλατφόρμες και πηγές δεδομένων

**Ενσωμάτωση του Hadoop** – Στο Apache Spark είναι διαθέσιμη ομαλή συμβατότητα με το Hadoop. Οι χρήστες που έχουν ξεκινήσει να εργάζονται με Hadoop μπορούν να βρουν σημαντική βοήθεια στο Spark. Πρόκειται για ένα πιθανό αντικαταστάτη του MapReduce και μπορεί να τρέξει στο cluster του Hadoop για προγραμματισμό πόρων με χρήση του YARN.

**Machine Learning** – Η MLlib του Spark είναι ένα machine learning στοιχείο του Spark αρκετά χρήσιμο για την επεξεργασία δεδομένων. Γι' αυτό το σκοπό, αυτό το στοιχείο του Spark χρησιμοποιεί διάφορα εργαλεία και παρέχει ισχυρή και ενοποιημένη machine learning μηχανή για data engineers και data scientists.

## 2.3. Ενσωματωμένες βιβλιοθήκες

Το Spark περιλαμβάνει ένα σύνολο από βιβλιοθήκες συμπεριλαμβανομένων των SQL και DataFrames, MLlib, GraphX και Spark Streaming που αυξάνουν την αποδοτικότητα του developer ενώ μπορούν να συνδυαστούν χωρίς προβλήματα στην ίδια εφαρμογή.




**Εικόνα 2.4**: Οι ενσωματωμένες βιβλιοθήκες του Spark

Οι δυνατότητες του Spark για γρήγορο υπολογισμό και εύκολη ανάπτυξη επιτυγχάνονται χάρη στα παρακάτω στοιχεία του:

- **Apache Spark Core**: Ο πυρήνας του Spark στον οποίο βασίζονται όλες οι δυνατότητες του. Αποτελεί την βασική μηχανή εκτέλεσης και επεξεργασίας και μπορεί να αναφέρεται σε σύνολα δεδομένων του εξωτερικού αποθηκευτικού συστήματος ενώ παρέχει και λειτουργίες υπολογισμού μνήμης. Επίσης φιλοξενεί το

API από το οποίο ορίζεται το RDD που είναι η βασική μονάδα δεδομένων στο Spark και στο οποίο θα αναφερθούμε αναλυτικότερα παρακάτω.

- **Apache Spark SQL**: Το στοιχείο Spark Core προσφέρει μια νέα συντόμευση δεδομένων και αυτή η συντόμευση ονομάζεται Schema RDD. Το Apache SQL υποστηρίζει τόσο δομημένα όσο και μη δομημένα δεδομένα.

- **Apache Spark Streaming**: Το Spark Streaming δίνει τη δυνατότητα της επεξεργασίας σε πραγματικό χρόνο. Το στοιχείο αυτό του Spark πραγματοποιεί τις αναλύσεις ροής. Τα δεδομένα διαιρούνται σε μικρά πακέτα ώστε η επεξεργασία δεδομένων να γίνεται για πακέτα δεδομένων. Το Spark Streaming επίσης εκτελεί το Dstream, το οποίο αποτελείται από μια σειρά από RDDs ενώ πραγματοποιεί και την επεξεργασία σε πραγματικό χρόνο.

- **MLlib (Machine Learning Library)**: Το Machine Learning Framework του Spark γνωστό ως MLlib αποτελείται από εργαλεία machine learning και αλγορίθμους. Οι βιβλιοθήκες του περιλαμβάνουν λειτουργίες clustering, παλινδρόμησης, ταξινόμησης και πολλές άλλες. Η επεξεργασία δεδομένων εντός μνήμης λόγω της οποίας αυξάνεται η απόδοση των επαναληπτικών αλγορίθμων, βελτιώνεται επίσης.

- **GraphX**: Το GraphX είναι διανεμημένο πλαίσιο επεξεργασίας γραφημάτων, λειτουργεί στην κορυφή του Spark και επιτρέπει την ταχύτατη επεξεργασία δεδομένων σε μεγάλη κλίμακα.

- **SparkR**: Το SparkR είναι ένας συνδυασμός Spark και R. Μέσω του SparkR μπορούν να εξερευνηθούν διαφορετικές τεχνικές ενώ μέσω της συγχώνευσης των λειτουργιών της R και των χαρακτηριστικών επεκτασιμότητας του Spark, ενισχύονται οι λειτουργίες του Spark.

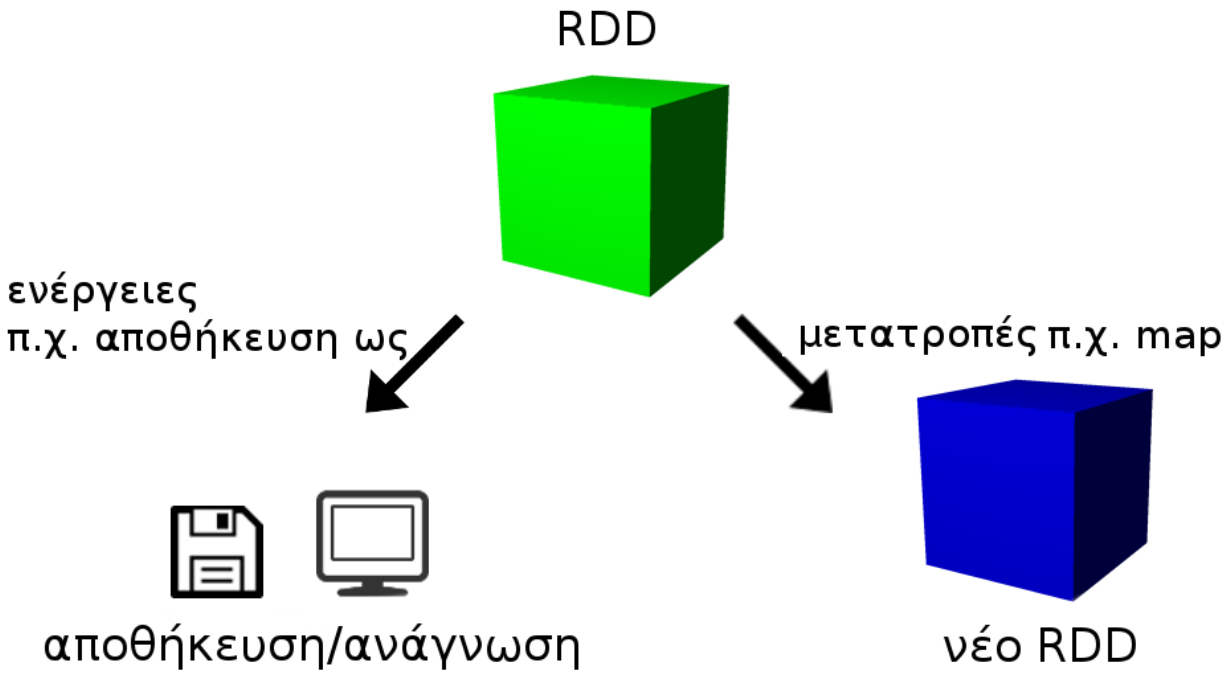
Ένα από τα πιο ελκυστικά χαρακτηριστικά του Spark για τους developers είναι τα εύκολα στην χρήση APIs που αυξάνουν την παραγωγικότητά τους, λειτουργώντας με μεγάλα σύνολα δεδομένων και σε διάφορες γλώσσες προγραμματισμού.

Από την έκδοση του Spark 1.3, το Spark και τα APIs του εξελίσσονται ώστε να είναι ευκολότερα, γρηγορότερα και εξυπνότερα. Ο στόχος είναι στις μελλοντικές εκδόσεις του Spark να ενοποιηθούν αυτές οι έννοιες ώστε οι big data developers να έχουν λιγότερες έννοιες για να αντιμετωπίσουν. Η μηχανή Spark SQL , με τον Catalyst βελτιστοποιητή και την παραγωγή κώδικα σε όλα τα στάδια, βοηθά τους developers στην κατανόηση της χρήσης των APIs και στην χρήση των καλύτερων πρακτικών για να δημιουργήσουν τις εφαρμογές τους στο Spark. Παρακάτω περιγράφονται  τρία σημαντικά APIs τα RDDs, DataFrames και Datasets.

## 2.4. RDD, Data Frames και Datasets

### Resilient Distributed Dataset (RDD)

Το RDD ή Resilient Distributed Dataset είναι βασική μονάδα δεδομένων στο Spark. Ήταν το κύριο API με το οποίο ερχόταν σε επαφή ο χρήστης στο Spark από την έναρξη λειτουργίας του. Τα RDDs είναι μία αμετάβλητη συλλογή από data sets τα οποία κατανέμονται σε διάφορους cluster κόμβους. Υποστηρίζουν παράλληλες λειτουργίες και είναι αμετάβλητα από την φύση τους. Μπορούν να δημιουργηθούν στο Spark έιτε μέσω εξωτερικών datasets είτε από παράλληλες συλλογές ή από ήδη υπάρχοντα RDDs.




**Εικόνα 2.5:** Λειτουργίες του RDD

Τα RDDs παρότι δεν μπορούν να τροποποιηθούν μπορούν να λειτουργούν παράλληλα με API χαμηλού επιπέδου που προσφέρει τις δυνατότητες μετασχηματισμού τους όπως π.χ. να μετατραπούν σε νέα RDDs και δράσεων που επιστρέφουν μια τιμή στο πρόγραμμα οδήγησης μετά από την εκτέλεση ενός υπολογισμού. Κάποιες συνήθεις περιπτώσεις χρήσης τους είναι:

- όταν θέλουμε χαμηλού επιπέδου μετασχηματισμό, ενέργειες και έλεγχο στο σύνολο των δεδομένων μας

- όταν τα δεδομένα μας είναι μη δομημένα όπως π.χ. streams μέσων ή streams κειμένου

- όταν θέλουμε να χειριστούμε τα δεδομένα μας με λειτουργικές δομές προγραμματισμού

- όταν δεν μας ενδιαφέρει η επιβολή ενός σχήματος κατά την επεξεργασία όπως π.χ. σε μορφή στήλης ή να έχουμε πρόσβαση σε χαρακτηριστικά των δεδομένων με βάση τη στήλη ή το όνομα

- τέλος όταν μπορούμε να προσπεράσουμε κάποια πλεονεκτήματα βελτιστοποίησης που συναντάμε στα DataFrames και στα Datasets για δομημένα και ημιδομημένα δεδομένα.

**Data Frames**

Τα Apache Spark Data Frames όπως και τα RDDs είναι αμετάβλητες συλλογές κατανεμημένων δεδομένων. Σε αντίθεση με τα RDDs στα Data Frames τα δεδομένα οργανώνονται σε στήλες, όπως ένας πίνακας σε μια σχεσιακή βάση δεδομένων και βελτιστοποιημένους πίνακες. Είναι σχεδιασμένα ώστε να διευκολύνουν την επεξεργασία μεγάλων συνόλων δεδομένων, επιτρέποντας στους developers να επιβάλλουν μια δομή σε μια κατανεμημένη δομή δεδομένων προσφέροντας υψηλού επιπέδου αφαιρετικότητα. Επίσης κάνει το Spark ευκολότερα προσβάσιμο σε κοινό εκτός των εξειδικευμένων μηχανικών δεδομένων. Τα Data Frames μπορούν να κατασκευαστούν από διάφορες πηγές δεδομένων όπως αρχεία δεδομένων, εξωτερικές βάσεις δεδομένων και υπάρχοντα RDDs. Τα χαρακτηριστικά τους είναι τα εξής:

- Σε ένα και μόνο cluster κόμβο μπορούν να επεξεργαστούν τεράστια ποσά δεδομένων

- Μέσω του Spark Core τα data frames μπορούν να ενσωματωθούν με τα Big Data εργαλεία

- Υποστηρίζουν Java, R, Scala και Python APIs

- Υποστηρίζουν διάφορες μορφές δεδομένων όπως Avro, CSV, HDFS και Hive πίνακες

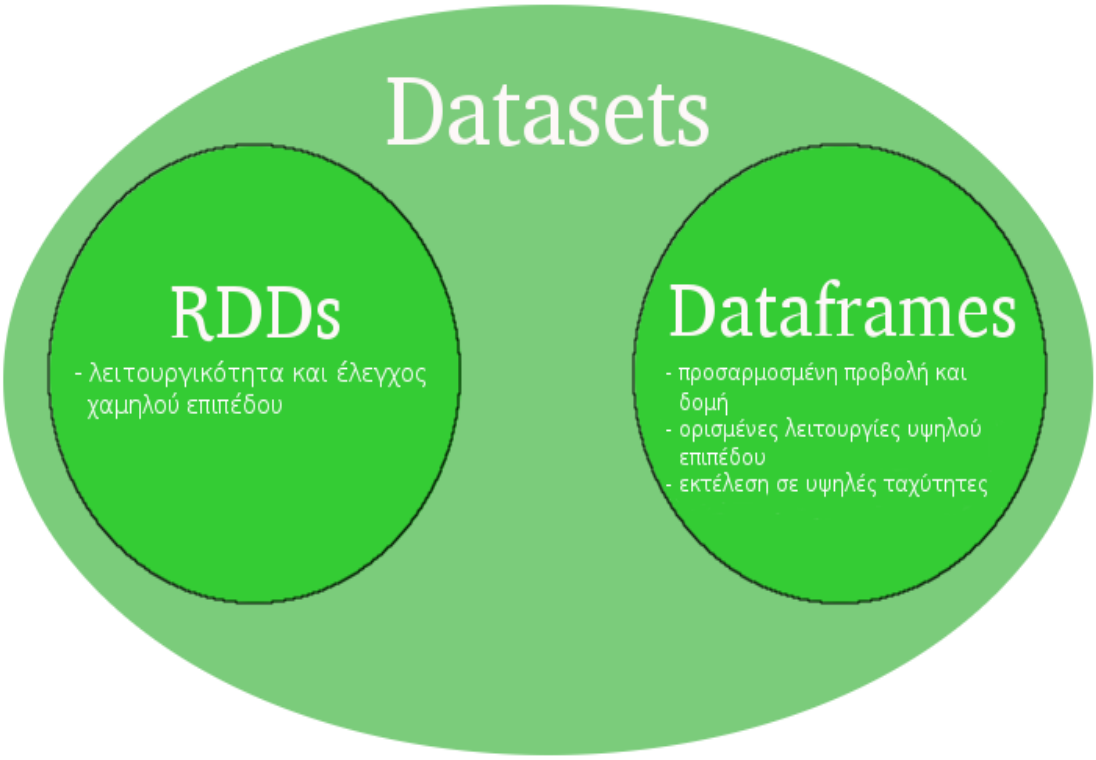
- Ο SQL καταλύτης μπορεί να βελτιστοποιήσει την απόδοση του κώδικα και να παράγει πιο ακριβείς εξόδους

**Datasets**

Ένα Dataset είναι μια συλλογή δεδομένων. Το Dataset είναι νεότερη διεπαφή του Spark σε σχέση με τις δύο προηγούμενες που προστέθηκε στο Spark 1.6 και προσφέρει τα πλεονεκτήματα του RDD με τα πλεονεκτήματα της βελτιστοποιημένης μηχανής εκτέλεσης Spark SQL. Ένα Dataset μπορεί να κατασκευαστεί από JVM αντικείμενα και μετά να το χειριστούμε όπως και τα RDDs χρησιμοποιώντας μετασχηματισμούς (map, filter, flatmap, select, aggregate) και δράσεις από τις οποίες ξεκινάνε υπολογισμοί και επιστρέφονται αποτελέσματα (count, show ή εγγραφή δεδομένων σε συστήματα αρχείων). Τα οφέλη των Datasets είναι ότι αντίθετα με τα RDDs αυτοί οι μετασχηματισμοί εφαρμόζονται σε μια δομημένη και ισχυρά τυποποιημένη κατανεμημένη συλλογή που επιτρέπει στο Spark να αξιοποιήσει τη μηχανή εκτέλεσης Spark SQL για βελτιστοποίηση.

Το Dataset API είναι διαθέσιμο σε Scala και Java αλλά δεν υποστηρίζεται από την Python. Εξαιτίας όμως της δυναμικής φύσης της Python πολλά οφέλη του Dataset API είναι διαθέσιμα. Παρόμοια περίπτωση είναι και η R. Στα Datasets οι υπολογισμοί ξεκινάνε μόνο όταν προκαλείται μία δράση. Εσωτερικά ένα Dataset αντιπροσωπεύει ένα λογικό σχέδιο που περιγράφει τον υπολογισμό που χρειάζεται για να παραχθούν τα δεδομένα. Όταν προκαλείται μια δράση, η βελτιστοποίηση ερωτήματος του Spark δημιουργεί ένα νέο σχέδιο βελτιστοποιώντας το λογικό σχέδιο ώστε να πραγματοποιηθεί αποτελεσματική εκτέλεση με παράλληλο και κατανεμημένο τρόπο.

Τυπικά υπάρχουν δύο τρόποι για την δημιουργία ενός Dataset. Ο πιο συνήθης τρόπος είναι υποδεικνύοντας στο Spark κάποια αρχεία σε συστήματα αποθήκευσης και χρησιμοποιώντας την λειτουργία ανάγνωσης που είναι διαθέσιμη σε ένα SparkSession. Ο άλλος τρόπος είναι η δημιουργία νέων Datasets μέσω διαθέσιμων μετασχηματισμών σε ήδη υπάρχοντα Datasets π.χ. με την εφαρμογή ενός φίλτρου σε ένα Dataset δημιουργείται ένα νέο Dataset. Οργανώνοντας τα Datasets σε ονομαστικές στήλες προκύπτουν DataFrames.
Συνοψίζοντας οι κύριες διαφορές ανάμεσα σε RDD, Data Frame και Dataset είναι η σειριοποίηση και η απόδοση. Ενώ το πρώτο προσφέρει λειτουργικότητα και έλεγχο σε χαμηλό επίπεδο τα άλλα επιτρέπουν προσαρμοσμένη προβολή και δομή, προσφέρουν λειτουργίες υψηλού επιπέδου και για συγκεκριμένους τομείς, εξοικονόμηση χώρου και εκτέλεση σε ανώτερες ταχύτητες.




**Εικόνα 2.6:** RDD vs DataFrame vs Dataset

Η επιλογή της χρήσης του RDD ή του DataFrame και/ή του Dataset εξαρτάται από ποια χαρακτηριστικά είναι επιθυμητά κάθε φορά.

## 2.5. Spark: Παρόν και Μέλλον

Από τα παραπάνω γίνεται σαφές ότι το Spark έχει κυριαρχήσει στον κόσμο των Big Data και θεωρείται ένα από τα δημοφιλέστερα εργαλεία για Big Data. Το πανίσχυρο αυτό πλαίσιο παρέχει την δυνατότητα ευκολίας χρήσης ενώ ενισχύει τις δυνατότητες του Big Data και την απόδοση των συστημάτων. Έχει γίνει ένα πολύ χρήσιμο εργαλείο για τους developers και η εξάπλωση της χρήσης του έχει αυξηθεί με εκπληκτική ταχύτητα. Χρησιμοποιείται πλέον από ένα ευρύ εύρος οργανισμών για την επεξεργασία μεγάλων συνόλων δεδομένων και κατασκευάζεται από ένα ευρύ σύνολο προγραμματιστών από περισσότερες από 300 εταιρείες. Από το 2009 περισσότεροι από 1200 προγραμματιστές έχουν συμβάλλει στην ανάπτυξη του Apache Spark. Έχει δει ταχεία υιοθέτηση από επιχειρήσεις σε ένα ευρύ φάσμα βιομηχανιών. Εταιρείες όπως το Netflix, Yahoo και eBay χρησιμοποιούν το Spark σε τεράστιο βαθμό, επεξεργάζοντας πολλαπλά petabytes

δεδομένων σε clusters άνω των 8000 κόμβων. Το Spark έχει γίνει πολύ γρήγορα η μεγαλύτερη κοινότητα ανοιχτού κώδικα στα Big Data.

Η Databricks, που κυριαρχεί στην ανάπτυξη και την μελλοντική κατεύθυνση του Spark, ανακοίνωσε την ανάπτυξη του project Delta. Πρόκειται για ένα πανίσχυρο αποθηκευτικό επίπεδο που αξιοποιεί την δύναμη του Spark επιτρέποντας στους χρήστες να κάνουν όλη την επεξεργασία δεδομένων σε κλίμακα και διακίνηση των δεδομένων μέσα στο cloud. Σε αυτή την πλατφόρμα θα συνδυάζονται επεξεργασία συνεχούς ροής και πακέτων δεδομένων, αποθήκες δεδομένων και machine learning, ενώ θα τρέχει στο cloud προσφέροντας επεξεργασία σε κλίμακα και ελαστικότητα.

**Εικόνα 2.7**: Το μέλλον του Spark βρίσκεται στο cloud

Παρότι το Spark στο cloud δεν είναι κάτι καινούριο με την προσθήκη νέων εργαλείων προστίθενται οι δυνατότητες των συναλλαγών και μεταδεδομένων. Τα μεταδεδομένα περιλαμβάνουν συμπίεση δεδομένων, αντιστοίχιση σε σχήμα, βελτιστοποίηση στατιστικών ερωτημάτων και ανάπτυξη χωρίς την ανάγκη ύπαρξης διακομιστών. Η εποχή του cloud για το Spark θα το ωθήσει στο μέλλον σε πολλές βελτιώσεις.

Επίσης οι δύο πιο γρήγορα αναπτυσσόμενες περιοχές του Spark είναι το streaming και το Deep Learning (DL) και ο στόχος είναι να δουλεύουν μαζί. Προς το παρόν είναι ακόμα μια περίπλοκη διαδικασία καθώς τα εργαλεία είναι νέα και δεν είναι τόσο καλά ενσωματωμένα.

Η ίδια κατάσταση είχε αντιμετωπιστεί παλιότερα και με το Hadoop. Σταδιακά γίνεται προσθήκη DL στη Machine Learning pipeline όπου αποδεικνύεται ότι αν προσθέσεις DL operators εκεί, πολλές κοινές περιπτώσεις χρήσης είναι εύκολο να εφαρμοστούν. Στόχος αποτελεί η διευκόλυνση των χρηστών που θέλουν να χρησιμοποιούν υπάρχοντα μοντέλα και να τα <<εκπαιδεύουν>> στα δεδομένα τους για εφαρμογές σε προβλήματα. Οι χρήστες αυτοί θέλουν να δημιουργούν γρήγορες εφαρμογές και αυτό που έχει σημασία είναι να μπορούν να συνθέτουν εφαρμογές από μικρότερα μέρη γρήγορα και αποδοτικά. Σε αυτό το σημείο το Spark θέλει να προσφέρει μια standard βιβλιοθήκη αλγορίθμων τους οποίους οι χρήστες μπορούν να συνδυάσουν και να ταιριάξουν όπως θέλουν και να έχουν καλή απόδοση. Σε αυτή την προσπάθεια για DL δυνατότητες η εμπειρία με το Hadoop προσφέρει σημαντική βοήθεια. Δεδομένου ότι το Hadoop είναι η ραχοκοκκαλιά τόσων πολλών συνόλων δεδομένων είναι λογικό να χρησιμοποιούνται δεδομένα αποθηκευμένα εκεί για DL. Γενικότερα το Spark κατευθύνεται σε πιο ευέλικτη μορφή όπου θα μπορούν να συνδυάζονται ταυτόχρονα αλγόριθμοι και διαφορετικές δυνατότητες προσφέροντας καλή απόδοση.

# 3. Longest Common Prefix (LCP)

## 3.1. Υπάρχουσα μελέτη και εφαρμογές

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 strings είναι το μεγαλύτερο string το οποίο είναι κοινό prefix για όλα τα strings. Εισήχθη το 1993 από τους Udi Manber και Gene Myers ώστε σε συνδυασμό με το suffix array να βελτιώσουν τον string search αλγόριθμο 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 αλγόριθμο τους μπορούσαν να υπολογίσουν το LCP. Ο Fischer το 2011 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 εκτός από την παραλληλοποίηση της brute force μεθόδου υπάρχουν δύο μέθοδοι για τον υπολογισμό του LCP. Στην μία μέθοδο χρησιμοποιείται ο αλγόριθμος των Kärkkäinen και Sanders που αναφέρθηκε παραπάνω, ο οποίος δεν υπολογίζει αποκλειστικά το LCP αλλά υπολογίζει και το suffix array. Η δεύτερη μέθοδος χρησιμοποιεί ένα παράλληλο αλγόριθμο για GPU’s που παρουσίασαν οι Deo και Kelly και βασίζεται στην παραλληλοποίηση του αλγορίθμου του Kasai. Ο αλγόριθμος αυτός υπολογίζει το LCP χωρίς να χρειάζεται να υπολογίσει πρώτα κάποια άλλη δομή. Επίσης υπάρχουν αλγόριθμοι που υπολογίζουν το LCP κατασκευάζοντας αρχικά το suffix tree και στην συνέχεια ερευνώντας το βάθος του κάθε εσωτερικού κόμβου στο δέντρο. Οι αλγόριθμοι αυτοί είναι λιγότεροι αποδοτικοί από την εύρεση του LCP από το suffix array και μάλιστα ο γρηγορότερος παράλληλος suffix tree αλγόριθμος στην πράξη χρειάζεται πρώτα την κατασκευή των suffix array και LCP array.

Το LCP σε συνδυασμό με το suffix array προτιμώνται σε σχέση με το suffix tree για ευρετήριο κειμένου λόγω των χαμηλών τους 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. Επιπρόσθετα οι παραπάνω 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 ενισχύοντας 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 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:

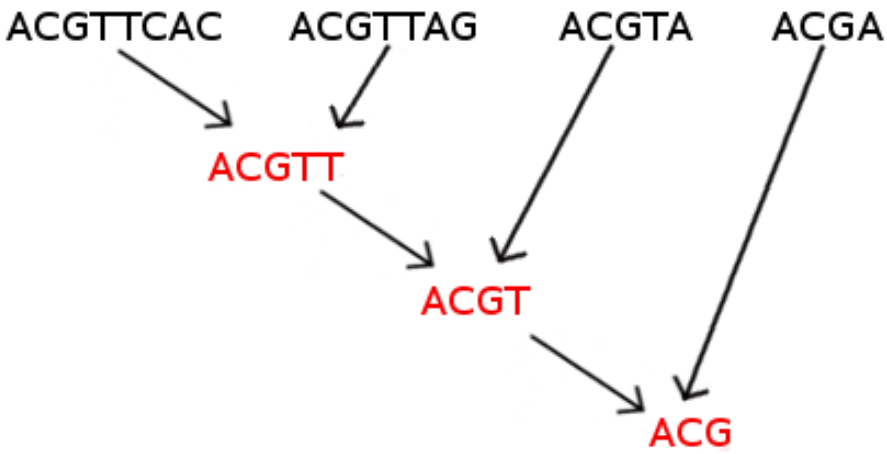
- Word by Word Matching
- Character by Character Matching
- Divide and Conquer
- Binary Search

## 3.2. 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:

**$LCP(s_1, s_2, s_3)=LCP(LCP(s_1,s_2), s_3)$**




**Σχήμα 3.1:** 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 strings $[s_1...s_i]$. 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 strings και m το μήκος του μεγαλύτερου string η χρονική πολυπλοκότητα του αλγορίθμου θα είναι O(n*m), καθώς στην χειρότερη περίπτωση διατρέχουμε όλα τα strings και όλους τους χαρακτήρες για το καθένα.

## 3.3. Character by Character Matching

Έστω ότι το μικρότερο string από αυτά που θέλουμε να εξετάσουμε βρίσκεται στο τέλος του συνόλου των strings. Η προηγούμενη προσέγγιση θα εξετάσει όλα τα strings βρίσκοντας πιθανώς πολλές φορές LCPs ανάμεσα στα strings με μέγεθος μεγαλύτερο του τελευταίου, που είναι το μικρότερο, 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” και “ACGA” θα ξεκινήσουμε από την πρώτη στήλη του πίνακα με όλα τα strings, δηλαδή θα ελέγξουμε τον χαρακτήρα στην πρώτη θέση όλων των strings οπότε θα έχουμε:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 3.1:** Πρώτος έλεγχος του αλγορίθμου

Όλοι οι χαρακτήρες στο πρώτο πέρασμα είναι “A” άρα το “A” μπαίνει στο prefix που θέλουμε. Συνεχίζουμε στην επόμενη θέση εκτελώντας τον ίδιο έλεγχο σε όλα τα strings. Ο αλγόριθμος θα σταματήσει αν βρεθεί κάποιο string ή περισσότερα όπου στην ίδια θέση με τα άλλα ο χαρακτήρας είναι διαφορετικός.

‘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 3.2:** Εκτέλεση δεύτερου ελέγχου από τον αλγόριθμο

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 3.3:** Επανάληψη διαδικασίας

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 3.4:** Εύρεση μη κοινού χαρακτήρα-τερματισμός αλγορίθμου

Αυτή την φορά παρατηρούμε ότι ο χαρακτήρας δεν είναι ο ίδιος και για τις τέσσερις 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” και ο αλγόριθμος τερματίζεται.

**Πολυπλοκότητα χρόνου:** Η πολυπλοκότητα θα είναι και εδώ O(n*m) αφού στην χειρότερη περίπτωση διατρέχουμε όλα τα strings 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 strings 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 strings, θα ξέρουμε ότι δεν υπάρχει κοινό prefix και δεν θα χρειαστεί να συνεχίσουμε την αναζήτηση. Αυτό είναι ένα μεγάλο πλεονέκτημα όταν τα strings είναι πολλά και κάνει πολύ πιο εύκολη την εύρεση του prefix.

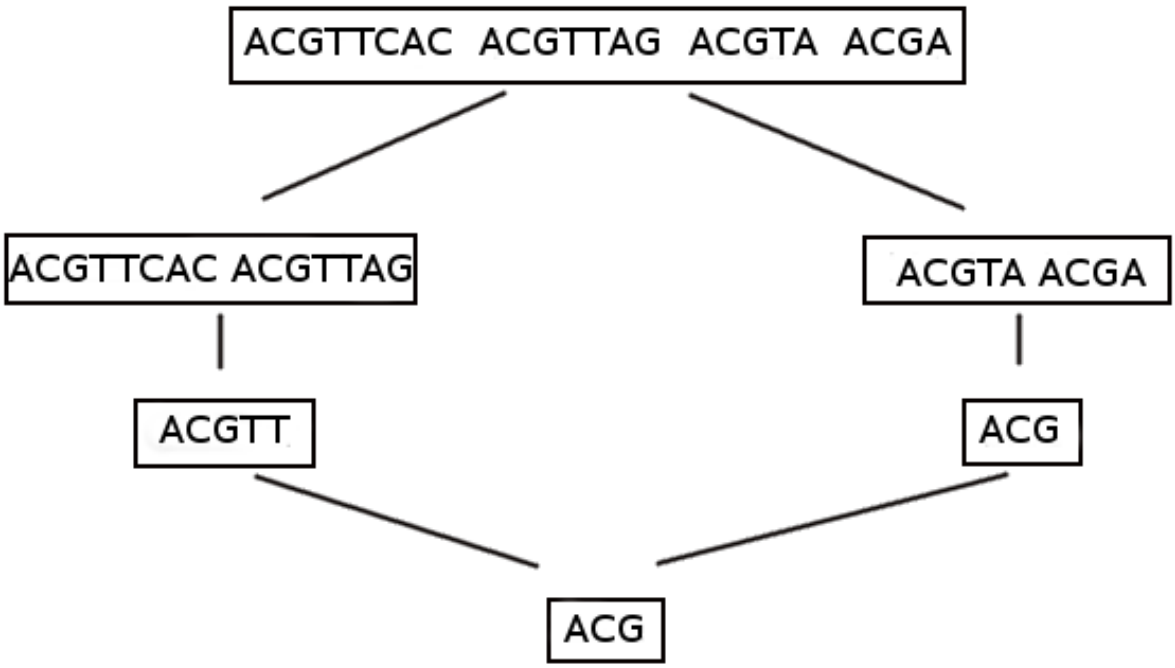
## 3.4. Divide and conquer

Μια διαφορετική προσέγγιση είναι η χρήση του αλγορίθμου Divide and Conquer για τον υπολογισμό του LCP. 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 στο επόμενο κεφάλαιο. 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 $LCP(s_i...s_j)$ θα προκύψουν δύο υποπροβλήματα $LCP(s_i...s_m)$ και $LCP(s_m...s_j)$, 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 τελική λύση $LCP(s_i...s_j)$, αρκεί να υπολογιστεί το κοινό prefix των δύο μερών.

Χρησιμοποιώντας τα ίδια strings με πριν, δηλαδή τις ακολουθίες: “ACGTTCAC”, “ACGTTAG”, “ACGTA”, “ACGA” η σχηματική απεικόνιση της εκτέλεσης των βημάτων του αλγορίθμου που περιγράφηκε παραπάνω θα είναι:



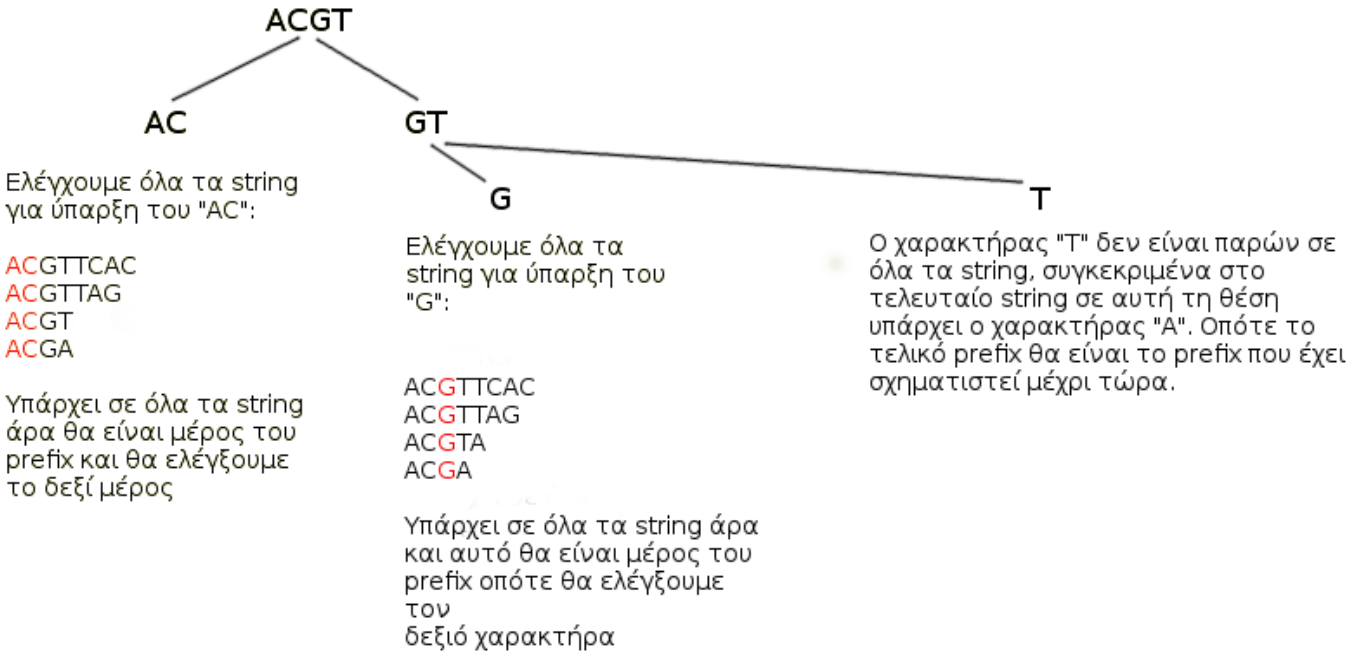

**Σχήμα 3.2:** Σχηματική απεικόνιση των βημάτων που ακολουθούνται κατά την εκτέλεση του αλγορίθμου Divide and Conquer

**Πολυπλοκότητα χρόνου**: Και εδώ εξετάζουμε όλα τα strings και όλους τους χαρακτήρες τους στη χειρότερη περίπτωση οπότε η πολυπλοκότητα θα είναι O(n*m).

## 3.5. 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 ίδια 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≤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.




**Σχήμα 3.3:** Στο παραπάνω σχήμα απεικονίζονται τα βήματα που ακολουθούνται κατά την εκτέλεση του αλγορίθμου 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.

**Πολυπλοκότητα χρόνου:** Προκύπτει από την επαναληπτική σχέση T(m)=T(m/2)+O(m*n) όπου m το μήκος του μεγαλύτερου string και n ο αριθμός των strings, οπότε μπορούμε να πούμε ότι η χρονική πολυπλοκότητα είναι O(n*m*logm) στη χειρότερη περίπτωση. 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 δεν θα χρειαστεί να κάνει αναζήτηση στο άλλο μισό.

# 4. Longest Common Substring (LCS)

## 4.1. Υπάρχουσα μελέτη και εφαρμογές

Το Longest Common Substring-LCS πρόβλημα είναι το πρόβλημα εξαγωγής μέγιστης κοινής υποσυμβολοσειράς και πρόκειται για το κλασικό πρόβλημα εύρεσης της μέγιστης υποσυμβολοσειράς η οποία είναι κοινή για όλα τα strings 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 strings, η οποία δεν απαιτείται να καταλαμβάνει συνεχόμενες θέσεις στα αρχικά strings. Η κατανόηση του πόσο όμοιες είναι δύο συμβολοσειρές και τι έχουν κοινό είναι ένα βασικό πρόβλημα στην μελέτη των strings.

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 γραμμικού 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 strings x και y είναι το LCP μερικών suffixes του x και μερικών suffixes του y. Η δεύτερη παρατήρηση είναι ότι το μέγιστο μήκος του LCP μεταξύ ενός suffix του x και suffixes του y προσεγγίζεται στα δύο suffixes του y τα οποία είναι πιο κοντά στο x σύμφωνα με την λεξικογραφική σειρά.

Εναλλακτικά μπορούμε να χτίσουμε μια δομή δεδομένων που θα υπολογίζει το το μέγιστο κοινό prefix για οποιοδήποτε ζευγάρι suffixes. Η δημιουργία μιας τέτοιας δομής είναι γνωστή σαν Longest Common Extension (LCE) πρόβλημα, με διάφορες γνωστές λύσεις, στο οποίο θα αναφερθούμε αναλυτικά σε επόμενο κεφάλαιο. Ένα τέτοιο παράδειγμα επίλυσης παρουσιάζεται από τον Bille με χρήση μιας ντετερμινιστικής δομής δεδομένων. Επίσης χρησιμοποιώντας την μέθοδο που περιέγραψε ο Rabin-Karp, όπου κάθε string x συσχετίζεται με ένα <<δαχτυλικό αποτύπωμα>>, μπορούμε να πάρουμε διάφορους τυχαίους αλγορίθμους υπογραμμικού χώρου. Η μέθοδος αυτή βασίζεται στην παρακάτω παρατήρηση: έστω ότι έχουμε έναν αλγόριθμο που εντοπίζει ένα substring σε κάποια από τα strings που του δίνουμε και επιστρέφει αυτό το substring. Τότε μπορούμε να βρούμε το LCS επαναλαμβάνοντας τον αλγόριθμο O(log|LCS|) φορές, αναζητώντας το substring με το μέγιστο μήκος.

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 αναλλοίωτα, 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 στοίχιση strings, το πρόβλημα δηλαδή της πολλαπλής στοίχισης 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 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. Επιπλέον το LCS έχει χρησιμοποιηθεί σε αλγορίθμους γρήγορης επιλογής ολιγονουκλεοτιδίων σε μεγάλη κλίμακα που έχουν προταθεί, όπου επιλέγονται ανιχνευτές ολιγονουκλεοτιδίων για πειράματα μικροσυστοιχιών σε πραγματικά μεγάλη κλίμακα. Η ταχύτητα αυτών των αλγορίθμων επιτυγχάνεται με χρήση του LCS ως μέτρο ειδικότητας για τα υποψήφια ολιγοειδή. Επίσης το LCS και παραλλαγές του μπορούν να χρησιμοποιηθούν για να προσεγγιστεί το πρόβλημα των φυλογενετικών αποστάσεων μεταξύ ακολουθιών DNA ή πρωτεϊνών με στόχο την φυλογενετική ανακατασκευή.

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.

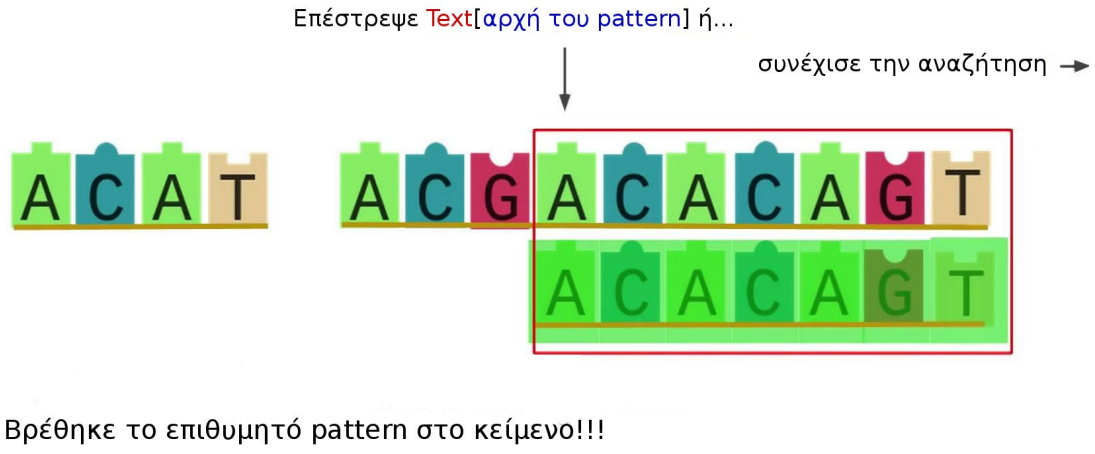
Παρακάτω θα εξεταστούν διάφοροι αλγόριθμοι επίλυσης του προβλήματος και η απόδοσή τους. Οι αλγόριθμοι αυτοί εκτελούν αναζήτηση για LCS ανάμεσα σε δύο strings, ως είσοδος δόθηκαν βιολογικά δεδομένα και είναι οι εξής:

- Naive Search
- Dynamic
- Suffix Tree
- Suffix Array

## 4.2. 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 κ.ο.κ. ώστε από τα κοινά substrings να επιλεχθεί το μεγαλύτερο. Για παράδειγμα το LCS των strings “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 μέγεθος 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 strings




**Εικόνα 4.1:** Η λειτουργία του αλγορίθμου KMP

**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 $O(n*m^2)$.

## 4.3. 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 προβλήματα.

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 strings 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 |

**Πίνακας 4.1:** 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 strings της εισόδου.

## 4.4. Suffix Array

Ο ορισμός του suffix Array είναι παρόμοιος 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 strings που δίνονται στην είσοδο σε ένα νέο string ως εξής:

**$s=x\$_1y\$_2$**

όπου x, y τα δύο strings εισόδου, s το νέο string που προκύπτει και $\$_1$, $\$_2$ δύο διαφορετικά σύμβολα που δεν εντοπίζονται μέσα στα προηγούμενα strings. 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 LCPs, το οποίο πρέπει να είναι μέρος και των δύο strings, θα είναι το ζητούμενο LCS των δύο strings 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 4.1:** 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:** Αν τα strings που δίνονται στην είσοδο έχουν μήκος n και m, τότε για τον υπολογισμό του LCS τους με χρήση της Suffix Array μεθόδου χρειάζεται χρόνος O(n+m) στην χειρότερη περίπτωση.

## 4.5. 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 στοίχιση.

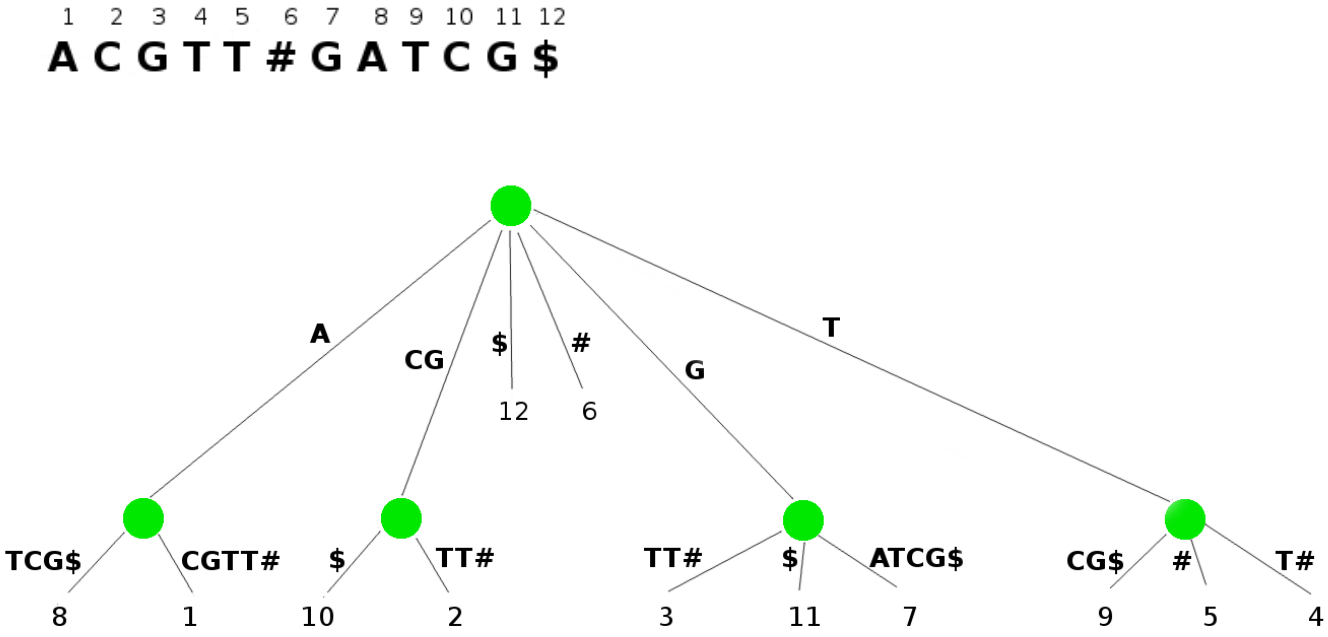
Τα βήματα που ακολουθεί ο αλγόριθμος είναι:

1. κατασκευή του Generalized Suffix Tree:
   - συνένωση των αρχικών strings
   - κατασκευή του suffix tree
2. ονομασία κάθε κόμβου ανάλογα σε ποια strings ανήκουν τα φύλλα μέχρι εκεί
3. διέλευση του δέντρου από την ρίζα προς τους κόμβους που έχουν σημειωθεί τα ονόματα και των δύο αρχικών strings, το μονοπάτι προς τον βαθύτερο κόμβο περιέχει το LCS

Ο αλγόριθμος ξεκινάει φτιάχνοντας το Generalized Suffix Tree των strings που δίνονται στην είσοδο με χρήση του αλγορίθμου McCreigth. Το Generalized Suffix Tree κατασκευάζεται 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 των x και y, το μονοπάτι από τη ρίζα προς τον <<βαθύτερο>> κόμβο θα μας δώσει το επιθυμητό LCS.

Έστω για παράδειγμα ότι θέλουμε να βρούμε το LCS των strings x="ACGTT" και y="GATCG" με την μέθοδο του suffix tree. Έστω ότι το νέο string που προκύπτει από αυτά τα δύο είναι το s="ACGTT#GATCG$". Το Generalized Suffix Tree για το string s θα είναι το παρακάτω:



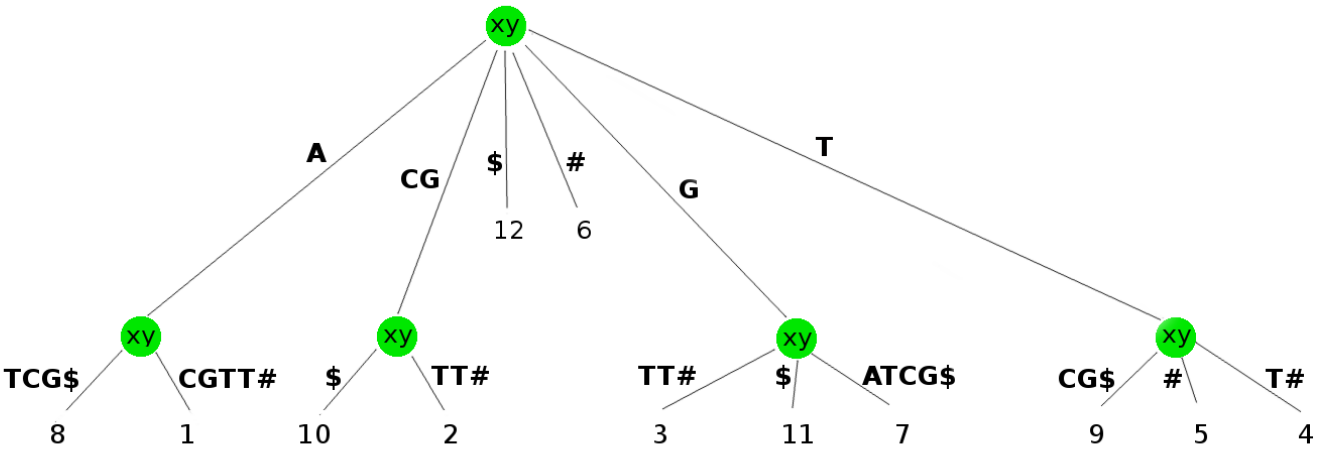

**Σχήμα 4.2:** Το Generalized Suffix Tree της συνένωσης των δύο αρχικών strings

Στο παραπάνω δέντρο θα αναζητήσουμε αν τα φύλλα με suffixes είναι σε μια από τις παρακάτω περιπτώσεις και ανάλογα θα δοθεί ένα όνομα στον κόμβο που τα ακολουθεί:

- Αν το suffix είναι κοινό και στα δύο strings τότε ο κόμβος σημειώνεται ως xy
- Αν το suffix υπάρχει μόνο στο x τότε ο κόμβος σημειώνεται ως x
- Αν το suffix υπάρχει μόνο στο y τότε ο κόμβος σημειώνεται ως y

Στο σχήμα αυτό τα φύλλα με δείκτες των suffixes [1, 5] είναι suffixes του string x και τα φύλλα με δείκτες των suffixes [7, 11] είναι suffixes του string y. Αυτό συμβάινει γιατί όταν συνενώθηκαν τα δύο strings δημιουργώντας το string s, η αρχή του x είναι το 1 και το

μήκος του 5 οπότε οι δείκτες των suffixes του θα είναι οι 1, 2, 3, 4, 5. Αντίστοιχα η αρχή του string y είναι το 7 και το μήκος του επίσης 5 οπότε οι δείκτες των suffixes του θα είναι οι 7, 8, 9, 10, 11.
Εκτελώντας την διαδικασία για το παραπάνω σχήμα το δέντρο πλέον θα είναι:




**Σχήμα 4.3:** Το δέντρο του Σχήματος 2.3 με σημειωμένους κόμβους πλέον

Στο παραπάνω σχήμα ξεκινώντας από την ρίζα και ακολουθώντας τα μονοπάτια που καταλήγουν σε κόμβο σημειωμένο ως xy, είναι φανερό ποια είναι τα κοινά substrings των strings x, y. Το μεγαλύτερο από αυτά είναι το επιθυμητό LCS και συγκεκριμένα το “CG”.

**Πολυπλοκότητα χρόνου:** Για είσοδο δύο strings 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).

# 5. Longest Common Subsequence (LCSub)

## 5.1. Υπάρχουσα μελέτη και εφαρμογές

Το LCSub πρόβλημα είναι το πρόβλημα εύρεσης 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 ανάμεσα 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 προγραμματισμού. 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 πολυπλοκότητα 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 strings είναι ίδια. Οι Rahman και Iliopoulos βελτιώνοντας έναν αλγόριθμο τους, παρουσίασαν έναν νέο αλγόριθμο χρόνου O(R*loglogn), όπου με εφαρμογή σε προβλήματα όπως εύρεση της 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. Συγκεκριμένα ιδιαίτερο ενδιαφέρον υπάρχει στην σύγκριση νουκλεοτιδίων των νουκλεϊκών οξέων καθώς η γενετική πληροφορία ενός οργανισμού κωδικοποιείται στη γραμμική διάταξη των νουκλεοτιδίων, τα οποία συνθέτουν τα μόρια DNA και RNA. Η σύγκριση αυτή χρησιμοποιείται για να χαρακτηρίσει την ομολογία, δηλαδή την αντιστοιχία ανάμεσα σε δύο ή περισσότερες σχετικές αλληλουχίες. Διαφοροποιήσεις του LCSub προβλήματος χρησιμοποιούνται για την μελέτη ομοιότητας δομών RNA. Οι Bereg και Zhu παρουσίασαν ένα νέο μοντέλο για δομική στοίχιση πολλαπλών RNA ακολουθιών. Επιπλέον χρησιμοποιείται στον υπολογισμό της απόστασης επεξεργασίας μεταξύ δύο αλληλουχιών RNA ώστε να είναι πιο αποδοτική η αναπαράσταση της δομικής πληροφορίας των RNA ακολουθιών, ιδιαίτερα των δευτερευόντων και τριτογενών δομικών χαρακτηριστικών ενός RNA που είναι από τους πιο σημαντικούς παράγοντες στον μοριακό μηχανισμό. 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. Επίσης το πρόβλημα της ολικής στοίχισης ακολουθιών είναι μια γενίκευση του LCSub προβλήματος. Αντί όμως να ερευνάται μόνο αν οι χαρακτήρες των ακολουθιών είναι ίδιοι ή διαφορετικοί, υπάρχει και μια συνάρτηση κόστους που καθορίζει τη σχέση ανάμεσα στους χαρακτήρες.

Κάποιες διαφοροποιήσεις και γενικεύσεις του LCSub προβλήματος είναι τα Constrained Longest Common Subsequence (CLCS), Multiple Longest Common Subsequence (MLCS), 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 απόστασης, το πρόβλημα διόρθωσης κυκλικών συμβολοσειρών, συγχώνευση strings και εύρεση των μικρότερων κοινών υπερακολουθιών.

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

- Naive Search
- Dynamic Programming
- Longest Increasing Subsequence

## 5.2. Naive Search

Η naive μέθοδος για την λύση του προβλήματος είναι να δημιουργηθούν όλες οι 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 κ.ο.κ.. 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 ακολουθιών 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 ακολουθιών είναι διαφορετικοί δηλαδή x[n-1]! =y[m-1] τότε το 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 |

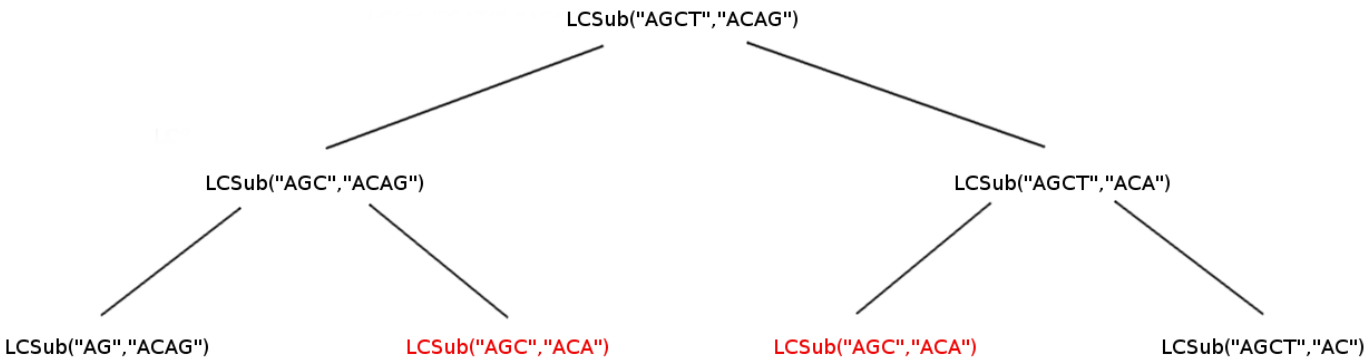
**Πίνακας 5.1:** Πίνακας απεικόνισης της εφαρμογής της naive μεθόδου για δύο strings

**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.

## 5.3. Dynamic Programming

Στον naive αλγόριθμο που περιγράφηκε πριν ελέγχονται όλοι οι συνδυασμοί των δύο ακολουθιών, μειώνοντας έτσι την ταχύτητά του. Επεκτείνοντας 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 5.1:** 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 strings “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 |

**Πίνακας 5.2:** 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.

## 5.4. Longest Increasing Subsequence (LIS)

Το LIS πρόβλημα είναι το πρόβλημα εύρεσης της μεγαλύτερης υποακολουθίας μιας δοσμένης ακολουθίας έτσι ώστε τα στοιχεία της υποακολουθίας να είναι ταξινομημένα με αυξανόμενη σειρά. Η ζητούμενη ακολουθία δεν χρειάζεται να είναι συνεχόμενη ή μοναδική. Για παράδειγμα το μήκος της LIS για τα στοιχεία {11, 25, 6, 40, 32, 55, 45, 79, 85} είναι 6 και η LIS είναι {11, 25, 40, 55, 79, 85}.

| array | 11 | 25 | 6 | 40 | 32 | 55 | 45 | 79 | 85 |
|---|---|---|---|---|---|---|---|---|---|
| LIS | 1 | 2 | - | 3 | - | 4 | - | 5 | 6 |

**Πίνακας 5.3:** Παράδειγμα εύρεσης της LIS για δοσμένη ακολουθία

Είναι ένα πρόβλημα που μελετάται σε διάφορες εφαρμογές στα μαθηματικά όπως σε αλγορίθμους, τυχαία θεωρία πινάκων και θεωρία αναπαράστασης. Ο αλγόριθμος που θα χρησιμοποιήσουμε εδώ για τον υπολογισμό του μήκους του LCSub μέσω του LIS περιγράφηκε από τους Crochemore και Porat (2010). Οι Crochemore και Porat περιγράφουν δύο αλγορίθμους για την εύρεση του LIS που μπορούν να χρησιμοποιηθούν έπειτα και για την εύρεση του LCSub. Στο παρόν κεφάλαιο θα γίνει χρήση του πρώτου από τους δύο αλγορίθμους.

Αρχικά περιγράφεται ο βασικός αλγόριθμος LIS που θα χρησιμοποιηθεί, ο οποίος ξεκινάει με τον υπολογισμό του μήκους του LIS. Ξεκινώντας από αριστερά προς τα δεξιά όταν βρεθεί ένα στοιχείο το οποίο είναι μεγαλύτερο από τα στοιχεία που έχουν ελεγχθεί μέχρι εκείνη την στιγμή, δημιουργείται μια αυξανόμενη υποακολουθία μεγαλύτερη από την προηγούμενη με αυτό το στοιχείο να είναι το τελευταίο της. Διαφορετικά αν το στοιχείο είναι μικρότερο γίνεται στοιχείο της υποακολουθίας που έχει υπολογιστεί μέχρι εκεί, χωρίς να αλλάξει το μέγεθος της καθώς αντικαθιστά το στοιχείο το οποίο είναι το αμέσως μεγαλύτερο του.

**Παράδειγμα:** Αν έχουμε την ακολουθία x=(23, 9, 12, 19, 6, 33, 5, 8, 15, 3, 4, 2) τότε οι διαδοχικές ακολουθίες που θα προκύπτουν μετά από κάθε έλεγχο που εκτελεί ο αλγόριθμος θα είναι:

**(23)**
**(9)**
**(9, 12)**
**(9, 12, 19)**
**(6, 12, 19)**
**(6, 12, 19, 33)**
**(5, 12, 19, 33)**
**(5, 8, 19, 33)**
**(5, 8, 15, 33)**
**(3, 8, 15, 33)**
**(3, 4, 15, 33)**
**(2, 4, 15, 33)**

Οπότε αφού το μήκος της ακολουθίας που προέκυψε μετά τον έλεγχο του τελευταίου στοιχείου της αρχικής ακολουθίας x είναι 4, αυτό θα είναι και το μήκος του LIS.

Αν επιπλέον θέλουμε να υπολογίσουμε και ποια είναι η ίδια η ακολουθία, αρκεί να επεκτείνουμε τον αλγόριθμο. Για κάθε τιμή που θα προστίθεται στην ακολουθία που υπολογίζεται, θα αποθηκεύεται και η αμέσως καλύτερη τιμή που υπήρχε στην ακολουθία πριν από αυτή που προστέθηκε. Έπειτα ξεκινώντας από το τελευταίο στοιχείο της ακολουθίας και ψάχνοντας προς τα πίσω για τις τιμές που άλλαξαν κάθε φορά θα μας δώσει την επιθυμητή LIS. Ο χρόνος εκτέλεσης του παραπάνω αλγορίθμου είναι O(n*logn).

Χρησιμοποιώντας αυτόν τον αλγόριθμο θα υπολογίσουμε το μήκος του LCSub δύο strings. Έστω x και y τα δύο strings μήκους n, για ένα ακέραιο αλφάβητο. Για κάθε γράμμα έστω a που εμφανίζεται στο x δημιουργείται μια ακολουθία p(a) με τις θέσεις εμφάνισης του γράμματος a στο y σε φθίνουσα διάταξη. Έστω S(x, y) η ακολουθία θέσεων του y αν ενώσουμε τις ακολουθίες p(x[i]) οπότε:

**S(x, y)=p(x[1])p(x[2]) . . . p(x[n])**

Το μήκος της S(x, y) είναι το σύνολο των κοινών στοιχείων ανάμεσα στο x, y το οποίο είναι το μήκος του συνόλου [(i, j) | x[i]=y[j]}, που είναι $O(n^2)$. Το συμπέρασμα λοιπόν είναι ότι για τα δύο μήκη θα ισχύει:

**LCSub(x, y)=LIS(S(x, y))**

Έπειτα επειδή η ακολουθία S(x, y) πιθανόν περιέχει διάφορες εμφανίσεις των ίδιων θέσεων θα την τροποποιήσουμε κατάλληλα. Για κάθε στοιχείο q που εμφανίζεται στη θέση i της S(x, y) δίνεται ένα νέο όνομα με τον βαθμό του ζευγαριού τιμών $(q, b_q)$ στην λεξικογραφικά ταξινομημένη λίστα όλων των αντίστοιχων ζευγαριών τιμών που θα προκύψουν. Η τιμή $b_q$ αντιστοιχεί στον αριθμό εμφανίσεων του στοιχείου q της θέσης i στην ακολουθία S(x, y). Αυτή η διαδικασία οδηγεί στην δημιουργία μιας ακολουθίας ακεραίων στην οποία μπορεί να εφαρμοστεί πλέον ο προηγούμενος LIS αλγόριθμος.

**Παράδειγμα:** Έστω δύο strings x=“ACGTCAG” και y=“CAGACGT”. Τότε οι ακολουθίες που σχηματίζονται για το κάθε στοιχείο είναι: p(A)= (4, 2), p(C)=(5, 1), p(G)=(6, 3), p(T)=(7) οπότε η ακολουθία S θα είναι:

**S(“ACGTCAG”, “CAGACGT”)=(4, 2, 5, 1, 6, 3, 7, 5, 1, 4, 2, 6, 3)**

και τα ζευγάρια τιμών με τον αριθμό των εμφανίσεων κάθε στοιχείου στην παραπάνω ακολουθία είναι:

**((4, 2), (2, 2), (5, 2), (1, 2), (6, 2), (3, 2), (7, 1), (5, 1), (1, 1), (4, 1), (2, 1), (6, 1), (3, 1))**

Όταν μετά αντικατασταθούν τα ζευγάρια από τους αντίστοιχους βαθμούς τους θα έχουμε την νέα ακολουθία:

**(8, 4, 10, 2, 12, 6, 13, 9, 1, 7, 3, 11, 5)**

για την οποία μπορεί να εφαρμοστεί ο LIS αλγόριθμος ο οποίος μας δίνει μήκος για την LIS 4 και η ίδια η ακολουθία είναι η (2, 6, 7, 11).

Το LCSub χρησιμοποιείται ευρέως στη σύγκριση ακολουθιών και στη κατά προσέγγιση αντιστοίχιση μοτίβων, ενώ επίσης επεκτείνεται και στην στοίχιση μεταξύ ακολουθιών. Στη βιοπληροφορική το FastA είναι ένα από τα πιο γρήγορα εργαλεία για την κατά προσέγγιση αναζήτηση με ένα πρότυπο σε μια βάση δεδομένων γονιδιωματικών ή πρωτεϊνικών αλληλουχιών. Ο αλγόριθμος πίσω από το λογισμικό βασίζεται σε μια ευρετική μέθοδο που εντοπίζει τις θέσεις όπου είναι πιθανό να βρεθεί το πρότυπο. Στη συνέχεια μέσω μιας τεχνικής δυναμικού προγραμματισμού η αναζήτηση επικεντρώνεται κοντά σε αυτές τις θέσεις. Στην τεχνική αυτή τρέχει ένας αλγόριθμος στοίχισης μέσα σε μία ζώνη γύρω από μια διαγώνιο του πίνακα του δυναμικού προγραμματισμού. Το εύρος αυτή της ζώνης είναι μια παράμετρος του λογισμικού που παρέχεται από τον χρήστη.

Η στοίχιση της ζώνης υπολογίζεται μέσω του υπολογισμού της LIS της ακολουθίας S(x, y)=p(x[1])p(x[2])...p(x[n]) όπου κάθε p(x[i]) περιορίζεται στην μειούμενη ακολουθία των θέσεων του x[i] στο τμήμα y[j-w...j+w], όπου w είναι η μεταβλητή που όπως αναφέρθηκε και πιο πάνω δίνεται από τον χρήστη και το j-i έχει σταθερή τιμή καθώς είναι ο δείκτης της επιλεγμένης διαγωνίου. Το μήκος της ακολουθίας S είναι O(n) και ο χρόνος υπολογισμού της LIS γίνεται O(n*loglogn).

**Πολυπλοκότητα χρόνου:** Αν k το μέγιστο μήκος της LCSub ανάμεσα σε δύο strings μήκους n και r ο αριθμός των ταιριασμάτων τους, τότε για τον υπολογισμό του k και την εύρεση του LCSub των δύο strings χρειάζεται χρόνος O(r*logk) ή ισοδύναμα $O(n^2 logk)$.

# 6. Longest Common Extension (LCE) πρόβλημα

## 6.1. Υπάρχουσα μελέτη και εφαρμογές

Το LCE πρόβλημα για ένα string έστω x και για ένα ζευγάρι τιμών i, j είναι το πρόβλημα εύρεσης του μέγιστου κοινού substring του 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 υπολογισμό των διαδοχικών επαναλήψεων (ακριβής ή κατά προσέγγιση), στον υπολογισμό παλίνδρομων και στο ταίριασμα τους με χρήση wildcards.

Το LCE πρόβλημα μπορεί να λυθεί αποδοτικά αν το string εισόδου υποστεί γραμμικού χρόνου επεξεργασία, ώστε το αποτέλεσμα για κάθε ζευγάρι i, j να μπορεί να υπολογιστεί σε σταθερό χρόνο. Δύο ισχυροί αλγόριθμοι χρησιμοποιούνται για αυτό το σκοπό. Ο πρώτος είναι ο σταθερού χρόνου υπολογισμός του <<χαμηλότερου>> κοινού προγόνου σε δέντρα. Όταν εφαρμοστεί σε ένα suffix δέντρο ενός string, υπολογίζει εύκολα την λύση του LCE προβλήματος. Ο δεύτερος χρησιμοποιεί σταθερού χρόνου υπολογισμό των Range Minimum Queries (RQM) σε arrays. Όταν εφαρμοστεί στην LCP διάταξη ενός string (που είναι μέρος της suffix arrays δομής δεδομένων του string) δίνει λύση για το LCE πρόβλημα, η οποία είναι και πιο αποδοτική σε πρακτικές εφαρμογές.

Οι δύο αυτές λύσεις από πρακτικής άποψης είναι αρκετά περίπλοκες. 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 αλγόριθμοι τους είναι οι καλύτεροι όταν εφαρμοστούν σε πρακτικές εφαρμογές, τόσο σε χρόνο όσο και σε χώρο, εκτός από τις περιπτώσεις που οι θέσεις i, j στο suffix array είναι πολύ κοντά όπου και έκαναν χρήση αλγορίθμου για απευθείας υπολογισμό των RQM. 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. Αργότερα χρησιμοποιώντας ιδέες από suffix arrays, δυναμικού προγραμματισμού και τεχνικές στοίχισης ο Crochemore παρουσίασε έναν αλγόριθμο για το LCCE πρόβλημα με ελαφρώς καλύτερο χρόνο εκτέλεσης.

Παρακάτω αρχικά παρουσιάζονται αρχικά δύο απλοί αλγόριθμοι για υπολογισμό του LCE και του LCE with k-mismatches και έπειτα εξετάζεται ένας αλγόριθμος που παρουσίασαν το 2017 οι Alamro H., Alzamel M., Iliopoulos C.S., Pissis S.P., Watts S., Sung WK. Ο

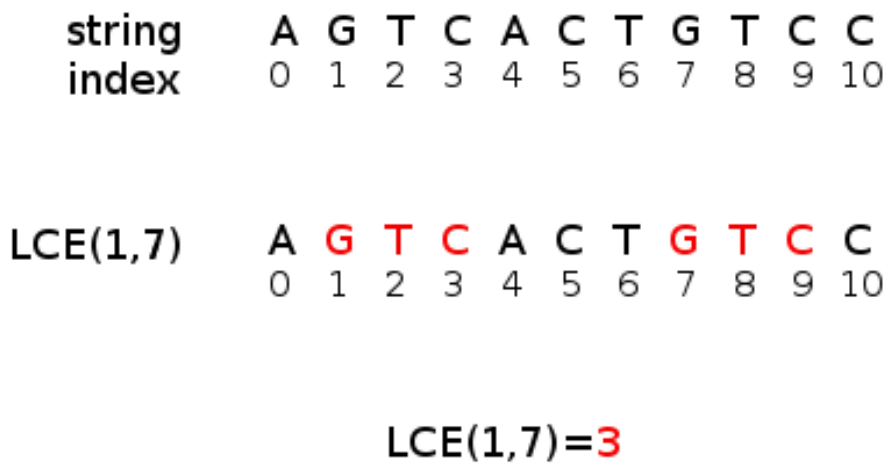
αλγόριθμος αυτός διαπιστώνει αν το string που του δίνεται ως είσοδος είναι k-Closed String και ποιο είναι το k-Closed Border του.

## 6.2. 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 6.1:** Εφαρμογή της Naive μεθόδου για εύρεση του LCE

**Πολυπλοκότητα χρόνου:** Αν n το μήκος του string εισόδου τότε για τον υπολογισμό του LCE στην χειρότερη περίπτωση χρειάζεται χρόνος O(n). 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 πρόκειται 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.

## 6.3. 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 6.2:** 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 στην είσοδο, 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.

**Πολυπλοκότητα χρόνου:** Αν n το μήκος του string εισόδου τότε για τον υπολογισμό του LCE-k στην χειρότερη περίπτωση χρειάζεται χρόνος O(n), καθς στην περίπτωση αυτή δεν θα υπάρχει κάποιο mismatch και ο αλγόριθμος θα συνεχίζει την αναζήτηση.

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

## 6.4. 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 strings 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 δοσμένου μεγέθους και αλφάβητου. Η Badkobeh είχε παρουσιάσει έναν αλγόριθμο για την μετατροπή ενός string μήκους n σε ακολουθία των μεγαλύτερων closed strings με πολυπλοκότητα 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 strings 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 στα δύο προηγούμενα κεφάλαια. Επίσης κάνει χρήση του αλγορίθμου Kangaroo, μιας μεθόδου που χρησιμοποιείται για εκτέλεση πολλαπλών LCE-k ερωτημάτων δοσμένου string χτίζοντας ένα suffix tree. Έπειτα ορίζονται 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 Border του x. Αν n≤1 ως k-Closed Border 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 Border του x. Αν n≤1 ως ασθενώς k-Closed Border 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 Border του x. Αν n≤1 ως ισχυρώς k-Closed Border ορίζεται το ε.

**Ορισμός 4:** Ένα string έστω x μήκους n καλείται ψευδώς 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 $\delta_H(u,v)\leq k$
- 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 $\delta^H(u,w)\leq k$

Τα u,v τότε αποκαλούνται ψευδώς k-Closed Border του x. Αν n≤1 ως ψευδώς k-Closed Border ορίζεται το ε.

Με βάση τους ορισμούς 1 και 4 προκύπτει το παρακάτω λήμμα:

**Λήμμα 1:** Αν το x είναι k-closed ⇔ υπάρχει k' , 0≤k'≤k, τέτοιο ώστε το x να είναι ψευδώς k'-closed

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:

**$LPM_k(x)[j]=LSM_k(x^R)[n-1-j]$**

**$LSM_k(x)[j]=LPM_k(x^R)[n-1-j]$**

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:

**$LPM_k(x)[j]=LCE_k(0, j)$ του x για $j \in [1, n-1]$**

**$LSM_k(x)[j]=LCE_k(0, n-1-j)$ του $x^R$ 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:

**Λήμμα 2:** Ένα string x μήκους n≥2 είναι k-closed όταν για έναν αριθμό k, 0≤k≤n υπάρχει ένα j ∈ {1, . . . , n − 1} και ένα k' ∈ {0, . . . , k} ώστε να ισχύουν οι εξής συνθήκες:

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

Για την απόδειξη του λήμματος πρώτα εξετάζεται τι συμβαίνει αν ισχύουν οι τρεις συνθήκες. Αν ισχύουν προκύπτει ότι το x είναι ψευδώς k'-closed και αφού ισχύει 0≤k'≤k με βάση και το λήμμα 1 προκύπτει ότι το x είναι k-closed.

Αντίστοιχα κάνοντας την ανάποδη υπόθεση, ότι το x είναι k-closed, πάλι από το λήμμα 1 προκύπτει ότι υπάρχει ένα k',  0≤k'≤k, για το οποίο το x είναι ψευδώς k-closed. Για αυτό το k' και με βάση τους ορισμούς που δόθηκαν προκύπτει τελικά ότι οι τρεις συνθήκες ικανοποιούνται.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x[j] | A | C | G | T | T | G | A | C | T | T | T | A | A | C | T |
| $LPM_2$[j] | -1 | 2 | 2 | 3 | 2 | 2 | 9 | 2 | 2 | 2 | 2 | 4 | 3 | 2 | 1 |
| $LSM_2$[j] | 1 | 2 | 3 | 4 | 3 | 2 | 2 | 2 | 9 | 4 | 3 | 2 | 2 | 3 | -1 |
| 1. j + $LPM_2$[j] = n | F | F | F | F | F | F | T | F | F | F | F | T | T | T | T |
| 2. $LPM_2$_peaks[j] | T | F | F | T | F | F | T | F | F | F | F | F | F | F | F |
| 3. $LSM_2$_peaks[n - 1 - j] | T | F | F | F | F | T | T | F | F | F | F | F | F | F | F |
| 2-Closed Borders | F | F | F | F | F | F | T | F | F | F | F | F | F | F | F |

**Πίνακας 6.1:** Εφαρμογή του αλγορίθμου για 2-closed string περίπτωση

Επίσης για την υλοποίηση του αλγορίθμου γίνεται χρήση του LCE-k που αναφέρθηκε στο προηγούμενο κεφάλαιο και της συνάρτησης Reverse(x) η οποία δέχεται ως είσοδο ένα string και επιστρέφει το αντεστραμμένο string.

Από όλα τα παραπάνω το συμπέρασμα για τον αλγόριθμο που παρουσιάστηκε είναι:

**Θεώρημα:** Για δοσμένο string x μήκους n ενός ακεραίου αλφαβήτου και για έναν φυσικό αριθμό k όπου 0<k<n, το k-Closed Border του x, εάν υπάρχει, μπορεί να βρεθεί σε χρόνο O(k*n) και σε O(n) χώρο

**Πολυπλοκότητα χρόνου:** Η απόδοση του αλγορίθμου εξαρτάται από τον υπολογισμό των δύο ακολουθιών $LPM_k(x)$ και $LSM_k(x)$, πάνω στις οποίες βασίζεται το χτίσιμό του. Σύμφωνα με το Λήμμα 5 ο υπολογισμός αυτών των δύο ακολουθιών θέτει το όριο για τον χρόνο που χρειάζεται για να διαπιστωθεί αν ένα string είναι k-closed, για κάθε τιμή του k', $0 \leq k' \leq k$. Για τον έλεγχο ενός k' , η πρώτη συνθήκη χρειάζεται χρόνο O(n) για να ελέγξει όλα τα πιθανά j. Οι συνθήκες 2 και 3 χρειάζονται χρόνο O(1) για κάθε j έχοντας πρώτα προεπεξεργαστεί τις ακολουθίες $LPM_k(x)$ και $LSM_k(x)$ σε χρόνο O(n) για να διαπιστώσουν που βρίσκονται τα κατάλληλα άκρα των δύο ακολουθιών. Οπότε ο συνολικός χρόνος για έλεγχο όλων των j και k' είναι O(k*n).

# 7. Πειραματικά Αποτελέσματα

Οι δοκιμές των αλγορίθμων που παρουσιάστηκαν στα προηγούμενα κεφάλαια έγιναν σε περιβάλλον Spark με χρήση πραγματικών δεδομένων. Τα δεδομένα που χρησιμοποιήθηκαν είναι βιολογικές ακολουθίες οι οποίες είναι διαθέσιμες στην βάση δεδομένων του National Center for Biotechnology Information (NCBI). Συγκεκριμένα χρησιμοποιήθηκαν μέρη από τις ακολουθίες των γονιδιωμάτων των βακτηρίων Escherichia coli K-12 και Streptococcus pneumoniae R6.

## 7.1. Μετρήσεις για το Longest Common Prefix πρόβλημα

Αρχικά έπειτα από προεπεξεργασία των δεδομένων ώστε να είναι σε κατάλληλη μορφή για τον σκοπό που τα θέλουμε, έγινε δοκιμή των αλγορίθμων Longest Common Prefix και τα αποτελέσματα που προέκυψαν παρουσιάζονται στους παρακάτω πίνακες. Αρχικά δίνοντας ως είσοδο 3 strings με μεγέθη 300, 450 και 125 χαρακτήρων προκύπτει LCP μεγέθους 76 χαρακτήρων και τα αποτελέσματα των χρόνων εκτέλεσης των αλγορίθμων είναι τα παρακάτω:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **Word by Word Matching** | **n*m** | **2.017** |
| **Character by Character Matching** | **n*m** | **1.554** |
| **Divide and Conquer** | **n*m** | **1.516** |
| **Binary Search** | **n*m*logm** | **0.952** |

**Πίνακας 7.1:** Αποτελέσματα 1ης δοκιμής των LCP αλγορίθμων

Έπειτα για είσοδο 3 strings με μεγέθη 2455, 2605 και 2060 χαρακτήρες προκύπτει LCP με μήκος 250 χαρακτήρων και οι χρόνοι εκτέλεσης των αλγορίθμων περιέχονται στον παρακάτω πίνακα:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **Word by Word Matching** | **n*m** | **2.876** |
| **Character by Character Matching** | **n*m** | **2.682** |
| **Divide and Conquer** | **n*m** | **1.733** |
| **Binary Search** | **n*m*logm** | **1.092** |

**Πίνακας 7.2:** Αποτελέσματα 2ης δοκιμής των LCP αλγορίθμων

Στη συνέχεια η είσοδος που δίνουμε στους αλγορίθμους είναι 3 strings με μεγέθη 8504, 8884 και 9948 χαρακτήρες και παίρνουμε ένα LCP μεγέθους 1054 χαρακτήρων. Ακολουθούν οι μετρήσεις των χρόνων εκτέλεσης:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **Word by Word Matching** | **n*m** | **3.587** |
| **Character by Character Matching** | **n*m** | **3.392** |
| **Divide and Conquer** | **n*m** | **1.969** |
| **Binary Search** | **n*m*logm** | **1.278** |

**Πίνακας 7.3:** Αποτελέσματα 3ης δοκιμής των LCP αλγορίθμων

Τέλος γίνονται μετρήσεις για strings με μήκη 16845, 21350 και 18790 όπου το prefix που προέκυψε είχε μήκος 2530 χαρακτήρων. Οι χρόνοι εκτέλεσης που καταγράψαμε είναι οι εξής:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **Word by Word Matching** | **n*m** | **5.749** |
| **Character by Character Matching** | **n*m** | **5.388** |
| **Divide and Conquer** | **n*m** | **2.611** |
| **Binary Search** | **n*m*logm** | **1.442** |

**Πίνακας 7.4:** Αποτελέσματα 4ης δοκιμής των LCP αλγορίθμων

Στις παραπάνω μετρήσεις ο πιο γρήγορος αλγόριθμος σε όλες τις περιπτώσεις ήταν ο Binary Search. Οι υπόλοιποι τρεις αλγόριθμοι παρότι θεωρητικά έχουν ίδια πολυπλοκότητα χρόνου στην πράξη αποδίδουν διαφορετικά. Ο πιο αργός είναι ο Word by Word Matching, έπειτα ακολουθεί ο Character by Character Matching που είναι λίγο πιο γρήγορος και ο πιο γρήγορος από τους τρεις είναι ο Divide and Conquer. Ο Binary Search είναι ο πιο γρήγορος καθώς εξετάζει τους χαρακτήρες από τον πρώτο μέχρι αυτόν που βρίσκεται στη θέση ίση με το μήκος του μικρότερου string. Επίσης διαιρεί περαιτέρω αυτό το substring στην μέση και αν το πρώτο μισό δεν είναι μέρος του prefix δεν θα εκτελέσει αναζήτηση στο άλλο μισό. Ο Word by Word είναι πιο αργός καθώς θα χρειαστεί να εξετάσει όλα τα strings και το prefix που θα προκύπτει κάθε φορά ανάμεσα στα ζεύγη από τα strings του συνόλου, μπορεί να είναι μεγαλύτερο από το τελικό prefix οπότε εκτελούνται και επιπλέον αχρείαστοι υπολογισμοί. Αντίθετα ο Character by Character εξετάζει μεν όλα τα strings αλλά κάθε φορά ένα χαρακτήρα στην ίδια θέση όλων τον strings. Έτσι μόλις κάποιος χαρακτήρας σε οποιοδήποτε string δεν είναι κοινός με των άλλων η αναζήτηση διακόπτεται, το επιθυμητό prefix είναι αυτό που έχει προκύψει μέχρι εκείνη τη στιγμή και δεν εκτελούνται περιττοί υπολογισμοί όπως πριν. Ο Divide and Conquer είναι πιο γρήγορος από τους δύο προηγούμενους καθώς από την φύση του είναι αλγόριθμος με σκοπό την μείωση χρόνου εκτέλεσης, σπάζοντας το πρόβλημα σε μικρότερα υποπροβλήματα και υπολογίζοντας την τελική λύση μέσω των λύσεων των υποπροβλημάτων.

## 7.2. Μετρήσεις για το Longest Common Substring πρόβλημα

Στη συνέχεια ακολούθησαν οι δοκιμές των αλγορίθμων που χρησιμοποιήθηκαν στην επίλυση του προβλήματος Longest Common Substring. Οι αλγόριθμοι αυτοί δέχονται ως είσοδο 2 strings και για την πρώτη δοκιμή δόθηκαν ως είσοδος δύο strings που τα μεγέθη τους είναι 300 και 280 χαρακτήρων. Το LCS των δύο strings που προέκυψε έχει μήκος 7 χαρακτήρων. Ο πίνακας που ακολουθεί περιέχει τα αποτελέσματα των χρόνων εκτέλεσης που μετρήσαμε:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης(s) |
|---|---|---|
| **Naive Search** | **$n*m^2$** | **3.824** |
| **Dynamic Programming** | **n*m** | **0.014** |
| **Suffix Array** | **n+m** | **0.003** |
| **Suffix Tree** | **$(n+m)^2$** | **0.029** |

**Πίνακας 7.5:** Αποτελέσματα 1ης δοκιμής των LCS αλγορίθμων

Στην επόμενη δοκιμή δίνουμε δύο μεγαλύτερα strings για είσοδο και συγκεκριμένα ένα μεγέθους 4575 και ένα 4270 χαρακτήρων. Το LCS των δύο strings που προέκυψε έχει μήκος 13 χαρακτήρων και οι μετρήσεις που πήραμε είναι οι παρακάτω:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Naive Search** | **$n*m^2$** | **$14.157*10^3$** |
| **Dynamic Programming** | **n*m** | **3.314** |
| **Suffix Array** | **n+m** | **0.221** |
| **Suffix Tree** | **$(n+m)^2$** | **0.303** |

**Πίνακας 7.6:** Αποτελέσματα 2ης δοκιμής των LCS αλγορίθμων

Έπειτα χρησιμοποιούμε δύο πιο μεγάλα από πριν strings με μεγέθη 7500 και 7000 χαρακτήρων και το LCS είναι ξανά μήκους 13 χαρακτήρων. Οι χρόνοι εκτέλεσης των αλγορίθμων περιέχονται στον παρακάτω πίνακα:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης(s) |
|---|---|---|
| **Naive Search** | **$n*m^2$** | **$58.136*10^3$** |
| **Dynamic Programming** | **n*m** | **8.018** |
| **Suffix Array** | **n+m** | **0.417** |
| **Suffix Tree** | **$(n+m)^2$** | **0.444** |

**Πίνακας 7.7:** Αποτελέσματα 3ης δοκιμής των LCS αλγορίθμων

Τέλος δίνουμε ως είσοδο στους αλγορίθμους δύο strings 14925 και 14070 χαρακτήρων με το LCS που προκύπτει να έχει μήκος 23 χαρακτήρων. Οι μετρήσεις των χρόνων εκτέλεσης είναι οι εξής:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Naive Search** | **$n*m^2$** | **$482.529*10^3$** |
| **Dynamic Programming** | **n*m** | **33.222** |
| **Suffix Array** | **n+m** | **1.585** |
| **Suffix Tree** | **$(n+m)^2$** | **1.396** |

**Πίνακας 7.8:** Αποτελέσματα 4ης δοκιμής των LCS αλγορίθμων

Τα αποτελέσματα δείχνουν ότι ο πιο αργός αλγόριθμος με διαφορά ήταν ο πρώτος που εφαρμόζει απλή προσέγγιση καθώς χρειάζεται αφού πρώτα βρεθούν όλα τα πιθανά strings για το ένα από τα δύο strings εισόδου, να γίνει αναζήτηση για το καθένα αν είναι substring του άλλου και το μεγαλύτερο από τα κοινά θα είναι το επιθυμητό LCS. Η μέθοδος Dynamic Programming είναι η αμέσως πιο γρήγορη. Σε αυτή την προσέγγιση θα δημιουργηθεί ένας πίνακας που θα περιέχει τα μήκη των μέγιστων κοινών suffixes των substrings των δύο strings, οι λύσεις κάθε φορά θα αποθηκεύονται στον πίνακα ώστε να χρησιμοποιούνται στους επόμενους υπολογισμούς και να μην υπολογίζονται ξανά. Με χρήση λοιπόν αναδρομής και μνήμης μειώνεται ο χρόνος εκτέλεσης. Η προσέγγιση με κατασκευή Suffix Array είναι αρκετά απλή και γρηγορότερη από τις δύο προηγούμενες σε όλες τις περιπτώσεις ενώ είναι γρηγορότερη και από τον Suffix Tree αλγόριθμο εκτός από την τελευταία περίπτωση. Ένα Suffix Array είναι μια απλή δομή δεδομένων που περιέχει όλες τις πληροφορίες που χρειαζόμαστε για την εύρεση του LCS. Δημιουργείται ένας πίνακας με όλα τα πιθανά suffixes του string που προκύπτει από τον συνδυασμό των strings εισόδου και αφού ταξινομηθεί με λεξικογραφική σειρά με χρήση ενός αλγορίθμου LCP, βρίσκουμε το LCP ανάμεσα σε κάθε τιμή με την επόμενη. Το μεγαλύτερο από τα LCPs είναι το επιθυμητό LCS. Ο Suffix Tree αλγόριθμος ειναι αρκετά πιο πολύπλοκος από τις προηγούμενες μεθόδους, ενώ για τα αρκετά μικρά strings της πρώτης δοκιμής ήταν πιο αργός ακόμα και από την Dynamic Programming μέθοδο. Για την εφαρμογή του χρειάζεται η κατασκευή του Generalized Suffix Tree το οποίο κατασκευάστηκε με χρήση του αλγορίθμου McCreight. Έπειτα από τα φύλλα που θα έχουν δείκτες ανάλογα με το ποιο string περιέχει το suffix τους, θα επιλέξουμε το μονοπάτι με το μεγαλύτερο suffix κοινό και για τα δύο strings. Παρατηρούμε ότι όσο αυξάνεται το μέγεθος των strings εισόδου η απόδοση του σε σχέση με τον Suffix Array αλγόριθμο βελτιώνεται. Μάλιστα στην τελευταία δοκιμή που χρησιμοποιήθηκαν τα μεγαλύτερα strings των δοκιμών του LCS προβλήματος, ο χρόνος εκτέλεσης του ήταν μικρότερος από αυτόν της μεθόδου με χρήση Suffix Array.

## 7.3. Μετρήσεις για το Longest Common Subsequence πρόβλημα

Στους παρακάτω πίνακες παραθέτουμε τα αποτελέσματα που προέκυψαν από δοκιμές που πραγματοποιήθηκαν για το Longest Common Subsequence πρόβλημα. Αρχικά δόθηκαν δύο strings ως είσοδος, με μέγεθος 20 χαρακτήρων το καθένα. Το LCSub που πήραμε ως αποτέλεσμα έχει μήκος 9 χαρακτήρων και τα αποτελέσματα ήταν τα εξής:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Naive Search** | **$2^n$** | **102.483** |
| **Dynamic Programming** | **n*m** | **0.001** |
| **Με χρήση LIS** | **$n^2$*logk** | **0.006** |

**Πίνακας 7.9:** Αποτελέσματα 1ης δοκιμής των LCSub αλγορίθμων

Στον παραπάνω πίνακα ο χρόνος της Naive προσέγγισης είναι τεράστιος ακόμα και για μικρά δεδομένα οπότε είναι φανερό ότι είναι ακατάλληλη μέθοδος για επίλυση του LCSub προβλήματος. Στην συνέχεια θα συνεχίσουμε πειράματα για τις άλλες δύο μεθόδους που είναι σαφώς πιο αποδοτικές. Αυτή τη φορά τα αποτελέσματα είναι για δύο strings 83 και 81 χαρακτήρων με το LCSub που προέκυψε να έχει μήκος 46 χαρακτήρων. Οι χρόνοι εκτέλεσης που καταγράφηκαν είναι οι παρακάτω:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Dynamic Programming** | **n*m** | **0.009** |
| **Με χρήση LIS** | **$n^2$*logk** | **0.617** |

**Πίνακας 7.10:** Αποτελέσματα 2ης δοκιμής των LCSub αλγορίθμων

Ακολουθούν τα αποτελέσματα σύγκρισης 2 strings 155 κ 158 χαρακτήρων των οποίων το LCSub έχει μήκος 90 χαρακτήρων:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Dynamic Programming** | **n*m** | **0.012** |
| **Με χρήση LIS** | **$n^2$*logk** | **21.415** |

**Πίνακας 7.11:** Αποτελέσματα 3ης δοκιμής των LCSub αλγορίθμων

Τέλος στην τελευταία δοκιμή που εκτελούμε για το LCSub πρόβλημα, δίνουμε είσοδο στους αλγόριθμους 2 strings 300 κ 280 χαρακτήρων που μας δίνουν ως αποτέλεσμα ένα LCSub 179 χαρακτήρων. Οι χρόνοι εκτέλεσης των αλγορίθμων παρουσιάζονται στον παρακάτω πίνακα:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (s) |
|---|---|---|
| **Dynamic Programming** | **n*m** | **0.037** |
| **Με χρήση LIS** | **$n^2$*logk** | **320.529** |

**Πίνακας 7.12:** Αποτελέσματα 4ης δοκιμής των LCSub αλγορίθμων

Η Naive μέθοδος για τον υπολογισμό του LCSub είναι με μεγάλη διαφορά η πιο αργή μέθοδος. Η πολυπλοκότητά της είναι εκθετική και στην χειρότερη περίπτωση είναι $O(2^n)$, οπότε κρίνεται ακατάλληλη για χρήση με μεγάλου μήκους ακολουθίες σαν αυτές που χρησιμοποιούνται για υπολογισμούς στην βιοπληροφορική, καθώς ο χρόνος εκτέλεσης της είναι τεράστιος. Η Dynamic Programming μέθοδος είναι σαφώς πιο γρήγορη γιατί ενώ ακολουθεί παρόμοιο τρόπο με την Naive, εδώ χρησιμοποιείται σαν μνήμη ένας πίνακας όπου αποθηκεύονται τα προσωρινά αποτελέσματα ώστε να χρησιμοποιηθούν σε επόμενους υπολογισμούς. Η προσέγγιση με LIS είναι επίσης πιο γρήγορη. Σε αυτή τη μέθοδο αφού δημιουργήσουμε μια ακολουθία με βάση τις θέσεις που εμφανίζονται οι χαρακτήρες τους ενός string στο άλλο, εφαρμόζουμε ένα LIS αλγόριθμο που θα μας δώσει το επιθυμητό LCSub. Αν k το μέγιστο μήκος της LCSub και τα δύο strings είναι μήκους n ο χρόνος εκτέλεσης είναι $O(n^2*logk)$.

## 7.4. Μετρήσεις για το Longest Common Extension πρόβλημα

Στις παρακάτω δοκιμές για το LCE πρόβλημα, επιλέξαμε θέσεις στα strings και αριθμό mismatches έτσι ώστε οι αλγόριθμοι LCE-k και k-Closed Border να μας δώσουν το ίδιο αποτέλεσμα και να συγκρίνουμε αν είναι ανάλογοι των διαφορετικών πολυπλοκοτήτων των δύο αλγορίθμων. Στον πίνακα που ακολουθεί παρουσιάζονται τα αποτελέσματα της δοκιμής σε ένα string 302 χαρακτήρων, όπου το strings που επιστρέφει η πρώτη μέθοδος έχει μήκος 15 χαρακτήρων, ενώ το string που επιστρέφουν η δεύτερη και τρίτη μέθοδος για k=5 είναι μήκους 38 χαρακτήρων:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **LCE naive** | **n** | **0.635** |
| **LCE-k naive** | **n** | **0.771** |
| **k-Closed Border** | **k*n** | **62.495** |

**Πίνακας 7.13:** Αποτελέσματα 1ης δοκιμής για τους αλγόριθμους σχετικούς με extension και k-Closed Border

Έπειτα στον επόμενο πίνακα καταγράφονται τα αποτελέσματα των χρόνων εκτέλεσης των αλγορίθμων για ένα string 1020 χαρακτήρων. Ο LCE αλγόριθμος επιστρέφει ένα strings 38 χαρακτήρων ενώ οι άλλοι δύο αλγόριθμοι για k=15 δίνουν ως αποτέλεσμα ένα string μεγέθους 91 χαρακτήρων:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **LCE naive** | **n** | **0.672** |
| **LCE-k naive** | **n** | **0.802** |
| **k-Closed Border** | **k*n** | **332.193** |

**Πίνακας 7.14:** Αποτελέσματα 2ης δοκιμής για τους αλγόριθμους σχετικούς με extension και k-Closed Border

Στην συνέχεια χρησιμοποιείται ένα ακόμα μεγαλύτερο string μεγέθους 2270 χαρακτήρων. Ο πρώτος αλγόριθμος επιστρέφει ένα extension μήκους 261 ενώ οι αλλοι για k=30 ένα μήκους 377 και τα αποτελέσματα των χρόνων εκτέλεσης είναι τα εξής:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **LCE naive** | **n** | **0.736** |
| **LCE-k naive** | **n** | **0.895** |
| **k-Closed Border** | **k*n** | **$1.469*10^3$** |

**Πίνακας 7.15:** Αποτελέσματα 3ης δοκιμής για τους αλγόριθμους σχετικούς με extension και k-Closed Border

Τέλος ο παρακάτω πίνακας περιέχει τα αποτελέσματα της δοκιμής σε ένα string 5368 με τον LCE αλγόριθμο να επιστρέφει ένα extension μήκους 505 χαρακτήρων και οι άλλοι δύο αλγόριθμοι για k=30 ένα string μήκους 543 χαρακτήρων, με τους χρόνους εκτέλεσης όλων των παραπάνω αλγορίθμων να είναι οι εξής:

| Προσέγγιση | Πολυπλοκότητα χρόνου | Χρόνος εκτέλεσης (ms) |
|---|---|---|
| **LCE naive** | **n** | **1.684** |
| **LCE-k naive** | **n** | **1.751** |
| **k-Closed Border** | **k*n** | **$8.349*10^6$** |

**Πίνακας 7.16:** Αποτελέσματα της τελευταίας δοκιμής για τους αλγόριθμους σχετικούς με extension και k-Closed Border

Για τα ίδια αποτέλεσματα όπως ήταν αναμενόμενο, η απλοική προσέγγιση του LCE-k αλγορίθμου, που είναι επέκταση του LCE Naive αλγορίθμου, είχε πολύ μικρότερο χρόνο εκτέλεσης καθώς η χρονική πολυπλοκότητα του Closed Border αλγορίθμου είναι k*n, όπου k ο αριθμός των mismatches που δώσαμε ως είσοδο ενώ του LCE-k είναι n. Επίσης παρότι η LCE Naive μέθοδος είναι αρκετά απλή, καθώς ουσιαστικά υπολογίζει το μέγιστο κοινό prefix ανάμεσα σε δύο θέσεις στο ίδιο strings, φαίνεται να έχει καλή απόδοση.

# 8. Συμπεράσματα

Στόχος της παρούσας εργασίας ήταν η μελέτη αλγορίθμων επίλυσης προβλημάτων με strings για βιολογικά δεδομένα σε περιβάλλον Spark και γλώσσα προγραμματισμού Python. Η χρήση Spark διευκόλυνε και επιτάχυνε την επεξεργασία των μεγάλου μεγέθους βιολογικών ακολουθιών, ενώ σε αυτό συνέβαλε και η ευελιξία και ευκολία στην χρήση της γλώσσας Python.

Σχετικά με το Longest Common Prefix παρατηρήσαμε ότι οι χρόνοι εκτέλεσης των αλγορίθμων δεν είναι ανάλογοι με τις χρονικές τους πολυπλοκότητες με εξαίρεση τον Binary Search που αποδεικνύεται και στην πράξη ο πιο γρήγορος. Οι άλλοι τρεις αλγόριθμοι παρότι θεωρητικά έχουν την ίδια πολυπλοκότητα σε πραγματικά δεδομένα είχαν διαφορετική απόδοση. Ο Divide and Conquer αλγόριθμος ήταν ο πιο αποδοτικός από τους τρεις και ακολουθούσαν με μικρή διαφορά μεταξύ τους οι Character by Character και Word by Word.

Όσον αφορά το Longest Common Substring τα αποτελέσματα μεταβάλλονταν ανάλογα με το μέγεθος των δεδομένων. Ο πρώτος αλγόριθμος, που εφαρμόζει απλοική προσέγγιση, όπως ήταν αναμενόμενο ήταν ο πιο αργός. Ο αλγόριθμος Dynamic Programming ήταν πιο γρήγορος από τον Naive αλγόριθμο, ενώ στην πρώτη δοκιμή όπου χρησιμοποιήθηκαν τα μικρότερα σε μέγεθος δεδομένα ήταν πιο γρήγορος και από τη μέθοδο με Suffix Tree. Αυξάνοντας όμως το μέγεθος των δεδομένων εισόδου ο Suffix Tree αλγόριθμος απέδωσε καλύτερα από τον αλγόριθμο δυναμικού προγραμματισμού, ενώ στην τελευταία δοκιμή ξεπέρασε και τον Suffix Array αλγόριθμο που μέχρι τότε ήταν ο πιο γρήγορος. Η προσέγγιση με χρήση Suffix Array ήταν με διαφορά η πιο γρήγορη αρχικά, ενώ η κατανόηση και η υλοποίηση της ήταν αρκετά απλή. Για πολύ μεγάλα δεδομένα όμως υπερτερούσε ο Suffix Tree αλγόριθμος, ενώ σε μελλοντική μελέτη ο McCreight αλγόριθμος που χρησιμοποιήθηκε για το χτίσιμο του δέντρου μπορεί να αντικατασταθεί από χρήση αλγορίθμου Ukonnen για ακόμα καλύτερα αποτελέσματα.

Στους αλγορίθμους επίλυσης του Longest Common Subsequence τα αποτελέσματα ήταν κοντά στα αναμενόμενα. Η απλοική προσέγγιση αποδείχτηκε και στην πράξη ακατάλληλη για χρήση με δεδομένα τόσο μεγάλου μεγέθους, ενώ ο δυναμικός προγραμματισμός επέφερε τα καλύτερα αποτελέσματα. Η προσέγγιση με χρήση Longest Increasing Subsequence είχε επίσης καλά αποτελέσματα αν και χειρότερα του Dynamic Programming, ενώ μελλοντικά με χρήση γρηγορότερου αλγορίθμου για τον υπολογισμό του LIS μπορεί να αποδώσει ακόμα καλύτερα.

Τέλος οι αλγόριθμοι για υπολογισμό του Longest Common Extension και k-Closed Border είχαν επίσης τα αποτελέσματα που περιμέναμε. Οι αλγόριθμοι υπολογισμού k-Closed Border και LCE-k έχουν στην πράξη απόδοση ανάλογη της θεωρητικής τους χρονικής πολυπλοκότητας ενώ η απλοική προσέγγιση για το LCE αποδίδει πολύ καλά παρότι θεωρητικά αποτελεί την πιο αργή μέθοδο για την επίλυση του LCE προβλήματος.

# 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. Boucher C, Ma B. *Closest string with outliers*. BMC Bioinformatics. 2011;12(Suppl 1):S55. Doi:10.1186/1471-2105-12-S1-S55.

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

7. Dibyajyoti, S., Talha Bin, E. & Swati, P., (2013). *Bioinformatics:The effects on the cost of drug discovery*. Galle Medical Journal. 18(1), pp.44–50.

8. 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.

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

10. Ferdous SM, Rahman MS (2015) *An Integer Programming Formulation of the Minimum Common String Partition Problem*. PLoS ONE 10(7): e0130266. https://doi.org/10.1371/journal.pone.0130266.

11. Fischer, Johannes; Heun, Volker (2007). *A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array*. Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. Lecture Notes in Computer Science. 4614. p. 459.

12. 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.

13. Henikoff S, Henikoff JG. A*mino acid substitution matrices from protein blocks.* Proceedings of the National Academy of Sciences of the United States of America. 1992;89(22):10915-10919.

14. J. Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, Louxin Zhang, D*istinguishing string selection problems, Information and Computation*, Volume 185, Issue 1, 2003, Pages 41-55, ISSN 0890-5401.

15. Jiang, Tao, Guohui Lin, Bin Ma and Kaizhong Zhang. *"A General Edit Distance between RNA Structures."* Journal of computational biology : a journal of computational molecular cell biology 9 2 (2002): 371-88.

16. Kasai, T.; Lee, G.; Arimura, H.; Arikawa, S.; Park, K. (2001). *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*. Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science. 2089. pp. 181–192.

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

18. Kevin Lanctot, J. (2004). *Some String Problems in Computational Biology.*

19. 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.

20. 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.

21. Madrigal P, Sela N, Lin SM (2011) *PLoS Computational Biology* Conference Postcards from I SMB/ECCB 2011. PLoS Comput Biol 7(11): e1002259. https://doi.org/10.1371/journal.pcbi.1002259

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

23. Maxime Crochemore, Ely Porat, *Fast computation of a longest increasing subsequence and application, Information and Computation*, Volume 208, Issue 9, 2010, Pages 1054-1059, ISSN 0890-5401.

24. Muir, P., Li, S., Lou, S., Wang, D., Spakowicz, D. J., Salichos, L., … Gerstein, M. (2016). The real cost of sequencing: scaling computation to keep pace with data generation. Genome Biology, 17, 53. http://doi.org/10.1186/s13059-016-0917-0.

25. Nong, Ge; Zhang, Sen; Chan, Wai Hong (2009). *Linear Suffix Array Construction by Almost Pure Induced-Sorting*. 2009 Data Compression Conference. p. 193.

26. Overby CL, Tarczy-Hornoch P. Personalized medicine: challenges and opportunities for translational bioinformatics. *Personalized medicine*. 2013;10(5):453-462.

27. P. Bille, I.L. Gørtz, B. Sach, and H.W. Vildhøj. Time-Space Trade-Offs for Longest Common Extensions. In Proc. 23rd CPM (LNCS 7354), pages 293–305, 2012.

28. 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.

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

30. Rahmann, Sven. *"Fast Large Scale Oligonucleotide Selection Using the Longest Common Factor Approach."* Journal of bioinformatics and computational biology 1 2 (2003): 343-61.

31. R.M. Karp and M.O. Rabin. *Efficient Randomized Pattern-Matching Algorithms*. IBM J. Res. Dev., 31(2):249–260, 1987.

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

33. Singh, Virdi Sabegh, Hong Wang and Robert A. Walker. "*Solving the Longest Common Subsequence ( LCS ) Problem using the Associative ASC Processor with Reconfigurable 2 D Mesh."* (2006).

34. 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.

35. 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.

36. 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.

## ΙΣΤΟΣΕΛΙΔΕΣ


1. *A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets,*
   **https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html**

2. *Aims of Bioinformatics,*
   **http://www.biology-today.com/bioinformatics-biostatistics/aims-of-bioinformatics/**

3. *Algorithms,*
   **https://www.geeksforgeeks.org/fundamentals-of-algorithms/**

4. *Apache Spark™ is a unified analytics engine for large-scale data processing,*
   **https://spark.apache.org/**

5. *Basic Local Alignment Search Tool,*
   **https://blast.ncbi.nlm.nih.gov/Blast.cgi**

6. *Bioinformatics,*
   **https://www.britannica.com/science/bioinformatics**

7. *Common methods in RDDs*,
   **http://www.prathapkudupublog.com/2018/02/common-methods-in-rdd.html**

8. *Dataset*,
   **https://spark.apache.org/docs/2.3.0/api/java/index.html org/apache/spark/sql/Dataset.html**

9. *DNA sequence at the National Museum of Emerging Science and Innovation, Miraikan, 8 June 2008, MIKI Yoshihito,*
   **https://commons.wikimedia.org/wiki/File:DNA_sequence.jpg**

10. *File:Alignment-Comparison-En.png,*
    **https://commons.wikimedia.org/wiki/File:Alignment-Comparison-En.png**

11. *File:EMBL-Bank growth.svg, Growth of EMBL-Bank database (1982-2012 inclusive) based on release notes 114 published on December 2012, Author: Ben Moore,*
    **https://commons.wikimedia.org/wiki/File:EMBL-Bank_growth.svg**

12. *Introduction to Datasets,*
    **https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-datasets.html**

13. *Knuth–Morris–Pratt algorithm, YouTube video, 4:24, posted by "YouTube," December 5, 2014,*
**https://youtu.be/5i7oKodCRJo**

14. *Learn about Apache Spark APIs and Best Practises*,
**https://databricks.com/blog/2017/09/12/learn-about-apache-spark-apis-and-best-practices.html**

15. *Logistic regression in Hadoop and Spark,*
**https://spark.apache.org/**

16. *Logo of Apache Spark, Apache software foundation, 2013,*
**https://spark.apache.org/images/spark-logo.eps**

17. *Sparks compatibility image, Databricks,*
**https://databricks.com/spark/about**

18. *Spark: Low Latency, Massively Parallel Processing Framework,*
**https://dzone.com/articles/spark-low-latency-massively**

19. *Spark RDDs Simplified,*
**http://vishnuviswanath.com/spark_rdd.html**

20. *The cost of sequencing a human genome,*
**https://www.genome.gov/27565109/the-cost-of-sequencing-a-human-genome/**

21. *The future of the future: Spark, big data insights, streaming and deep learning in the cloud*,
**https://www.zdnet.com/article/the-future-of-the-future-spark-big-data-insights-streaming-and-deep-learning-in-the-cloud/**

22. *YouTube.* (2014, December 5). *"Knuth–Morris–Pratt algorithm" Online video clip. 4:24. YouTube,*
**https://www.youtube.com/watch?time_continue=43&v=5i7oKodCRJo**

23. *What is Apache Spark,*
**https://www.janbasktraining.com/blog/what-is-spark/**

24. *What is Apache Spark?*,
**https://databricks.com/spark/about**

# ΠΑΡΑΡΤΗΜΑ

Παρακάτω παρατίθενται οι κώδικες που υλοποιήθηκαν

# Longest Common Prefix

## Word by Word Matching

```
# LCP για δύο strings
def lcp(x,y):
.
.
.
    # εάν οι δύο χαρακτήρες είναι ίδιοι προστίθενται στο
    # prefix αλλιώς break
    for j in range(n):
        if x[j]!=y[j]:
            break
        result += x[j]

    return result

def LCP(x):
.
.
.
    # ξεκινώντας από το πρώτο string θα υπολογίζεται κάθε φορά
    # το LCP με το αμέσως επόμενο
    for i in xrange(1,n):
        result=lcp(result,x[i])

    return result
```

## Character by Character Matching

```
def LCP(x):
.
.
.
        # γίνεται έλεγχος του χαρακτήρα σε μια θέση j του πρώτου string
        # με τον αντίστοιχο χαρακτήρα στις ίδιες θέσεις όλων των υπόλοιπων strings
        # αν ειναι ίδιοι είναι μέρος του LCP αλλιώς break
        for j in range(m):
                y=x[0][j]
                for k in range(n):
```

```
                if x[k][j]!=y:
                    break
            else:
                continue
            break

        # το αποτέλεσμα θα είναι το string  από τον πρώτο  χαρακτήρα
        # μέχρι τον χαρακτήρα που προκάλεσε το break
        result += x[0][0:j+1]

        return result
```

## Divide and Conquer

```
def lcp2(x,y):
.
.
.
    # εάν οι δύο χαρακτήρες ειναι ίδιοι προστίθενται στο
    # prefix αλλιώς break
    for j in range(n):
        if x[j]!=y[j]:
            break
        result += x[j]

    return result

def LCP(x,low,high):
.
.
.
    if high>low:
        mid=low+(high-low)/2

        # το string σπάει σε δύο μέρη στα οποία εκτελείται η ίδια
        # διαδικασία κ.ο.κ.
        str1=LCP(x,low,mid)
        str2=LCP(x,mid+1,high)

        # το αποτέλεσμα θα προκύψει απο τις επιμέρους λύσεις
        # των δυο μερων
        result=lcp2(str1,str2)

        return result
```

### Binary Search

```
def LCP(x):
.
.
.
    low=0
    high=m
    mid=low+(high-low)/2

    # έλεγχος στο πρώτο μισό, αν δεν υπάρχει κοινός χαρακτήρας
    # σε όλα τα strings break αλλιώς συνεχίζει τον έλεγχο στο δεύτερο μισό
    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]

            # έλεγχος στο δεύτερο μισό όμως με Character by Character matching
            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
```

## Longest Common Substring

### Naive Search

```
def lcs_naive(x,y):
.
.
.
    # αποθήκευση ολων των substrings του x
```

```
    for i in range(length):
        for j in range(i,length):
            z.append(x[i:j+1])
.
.
.
    lz=len(z)

    # με χρήση του KMP αλγορίθμου γίνεται αναζήτηση
    # για τα κοινά substring με το y και κρατάμε κάθε φορά το μεγαλύτερο
    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
```

## Dynamic Programming

```
def lcs(x,y):
.
.
.
    # δημιουργία πίνακα με διαστάσεις τα μήκη των δύο strings
    L=[[0 for v in range(m)]for w in range(n)]

    # αν δύο χαρακτήρες είναι ίδιοι στην αντίστοιχη θέση του πίνακα
    # θα προστίθεται το ένα με την τιμή του πάνω αριστερά κελιού
    # και αν η τιμή του κελιού είναι μεγαλύτερη από προηγούμενες τιμές
    # κελιών αλλάζουν οι τιμές μήκους, γραμμής και στήλης
    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"

    # ξεκινώντας από την θέση που δείχνουν οι μεταβλητές γραμμής και
    # στήλης διαπερνάμε διαγώνια προς τα πίσω τον πίνακα ώστε να επιστραφεί
    # το αποτέλεσμα
    while L[row][col]!=0:
        result=x[row]+result
        row-=1
```

```
        col-=1

return result
```

## Suffix Array

```
def lcs(x,y):

    # συνδυασμός των δύο strings με χρήση συμβόλων
    # για την δημιουργία ενός νέου
    s=x+"#"+y+"$"
    .
    .
    .
    # δημιουργία διάταξης με όλα τα πιθανά suffixes
    # του string που δημιουργήθηκε
    for i in range(n):
        r.append(s[i:n])

    # λεξικογραφική ταξινόμηση της παραπάνω λίστας
    r=sorted(r)

    # εύρεση του LCP ανά ζεύγος suffixes, το μεγαλύτερο
    # LCP θα είναι το επιθυμητό LCS
    for j in range(n-1):
        l=LCP([r[j],r[j+1]])
        if len(l)>len(lcs):
            lcs=l

    return lcs
```

## Suffix Tree

```
import sys

# η κλάση που αντιπροσωπεύει το suffix tree
class Stree():

    # συνάρτηση δημιουργού
    def __init__(self, input=''):
        .
        .
        .
        if not input == '':
```

```
        self._build_generalized(input)

    # συνάρτηση δημιουργίας suffix tree
    def _build(self, x):
        self.word=x
        self._build_McCreight(x)

    # συνάρτηση δημιουργίας suffix tree με χρήση
    # του O(n) αλγορίθμου McCreight
    def _build_McCreight(self, x):
    .
    .
    .
        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

    # συνάρτηση δημιουργίας κόμβων
    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

    # συνάρτηση δημιουργίας “φύλλων”
    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
.
.
```

.

```
    # συναρτηση δημιουργιας Generalized Suffix Tree (GST) απο
    # το array εισοδου
   def _build_generalized(self, xs):


       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)

    # βοηθητικη μεθοδος που σημειωνει στους κομβους
    # του GST  δεικτες των strings
    def _label_generalized(self, node):

       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

   # βοηθητικη μεθοδος που επιστρεφει τον δεικτη του
   # string με βαση τον κομβο
   def _get_word_start_index(self, idx):
   .
   .
   .
       for _idx in self.word_starts[1:]:
          if idx < _idx:
             return i
          else:
             i+=1
       return i

    # επιστρεφει το LCS του stringIdxs.
    def lcs(self, stringIdxs=-1):

       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]
```

```
    # βοηθητικη μεθοδος που επιστρεφει το LCS διαπερνωντας
    # τους σημειωμενους κομβους του GSD
    def _find_lcs(self, node, stringIdxs):
            nodes=[self._find_lcs(n, stringIdxs)
            for (n,_) in node.transition_links
            if n.generalized_idxs.issuperset(stringIdxs)]
    .
    .
    .
        deepestNode=max(nodes, key=lambda n: n.depth)
        return deepestNode

    # βοηθητικη μεθοδος που επιστρεφει τους αρχικους
    # δεικτες των strings στο GST
    def _generalized_word_starts(self, xs):
        .
        .
        .
        for n in range(len(xs)):
            self.word_starts.append(i)
            i += len(xs[n]) + 1

    #  γεννητρια μοναδικων τερματικων συμβολων για την
    # κατασκευη του GST
    def _terminalSymbolsGenerator(self):
    .
    .
    .
        for i in UPPAs:
            if py2:
                yield(unichr(i))
            else:
                yield(chr(i))
        raise ValueError("Too many input strings.")

# κλάση που αντιπροσωπεύει ένα κόμβο στο suffix tree
class _SNode():
    def __init__(self, idx=-1, parentNode=None, depth=-1):

        # συνδεσμοι
        self._suffix_link=None
        self.transition_links=[]

        # ιδιοτητες
        self.idx=idx
        self.depth=depth
        self.parent=parentNode
        self.generalized_idxs={}
```

# Longest Common Subsequence

## Naive Search

```
def lcs_n(x,y):
.
.
.
    # αν οι τελευταίοι χαρακτήρες των δύο strings είναι ίδιοι
    # ο χαρακτήρας αυτός είναι μέρος του LCSub και η αναζήτηση συνεχίζεται
    # στις αρχικές ακολουθίες μείον αυτόν τον χαρακτήρα
    if x[n-1]==y[m-1]:
        return lcs_n(x[0:n-1],y[0:m-1])+x[n-1]

    # αν οι τελευταίοι χαρακτήρες των δυο strings είναι διαφορετικοί
    # τότε υπολογίζεται το LCSub της x μείον τον τελευταίο της y και της y
    # μείον τον τελευταίο του x
    l1=lcs_n(x,y[0:m-1])
    l2=lcs_n(x[0:n-1],y)

    # το μεγαλύτερο από τα δύο παραπάνω LCSub
    # είναι το τελικό LCSub
    return max(l1,l2)
```

## Dynamic Programming

```
def lcs(x,y):
.
.
.
    # δημιουργία ενός πίνακα με διαστάσεις τα μήκη των δύο strings
    # αυξημένα κατά 1
    L=[[0 for v in range(m+1)]for w in range(n+1)]

    for i in range(n+1):
        for j in range(m+1):
            # η πρώτη γραμμή και πρώτη στήλη θα έχουν
            # μόνο μηδενικά
            .
            .
            .
            # αν οι χαρακτήρες ενός κελιού είναι ίδιοι
            # στο δεξιό κάτω διαγώνιο από αυτό κελί μπαίνει η τιμή
            # του κελιού αυξημένη κατά 1
            elif x[i-1]==y[j-1]:
```

```
                L[i][j]=L[i-1][j-1]+1

            # αν είναι διαφορετικοί επιλέγεται κατεύθυνση στον
            # πίνακα με βάση το αποτέλεσμα της παρακάτω σύγκρισης
            else:
                L[i][j]=max(L[i-1][j], L[i][j-1])
.
.
.
    # ακολουθώντας την διαδρομή του αλγορίθμου προς τα πίσω
    # προκύπτει το LCSub αντεστραμμένο
    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

    # αντιστρέφοντας το προηγούμενο αποτέλεσμα προκύπτει
    # το τελικό LCSub
    return "".join(reversed(lcs))
```

## Longest Increasing Subsequence

```
def lis(x):
.
.
.
    y[0]=x[0]
    for i in range(n-1):
        k=len(y)

        # αν το δεύτερο στοιχείο είναι μικρότερο του πρώτου
        # μπαίνουν και τα δύο στον πίνακα l
        if len(y)==1 and x[i+1]<y[0]:
            l.extend([y[0],x[i+1]])
            y[0]=x[i+1]

        # αν ένα στοιχείο είναι μεταλύτερο από το τελευταίο
        # του πίνακα y γίνεται νέο στοιχείο αυτού του πίνακα
        # ενώ προστίθεται και στον πίνακα l μαζί με το στοιχείο
        # που αντικατέστησε
        elif x[i+1]>y[k-1]:
            y.extend([x[i+1]])
```

```
            l.extend([y[k-1],x[i+1]])

        # αν ένα στοιχείο είναι μικρότερο του τελευταίου στοιχείου
        # του πίνακα y εντοπίζουμε το αμέσως μεγαλύτερο του
        # το οποίο αντικαθιστά
        elif x[i+1]<y[k-1] and len(y)>1:
            for j in range(k):
                if x[i+1]<y[j]:
                    if j==0:
                        l.extend([y[j],x[i+1]])
                    else:
                        l.extend([y[j-1],x[i+1]])
                    y[j]=x[i+1]
                    break
                elif j==k-1:
                    l.extend([y[0],x[i+1]])
                    y[0]=x[i+1]
    .
    .
    .
    # διαπερνώντας προς τα πίσω τον πίνακα l στον οποίο
    # έχουμε αποθηκεύσει τις τιμές που αντικαταστάθηκαν
    # από άλλες τιμές βρίσκουμε το μήκος της LIS καθώς και
    # την ίδια την ακολουθία
    for g in range(len(y)-1):
        for f in range(len(l),0,-1):
            if f%2==1 and l[f]==s[0]:
                s.insert(0,l[f-1])
                break

    print "Mikos LIS:",len(y),"\n","LIS:",s

def lcsub(x,y):
.
.
.
    # δημιουργούνται οι ακολουθίες p(A), p(C), p(G), p(T)
    # οι οποίες περιέχουν τις θέσεις εμφάνισης στο y των χαρακτήρων
    # του x δηλαδή των A, C, G, T  αντίστοιχα σε φθίνουσα διάταξη
    for i in range(m-1,-1,-1):
        if y[i]=="A":
            pA.append(str(i+1))
        elif y[i]=="C":
            pC.append(str(i+1))
        elif y[i]=="G":
            pG.append(str(i+1))
        elif y[i]=="T":
            pT.append(str(i+1))

    # δημιουργείται η ακολουθία s που είναι η ακολουθία
```

```
# θέσεων του y αν ενωθούν οι αντίστοιχες ακολουθίες
# που δημιουργήθηκαν πριν
for j in range(n):
    if x[j]=="A":
        s.append(''.join(pA))
    elif x[j]=="C":
        s.append(''.join(pC))
    elif x[j]=="G":
        s.append(''.join(pG))
    elif x[j]=="T":
        s.append(''.join(pT))
.
.
.
# αντικαθιστούμε κάθε στοιχείο της ακολουθίας s
# με ένα ζευγάρι τιμών, που είναι το στοιχείο με τον
# αριθμό εμφάνισης του στο s
for w in range(len(s)):
    p=0
    for g in range(w,len(s)):
        if s[w]==s[g]:
            p+=1
        if g==len(s)-1:
            f.append([int(s[w]),p])

f2=sorted(f)

# σε κάθε ζευγάρι τιμών αποδίδεται ο βαθμός του
# με βάση την λεξικογραφική σειρά της ακολουθίας
for u in range(len(f2),0,-1):
    f2[u-1].append(u)

# αντικατάσταση των ζευγαριών τιμων
# με τους αντίστοιχους βαθμούς τους
for z in range(len(f)):
    f[z]=f[z][2]

# επιστρέφεται η LIS της ακολουθίας f
# που δημιουργήθηκε που είναι το
# επιθυμητό LCS
return lis(f)
```

# Longest Common Extension

## Naive Search

```
def LCE(x,i,j):
.
.
.
# ξεκινώντας από τις δοσμένες θέσεις όσο
# οι χαρακτήρες είναι ίδιοι θα ελέγχεται ο επόμενος
# στην ίδια θέση και για τα  δύο strings
# όταν βρεθεί κάποιος διαφορετικός για τα
# δύο strings τότε break
while x[i+length]==x[j+length]:
        length+=1
        if j+length==n:
                break

return length+1
```

## Naive Search with k-Mismatches

```
def LCE_k(x,i,j,k):
.
.
.
    # όσο οι χαρακτήρες των δύο strings ξεκινώντας
    # από τις δοσμένες θέσεις είναι ίδιοι θα ελέγχονται
    # οι επόμενοι
    while True:
        if x[i+length]==x[j+length]:
                length+=1

        # αν είναι διαφορετικοί τότε αυξάνεται μια μεταβλητή
        # αν φτάσει την τιμή που εχει καθοριστεί break
        else:
            p+=1
            if p==k+1:
                break
            length+=1

        if j+length==n:
            break
```

```
    return length
```

## k-Closed Border

```
import numpy as np

# η συνάρτηση αυτή δέχεται ως είσοδο ένα string
# και επιστρέφει το αντιστραμμένο
def Reverse(x):
    x_l=list(x)
    x_l.reverse()
    x_rl=x_l
    x_r=''.join(x_rl)

    return x_r

# συνάρτηση που για κάθε θέση
def LPM(x,k):
.
.
.
    # δημιουργείται και αρχικοποιείται ένας  μονοδιάστατος
    # πίνακας με την τιμή -1 σε όλες τις θέσεις του
    lpm=np.arange(l)
    for i in range(l):
        lpm[i]=-1

    # οι τιμές του πίνακα προκύπτουν υπολογίζοντας κάθε
    # φορά το LCE_k με είσοδο τις θέσεις του x μεγαλύτερες του 0
    # και το k
    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):
.
.
.
    # δημιουργείται και αρχικοποιείται ένας  μονοδιάστατος
    # πίνακας με την τιμή -1 σε όλες τις θέσεις του
    lsm=np.arange(l)
    for i in range(l):
        lsm[i]=-1
```

```
    # αντιστρέφεται το x με χρήση της συνάρτησης Reverse
    x=Reverse(x)

    # οι τιμες του πινακα προκύπτουν υπολογίζοντας κάθε
    # φορά το LCE_k με είσοδο τις θέσεις του x μικρότερες
    # της τελευταίας από την μεγαλύτερη προς την μικρότερη
    # και το k
    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

# η βασική συνάρτηση
def getBorder(x,k):
.
.
.
    # δημιουργία των τριών πινάκων που θα περιέχουν
    # τις τιμές για τις τρεις συνθήκες
    con1=np.arange(n)
    con2=np.arange(n)
    con3=np.arange(n)
    con=np.arange(n)
.
.
.
    # δημιουργία δύο πινάκων με χρήση των
    # συναρτήσεων LPM και LSM
    lpm=LPM(x,k)
    lsm=LSM(x,k)

    for j in range(n):
        .
        .
        .
        # έλεγχος για την πρώτη συνθήκη
        if j+lpm[j]==n:
            con1[j]=1

        # έλεγχος για τη δεύτερη συνθήκη
        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
```

```
  # έλεγχος για την τρίτη συνθήκη
  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

# το s για το οποίο η τιμή του πίνακα con θα
# είναι 1, είναι η θέση του k-Closed Border του x
for s in range(n-1):
  con[s]=con1[s]*con2[s]*con3[s]
  if con[s]==1:
      thesi=s
      mikos=n-thesi

if thesi=="":
    print "Δεν υπάρχει",k,"-closed Border"
else:
    print “Υπάρχει”,k,"-Closed Border στη θέση",thesi,"με μήκος",mikos
```