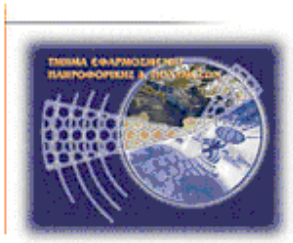
Τεχνολογικό Εκπαιδευτικό Ίδρυμα Κρήτης

Σχολή Τεχνολογικών Εφαρμογών
Τμήμα Εφαρμοσμένης Πληροφορικής & Πολυμέσων



ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Τίτλος : Αλγόριθμοι για επεξεργασία δυναμικών ροών δεδομένων (data streams)

ΣΑΝΔΑΛΑΚΗΣ ΒΑΣΙΛΕΙΟΣ (ΑΜ:2432)

Επιβλέπων καθηγητής: Παπαδάκης Νικόλαος

Ευχαριστίες

Με την λήξη αυτού του ακαδημαϊκού κύκλου, θεωρώ χρέος να ευχαριστήσω όλους όσους με στήριξαν κατά τη διάρκεια της εκπόνησης της πτυχιακής εργασίας μου.

Οι θερμές ευχαριστίες απευθύνονται προς τον επιβλέποντα καθηγητή κ. Νικόλαο Παπαδάκη για την συμπαράσταση και τις κατευθύνσεις που έδωσε οι οποίες έφεραν σε πέρας την πτυχιακή αυτή, καθώς και τον κ. Ιωάννη Ξεζωνάκη για τις γνώσεις του οι οποίες μεταδώθηκαν άριστα κατά την διάρκεια των διαλέξεων.

Θα ήθελα παράλληλα να ευχαριστήσω όλους τους καθηγητές του τμήματος Εφαρμοσμένης Πληροφορικής & Πολυμέσων, που με την μεθοδικότητά τους συντέλεσαν στο μέγιστο στη διαδικασία της μάθησης.

Abstract

The purpose of this work is the study of data streams and that can be combined (couplers / join) between them. These flows can be reported for example in on-line auctions where a flow is the auctioned items and the second bids. Because the memory capacity is not unlimited, the various streams should be stored in the queue.

In this paper we studied various replacement algorithms to achieve the maximum number of combinations. We performed extensive simulations by changing the size of the queues and the size of flows.

Σύνοψη

Σκοπός της εργασίας αυτής είναι η μελέτη ροών δεδομένων (data streams) και πως μπορούν να συνδιαστούν (συζευκτούν/συνενωθούν) μεταξύ τους. Οι ροές αυτές μπορεί να αναφέρονται για παράδειγμα σε on-line δημοπρασίες όπου η μία ροή είναι τα δημοπρατούμενα αντικείμενα και η δεύτερη οι προσφορές. Επειδή η χωρητικότητα στη μνήμη δεν είναι απεριόριστη, οι διάφορες ροές πρέπει να αποθηκεύονται σε ουρές.

Στην εργασία αυτή μελετήσαμε διάφορους αλγόριθμους αντικατάστασης έτσι ώστε να επιτευχθεί ο μέγιστος αριθμός συνενώσεων. Πραγματοποιήσαμε εκτενής προσομοιώσεις αλλάζοντας το μέγεθος των ουρών και το μέγεθος των ροών.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Η πτυχιακή εργασία (ΠΕ) είναι ένα από τα ποιο σημαντικά μέρη του πτυχίου. Ο φοιτητής δραστηριοποιείται σε μία «προσομοίωση» ενος πραγματικού εργασιακού περιβάλλοντος καθώς έχει την καθοδήγηση του επιβλέπων καθηγητή. Ο σπουδαστής οφείλει να εκμεταλλευτεί όσον το δυνατόν περισσότερο την ευκαιρία αυτή προκειμένου να εφοδιαστεί με την κατάλληλη εμπειρία που θα έχει ως γνώμονα στην εύρεση εργασίας.

1.1 Περίληψη

Ο σκοπός της πτυχιακής αυτής είναι η μελέτη, σχεδίαση και υλοποίηση διαφόρων προγραμμάτων για την εύρεση ενός βέλτιστου αλγορίθμου. Καθώς έρχονται δύο διαφορετικές ροές τυχαίων αριθμών, ο αλγόριθμος αυτός αναλαμβάνει να επιτύχει όσες περισσότερες συνδεσμολογίες (***join***) μπορεί ανάμεσα σε αυτες. Αυτό επιτυγχάνεται με την εύρεση του τυχαίου αριθμού που θα πρέπει να αποχωρήσει από την αποθηκευμένη ροή ουτως ώστε η νέα ροή να επιτυγχάνει περισσότερες συνδεσμολογίες.

Η τεχνολογία που χρησιμοποιήθηκε είναι η **C++** με την βοήθεια του προγράμματος *Dev C++*.

Έτσι το βέλτιστο πρόγραμμα, το οποίο επιλέχτηκε ανάμεσα απο άλλα σύμφωνα με μετρήσεις, διευκολύνει την εύρεση δύο διαφορετικών ροών οι οποίες έχουν τις περισσότερες δυνατές συνδεσμολογίες μεταξύ τους.

1.2 Κίνητρο για την Διεξαγωγή της Εργασίας

Στις μέρες μας όλα γίνονται μέσω διαδικτύου, γι'αυτό και δημιουργήθηκε η ανάγκη για on-line δημοπρασίες, οι οποίες επιτυγχάνονται με την χρήση διαφόρων αλγορίθμων αντικατάστασης.

1.3 Σκοπός και Στόχοι της Εργασίας

Σκοπός της εργασίας αυτής είναι η μελέτη ροών δεδομένων (data streams) και πως μπορούν να συνδιαστούν μεταξύ τους.

Στόχος μας είναι να δημιουργήσουμε ένα πρόγραμμα που θα δέχεται δύο ροές τυχαίων ακέραιων αριθμών και θα αναλαμβάνει να πετύχει όσες περισσότερες διασυνδέσεις γίνονται, ανάμεσα σε αυτες. Επίσης, αποθηκεύεται ένα μικρό σε αριθμό μέγεθος σε σχέση με το μέγεθος της ροής, με αποτέλεσμα να απαιτεί διερεύνηση για το ποιος κατωχυρομένος αριθμός πρέπει να αποχωρήσει για να εισαχθεί ο νέος. Με βάση αυτόν τον προβληματισμό θα δούμε παρακάτω 4 διαφορετικούς αλγόριθμους, που διαφέρουν, δηλαδή, στην αντιμετώπιση του προβλήματος αυτού. Οι αλγόριθμοι είναι οι εξής: **LFU** (Less Frequency Used), **TimeOfJoin, LRU** (Less Frequency Used) και **FIFO** (First In First Out).

1.4 Δομή της Εργασίας

Στο 1^{ο} κεφάλαιο προσπαθούμε να εισάγουμε τον αναγνώστη για τους σκοπούς και τους στόχους της υλοποίησης αυτής της εφαρμογής. Στο 2^{ο} κεφάλαιο αναφερόμαστε στο θεωρητικό υπόβαθρο που έχουμε για την υλοποίηση αυτή. Στο 3^{ο} κεφάλαιο αναλύουμε θεωρητικά τα μοντέλα που υλοποιήθηκαν, και για ποιό λόγο είναι προτιμότερα, καθώς και συμπεράσματα και μελλοντική δουλειά. Στο 4^{ο} κεφάλαιο αναλύουμε και επεξηγούμε τους τέσσερις αλγορίθμους. Στο 5^{ο} κεφάλαιο ασχολούμαστε με τα αποτελέσματα της εφαρμογής, τις μετρήσεις και τις γραφικές παραστάσεις. Τέλος, στο παράρτημα Α έχουμε τους τέσσερις αλγορίθμους.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Θεωρούμε ότι το πρόβλημα της προσέγγισης του συρόμενου παραθύρου εντάσσεται σε ροές δεδομένων με ένα σύστημα επεξεργασίας ροής δεδομένων με περιορισμένους πόρους. Στο μοντέλο μας, έχουμε να κάνουμε με περιορισμούς πόρων απορρίπτοντας φορτίο με τη μορφή της εγκατάληψης πλειάδων από τις ροές δεδομένων. Πρώτα συζητάμε για εναλλακτικά αρχιτεκτονικά μοντέλα για την επεξεργασία συνένωσης ροών δεδομένων και ερευνούμε τα κατάλληλα μέτρα για την ποιότητα μιας προσέγγισης set-value αποτελέσματος του ερωτήματος (που αποτιμάται σαν σύνολο). Μετά θεωρούμε τον αριθμό πλειάδων αποτελεσμάτων που δημιουργείται σαν ποιοτική μέτρηση και δίνουμε βέλτιστη σύνδεση και γρήγορη σύνδεση αλγορίθμων για αυτό. Σε μια πλήρη πειραματική μελέτη με συνθετικά και πραγματικά δεδομένα δείχνουμε την αποτελεσματικότητα των λύσεών μας. Για εφαρμογές με απαίτηση ακριβών αποτελεσμάτων εισάγουμε ένα καινούργιο Αρχείο – μετρικών το οποίο καταγράφει τον όγκο επεξεργασίας που απαιτείται για την ολοκλήρωση της συνένωσης σε περίπτωση που οι ροές αρχειοθετούνται για μελλοντική επεξεργασία.

Σε πολλές εφαρμογές απο την διαχείρηση δικτύου IP σε τηλεφωνική ανίχνευση απάτης, τα δεδομένα φτάνουν με ροές υψηλών ταχυτήτων και τα ερωτήματα πάνω σε αυτές τις ροές πρέπει να υποστούν επεξεργασία σε ένα online τρόπο που παρέχει απαντήσεις σε πραγματικό χρόνο. Οι ροές δεδομένων θέτουν μια σοβαρή πρόκληση για συστήματα διαχείρησης δεδομένων καθώς το παραδοσιακό μοντέλο DBMS της προσανατολισμένης στο σύνολο επεξεργασίας πλειάδων στο δίσκο δεν ισχύει. Πρόσφατα καινούργιες προτάσεις για συστήματα επεξεργασίας ροής δεδομένων έχουν προκύψει. Αυτά τα συστήματα θεσπίζονται ειδικά για την επεξεργασία δεδομένων σε πραγματικό χρόνο.

Όσον αφορά τα συστήματα σχεσιακών βάσεων δεδομένων, ο χειριστής συνενώσεων είναι ένας πολύ σπουδαίος χειριστής σε ένα σύστημα επεξεργασίας ροής δεδομένων. Ας θεωρήσουμε ότι R και S είναι δύο ροές δεδομένων που περιέχουν ένα κοινό χαρακτηριστικό A, το οποίο επιλέγεται σαν το χαρακτηριστικό συνένωσης. Η ισο-συνένωση (equi-join) των R και S είναι το υποσύνολο πολλαπλών προϊόντων των δύο ροών το οποίο περιέχει ακριβώς αυτα τα ζεύγη των πλειάδων (r , s) έτσι ώστε rER , sES , και r.A = s.A.

Ενώ οι διασυνδέσεις είναι πολύ σημαντικές, ο υπολογισμός τους είναι πηγή έντασης. Για παράδειγμα, μια τυπική ισο-συνένωση μεταφέρει εννοιολογικά απέραντη κατάσταση για δύο άπειρες ροές εισόδου. Για την αντιμετώπιση του προβλήματος της απέραντης κατάστασης

για συνενώσεις ροής δεδομένων, η σημασιολογία της συνένωσης συνήθως αλλάζει για να περιορίσει το σύνολο των πλειάδων που συμμετέχουν στην συνένωση σε ένα οριοθετημένου-μεγέθους παράθυρο των πιο πρόσφατων πλειάδων. Δεδομένου του ότι το παράθυρο εννοιολογικά μόλις που θίγει τις ροές εισόδου, αυτός ο τύπος συνένωσης συχνά ονομάζεται συρόμενο παράθυρο συνένωσης. Παρατηρήστε ότι υπάρχουν κάποιες πιθανότητες να ορίσουμε τα όρια του παραθύρου, βασιζόμενοι σε μονάδες χρόνου, αριθμό πλειάδων ή ορόσημα. Για απλούστευση θα υποθέσουμε ότι το παράθυρο ορίζεται με βάση των μονάδων χρόνου (χρόνος ρολογιού τοίχου) και ότι σε κάθε μονάδα χρόνου μια καινούργια πλειάδα φθάνει σε κάθε ροή εισόδου. Η συζήτηση μας και οι τεχνικές μπορούν να γενικευτούν σε παράθυρα που ορίζονται με βάση τον αριθμό των πλειάδων και την ασύγχρονη άφιξη πλειάδας.

Στα επόμενα θα χρησιμοποιήσουμε w για να χαρακτηρίσουμε το μέγεθος του παραθύρου. Ας θεωρήσουμε ότι r(i) αναφέρεται στην πλειάδα ροής R που φτάνει στον χρόνο i. Για απλούστευση θα χρησιμοποιήσουμε r(i) για να χαρακτηρίσουμε την αξία του χαρακτηριστικού συνένωσης της πλειάδας (s(i) ορίζεται και χρησιμοποιείται ομοίως). Σύμφωνα με το μοντέλο μας, σε κάθε χρονική στιγμή t το ολισθένον παράθυρο περιέχει όλες τις πλειάδες r(i) και s(i) με t-w < i <= t.

Η σε απευθείας σύνδεση φύση των ροών δεδομένων και τα ενδεχομένως υψηλά ποσοστά άφιξης επιβάλλουν υψηλές απαιτήσεις πόρων πάνω στα συστήματα επεξεργασίας ροής δεδομένων. Ειδικά σε εφαρμογές όπου διάφορα ερωτήματα υποβάλλονται σε επεξεργασία ταυτόχρονα, η διαθεσιμότητα των πόρων που μπορεί να αφιερωθεί σε κάθε ερώτημα είναι περιορισμένη και μπορεί να μεταβάλλεται με την πάροδο του χρόνου. Ως άλλο παράδειγμα, είναι συχνά αδύνατον να υπολογίσουμε το μέγιστο ποσοστό άφιξης πλειάδας για ροές δεδομένων και ως εκ τούτου η διαστασιολόγηση ενός συστήματος ροής δεδομένων για φορτία αιχμής είναι ένα δύσκολο πρόβλημα. Έτσι, αν και η αλλαγή της σημασιολογίας του χειριστή συνένωσης στα συρόμενα παράθυρα έχει ήδη μειώσει τις απαιτήσεις των πόρων του χειριστή συνένωσης (φορέα συνένωσης) υπολογίζοντας ότι οι συνενώσεις του συρόμενου παραθύρου μπορούν να υπερβαίνουν την διαθεσιμότητα των πόρων.

Οι περιορισμοί πόρων μπορούν να έχουν δυο συνέπειες. Για ροές με υψηλά ποσοστά άφιξης, η CPU μπορεί να μην είναι αρκετά γρήγορη στο να επεξεργάζεται όλες τις εισερχόμενες πλειάδες σε εύθετο χρόνο, δηλαδή έχουμε μια αργή CPU σε σύγκριση με το ποσοστό άφιξης της ροής. Για τα μεγάλα παράθυρο w η διαθέσημη κύρια μνήμη M μπορεί να είναι πάρα πολύ μικρή για να κρατήσει όλες τις σχετικές πλειάδες στη μνήμη (και η συχνή πρόσβαση στο σκληρό δίσκο θα είναι πολύ αργή όταν τα ποσοστά άφιξης είναι υψηλά).

Για την αντιμετώπιση των περιορισμών των πόρων κατα ένα κομψό τρόπο, το να επιστρέφεις κατά προσέγγιση απαντήσεις των ερωτημάτων αντί για ακριβείς απαντήσεις έχει

αναδειχτεί σαν μια πολλά υποσχόμενη προσέγγιση για να σώσει πηγές. Σε συστήματα επεξεργασίας ροής δεδομένων, ένας τρόπος προσέγγισης απαντήσεων στα ερωτήματα είναι να ρίξει φορτίο, για παράδειγμα, ρίχνοντας πλειάδες πριν φυσικά λήξουν (δηλαδή, αφήστε το παράθυρο) ή ακόμα και πριν φτάσουν στον χειριστή (φορέα). Η τρέχουσα κατάσταση της τέχνης αποτελείται από δύο βασικές προσεγγίσεις. Η πρώτη βασίζεται σε τυχαία απόρριψη φορτίου,δηλαδή αφαιρούνται πλειάδες με βάση τα ποσοστά άφιξης αλλά όχι οι πραγματικές τιμές τους. Η δεύτερη προτείνει να συμπεριληφθούν QoS προδιαγραφές οι οποίες αναθέτουν προτεραιότητες στις πλειάδες και στη συνέχεια να ρίξουν αυτές με χαμηλή προτεραιότητα πρώτα. Ωστόσο, το αποτέλεσμα μιας συνένωσης αποτελείται απο ζεύγη ταιριαστών πλειάδων, ως εκ τούτου και τα δύο, και το χαρακτηριστικό της συνένωσης μιας πλειάδας και ο αριθμός των πλειάδων εταίρους της (δηλαδή εκείνες που ταιριάζουν στην πλειάδα) στην άλλη ροή, καθορίζουν την απόδοση. Για το λόγο αυτό και η τυχαία απόρριψη φορτίου και οι απλές QoS μεταβιβάσεις σε ενιαίες πλειάδες δεν αντικατοπτρίζουν πλήρως τη σημασιολογία της συνένωσης.

Για παράδειγμα, είναι γνωστό ότι η τυχαία δειγματοληψία από τις εισόδους R και S μιας συνένωσης, ή η προκατειλημμένη δειγματοληψία από R και S χωρίς να λαμβάνεται υπ΄όψιν η διανομή της άλλης σχέσης, μπορεί πολύ να παραποιήσει την έξοδο της συνένωσης και να οδηγήσει στη χειρότερη περίπτωση σε μια άδεια έξοδο συνένωσης ακόμα και αν το πραγματικό μέγεθος της συνένωσης είναι πολύ μεγάλο.

Σημασιολογική Περικοπή Φορτίου. Στην εργασία αυτή, απευθυνόμαστε στα προβλήματα που περιγράφονται παραπάνω με την εισαγωγή της έννοιας της σημασιολογικής περικοπής φορτίου.Στην σημασιολογική περικοπή φορτίου, προσεγγίζουμε την απόδοση ενός φορέα (χειριστή) με το να μεγιστοποιούμε του προσδιοριζόμενου χρήστη το μέτρο ομοιότητας ανάμεσα στην ακριβή απάντηση και στην (κατα προσέγγιση) απάντηση που επιστρέφεται από το σύστημα. Η σημασιολογική περικοπή φορτίου αποφεύγει τα παραπάνω προβλήματα με το να επιλέγει έξυπνα ποίες πλειάδες να ρίξει και πότε θα πρέπει να μειωθούν – όλα προκειμένου να ελαχιστοποιήσει το σφάλμα της απόδοσης του ερωτήματος. Αυτή η εργασία περιέχει μια εις βάθος μελέτη αυτού του προβλήματος για την περίπτωση της συνένωσης του συρόμενου παραθύρου. Ας συζητήσουμε εν συντομία κάποια σχετικά σενάρια όπου η σημασιολογική περικοπή φορτίου μπορεί να οδηγήσει σε μεγάλες βελτιώσεις.

Στατική συνένωση: Θεωρείστε ένα δίκτυο με αισθητήρες μικρών μπαταριών ισχύος με περιορισμένη ταχύτητα της CPU και μνήμη οι οποίες μετρούν περιβαλλοντικά δεδομένα. Επιπλέον, υπάρχουν πληρεξούσια αισθητήρων στο δίκτυο που δεν έχουν περιορισμούς ισχύος και έχουν αρκετά CPU και πόρους μνήμης. Ο σκοπός των πληρεξούσιων είναι να συλλέγουν δεδομένα αισθητήρων και να εκτελούν ερωτήματα παρεχόμενα από τον χρήστη, για παράδειγμα μια συνένωση πάνω από ένα χαρακτηριστικό των μετρημένων πλειάδων

δεδομένων. Για να υπολογίσουμε ότι η συνένωση για ένα δεδομένο χρονικό διάστημα, το πληρεξούσιο χρειάζεται να θέσει υπο αμφισβήτηση τους αισθητήρες για τα δεδομένα τους. Δεδομένου ότι η διαβίβαση των στοιχείων είναι πολύ δαπανηρή από την άποψη της ισχύος της μπαταρίας του αισθητήρα, ο στόχος του συστήματος είναι να διαβιβάσει όσο το δυνατόν λιγότερα δεδομένα για να επεκτείνει την διάρκεια ζωής των αισθητήρων. Ως εκ τούτου, έχουμε ένα πρόβλημα βελτιστοποίησης για να επιλέξουμε τα σωστά δεδομένα προς μετάδοση έτσι ώστε το σφάλμα προσέγγισης του αποτελέσματος ελαχιστοποιημένο υπόκειται στους περιορισμους κατανάλωσης ενέργειας (το οποίο είναι ισοδύναμο προς τους περιορισμούς μετάδοσης δεδομένων).

Συνένωση με υποστήριξη Αρχείου: Οι ροές δεν φτάνουν πάντα με ένα σταθερά υψηλό ρυθμό και δεν είναι πάντα επιθυμητό να αποκτήσεις αποτελέσματα περιορισμένης ακρίβειας. Ένα παράδειγμα είναι εφαρμογές λιανικής όπου η δραστηριότητα πωλήσεων είναι πολύ μεγαλύτερη κατα τη διάρκεια της ημέρας από ότι κατά τη διάρκεια της νύχτας. Επίσης, η ανάλυση των επιχειρήσεων τυπικά είναι πιο ενεργή και ως εκ τούτου τα εισερχόμενα δεδομένα είναι αρχειοθετημένα για μελλοντική ανάλυση. Σε άλλες εφαρμογές, π.χ., η παρακολούθηση εφυίας και καταστροφών, η απόρριψη φορτίου απλά δεν είναι αποδεκτή λόγω του κινδύνου "να χάσεις τη βελόνα στα άχυρα". Σε τέτοια σενάρια το σύστημα επεξεργασίας ροής θα λειτουργούσε σε συνεργασία με ένα αρχείο, π.χ. μια αποθήκη δεδομένων. Σε ημερήσιο (δηλαδή, φορτίο αιχμής) mode θα παράγει κατά προσέγγιση αποτελέσματα για τα εισερχόμενα δεδομένα σε πραγματικό χρόνο. Σε βραδυνό (δηλαδή, χαμηλό φορτίο) mode όταν ο ρυθμός άφιξης των καινούριων πλειάδων είναι χαμηλός, τα δεδομένα από το αρχείο διαβάζονται και γίνεται επεξεργασία προκειμένου να βελτιωθούν πρώιμα κατά προσέγγιση αποτελέσματα. Για τον χειρηστή συνένωσης αυτό σημαίνει ότι κατά τη διάρκεια της ημέρας μόνο ένα υποσύνολο του συνολικού αποτελέσματος συνένωσης υπολογίζεται. Όλες οι πλειάδες που δεν υποβάλλονται σε επεξεργασία πλήρως στη συνέχεια υποβάλλονται σε επεξεργασία το βράδυ με το να υπάρχει πρόσβαση στο αρχείο. Παρατηρήστε ότι τώρα το φορτίο δεν αποβάλλεται πράγματι, αλλά μάλλον ανατίθενται. Ως εκ τούτου θα χρησιμοποιήσουμε τον όρο σημασιολογική εξομάλυνση φορτίου.

Συνεισφορές αυτής της εργασίας: Σε αυτήν την εργασία δίνουμε μια σε βάθος εξέταση της σημασιολογικής απόρριψης φορτίου για την συνένωση σε παράθυρο ροής δεδομένων. Παρουσιάζουμε αλγόριθμους καινοφανείς για προσέγγιση set-value αποτελεσμάτων συνένωσης σε διακριτότητα πλειάδας. Οι αλγόριθμοι μας επιτυγχάνουν το καλύτερο δυνατό αποτέλεσμα κατά προσέγγιση σύμφωνα με μιά δεδομένη μέτρηση σφάλματος που υπόκειται σε δεδομένους περιορισμούς πόρων. Συγκεκριμένα, κάνουμε τις ακόλουθες συνεισφορές:

- Σκιαγραφούμε το χώρο του σχεδιασμού των πιθανών μέτρων λάθους και εισάγουμε ένα καινούριο μετρικό αρχείου για την συνένωση του συρόμενου παραθύρου με την

υποστήριξη αρχείου. Περιγράφουμε αρχιτεκτονικά μοντέλα για προσέγγιση συνένωσης σε συρόμενο παράθυρο ροής δεδομένων.

- Στη συνέχεια παρουσιάζουμε αποτελέσματα για μια επιλεγμένη μέτρηση λάθους – την MAX μέτρηση υποσυνόλου, η οποία μεγιστοποιεί τον αριθμό των πλειάδων στην κατά προσέγγιση απόδοση της συνένωσης. Πιο συγκεκριμένα, παρουσιάζουμε αποτελέσματα σκληρότητας και αλγόριθμους για την περίπτωση στατικής συνένωσης και βέλτιστους αλγόριθμους εκτός σύνδεσης και πολύ γρήγορη και ελαφριά ευρεστική σύνδεση για το πρόβλημα της συνένωσης σε συρόμενο παράθυρο.

- Αξιολογούμε τους αλγόριθμους σε ένα μεγάλο σύνολο συνθετικών και πραγματικής ζωής δεδομένων.

Μια συζήτηση σχετικής δουλειάς και μια σύνοψη και προοπτικές για μελλοντική δουλειά ολοκληρώνουν αυτό το άρθρο.

Κεφάλαιο 3

Μοντέλα και μέτρα – Συμπέρασμα

3.1 Εισαγωγή

Σε αυτήν την ενότητα ορίζουμε το χώρο του προβλήματος. Ειδικότερα εισάγουμε διάφορα μοντέλα για την προσέγγιση του προβλήματος υπολογισμού συνένωσης και συζητάμε για μέτρα αξιολόγησης της ποιότητας ενός κατά προσέγγιση αποτελέσματος συνένωσης.

3.2 Μοντέλα

Στόχος μας είναι να επεξεργαστούμε ένα συρόμενο παράθυρο ισο-συνένωσης ανάμεσα σε δύο ροές δεδομένων R και S. Ο χειριστής συνένωσης διαθέτει μνήμη για την διατήρηση εσωτερικής κατάστασης, δηλαδή, εκείνες οι πλειάδες που βρίσκονται εντός του τρέχοντος παραθύρου. Νεοαφιχθέντες πλειάδες ενώνονται με όλες τις αντίστοιχες πλειάδες της άλλης ροής στη μνήμη συνένωσης. Προκειμένου να διασφαλιστεί ο υπολογισμός του ακριβούς αποτελέσματος το σύστημα σε οποιαδήποτε χρονική στιγμή πρέπει να δεκτεί τις 2 w πλειάδες του τρέχοντος παραθύρου και πρέπει να επεξεργαστεί τις εισερχόμενες πλειάδες τουλάχιστον τόσο γρήγορα όσο την άφιξή τους. Αν πληρούνται αυτές οι προϋποθέσεις, μια εισερχόμενη πλειάδα θα παραμένει στη μνήμη μέχρι τη λήξη, δηλαδή, θα περάσει όλη τη ζωή της (w μονάδες χρόνου) στη μνήμη και θα δημιουργήσει απόδοση με όλες τις αντίστοιχες πλειάδες στην άλλη ροή. Ωστόσο σε περίπτωση έλλειψης πόρων, οι πλειάδες πρέπει να απορρίπτονται πριν λήξουν.

Modular vs. Integrated. Εντοπίζουμε δύο γενικά μοντέλα για επεξεργασία συνένωσης σε παράθυρο – modular και integrated. Και στις δύο περιπτώσεις υπάρχει μνήμη συνένωσης μεγέθους M για τις πλειάδες στο τρέχων παράθυρο και μια ουρά για καινούριες αφίξεις πλειάδων ροής. Για να πάρουμε μελετημένες αποφάσεις σχετικά με ποιές πλειάδες να εκδιώξουμε από την συνένωση ή την ουρά σε περίπτωση ελλείψεων πόρων μια ενότητα

στατιστικών συγκεντρώνει πληροφορίες σχετικά με την κατανομή της πλειάδας. Η κύρια διαφορά μεταξύ των δύο μοντέλων έγκειται στο βαθμό ολοκλήρωσης ανάμεσα στα συστατικά.

Στην περίπτωση που η αρθρωτή ουρά έχει μόνο περιορισμένη γνώση σχετικά με τα περιεχόμενα μνήμης συνένωσης (για παράδειγμα, μόνο ένα ιστόγραμμα για τις συχνότητες της χαρακτηριστικής ιδιότητας επιπέδου αξιών συνένωσης στη μνήμη) και το αντίστροφο. Κάθε ενότητα χρησιμοποιεί τη δική της τακτική για να αποφασισθεί ποιές πλειάδες να απορριφθούν σε περίπτωση έλλειψης πόρων. Αυτές οι αποφάσεις επιρεάζουν μόνο από την είσοδο από την μονάδα στατιστικών. Η αρθρωτή (modular) προσφέρει το πλεονέκτημα ότι διάφοροι χειριστές μπορούν να σχεδιαστούν και να εφαρμοστούν ανεξάρτητα. Αν οι ροές παρέχουν είσοδο για πολλούς χειριστές (φορείς), οι ουρές μπορούν να μοιραστούν αυξάνοντας την αποδοτικότητα της μνήμης. Σημειώστε ότι διάφοροι χειριστές μπορεί να έχουν διαφορετικές προτιμήσεις για το ποιές πλειάδες να εκδιώξουν από την ουρά. Αυτό μπορεί να ληφθεί υπόψη από την εξέταση της εισόδου από διάφορες ενότητες στατιστικών. Το ολοκληρωμένο μοντέλο (integrated model) συνδιάζει συνδιάζει την ουρά με την μνήμη συνένωσης. Τα οφέλη είναι δυνητικά καλύτερες αποφάσεις με βάση τη συνδυασμένη ακριβή γνώση και των δύο περιεχομένων της μνήμης. Αφ ετέρου η εσωτερική ουρά δεν μπορεί να μοιραστεί εύκολα με άλλους χειριστές.

Γρήγορο CPU vs. Αργού CPU. Για τους σκοπούς της ανάλυσης θα κάνουμε διάκριση ανάμεσα στη περίπτωση του γρήγορου CPU και του αργού CPU. Το σύστημα είναι ένα γρήγορο CPU σύστημα αν οι εισερχόμενες πλειάδες μπορούν να υποβάλλονται σε επεξεργασία τουλάχιστον τόσο γρήγορα όσο φτάνουν. Η ουρά δεν χρειάζεται, δεδομένου του ότι οι πλειάδες άμεσα σπρώχνονται στην συνένωση, ως εκ τούτου και το modular και το ολοκληρωμένο (integrated) μοντέλο ουσιαστικά είναι ισοδύναμα. Εννοιολογικά η συνένωση έχει εσωτερική κατάσταση μεγέθους M και δύο επιπλέον κελιά απομόνωσης για την νεοφερμένη πλειάδα κάθε ροής. Όταν οι πλειάδες φτάνουν άμεσα ενώνονται με τις πλειάδες συντρόφους τους της άλλης σχέσης στη μνήμη συνένωσης. Στη συνέχεια αποφασίζεται εάν η πλειάδα θα προστεθεί στη μνήμη συνένωσης (ενδεχομένως διώχνοντας μια άλλη πλειάδα). Ως εκ τούτου η πλειάδα που φθάνει πάντα θα φαίνεται από την συνένωση. Ο στόχος βελτιστοποίησης είναι να εκδιώξει και να διατηρήσει πλειάδες έτσι ώστε το λάθος προσέγγισης να ελαχιστοποιείται. Πιθανά μέτρα λάθους θα συζητηθούν σύντομα.

Στην περίπτωση του αργού CPU οι πλειάδες φτάνουν πιο γρήγορα από ότι μπορούν να υποβάλλονται σε επεξεργασία. Αυτό σημαίνει ότι η ουρά είναι αναγκαία για κράτηση για λίγο εισερχόμενων πλειάδων. Ο χειριστής συνένωσης τραβά τις πλειάδες από την ουρά κάθε φορά που έχει επεξεργαστεί την προηγούμενη είσοδο. Σαφώς, η ουρά θα γεμίζει την πάροδο του χρόνου και θα υπερχειλίσει, ως εκ τούτου οι πλειάδες πρέπει να μειωθούν από αυτό χωρίς ποτέ να φτάσουν στην συνένωση. Αυτό αναφέρεται σαν απόρριψη φορτίου. Εάν μια πλειάδα φτάσει στην συνένωση, υποβάλλεται σε επεφεργασία, όπως αναφέρεται για την γρήγορη

περίπτωση CPU. Η αργή περίπτωση CPU επομένως αποτελεί μια γενίκευση γρήγορης περίπτωσης CPU. Στην τελευταία περίπτωση, προσεγγίσεις προκύπτουν λόγω των περιορισμών της μνήμης, ενώ στην πρώτη περίπτωση, προσεγγίσεις προκύπτουν λόγω και των περιρισμών μνήμης και των περιορισμών επεξεργασίας. Η απόρριψη φορτίου στην ουρά επιρεάζει τα περιεχόμενα των ροών που φτάνουν στον χειριστή συνένωσης, επομένως ο στόχος βελτιστοποίησης είναι να βρεθεί η καλύτερη προσέγγιση συνένωσης σε όλες τις πιθανές στρατηγικές περικοπής φορτίου.

3.3 Μέτρα σφάλματος

Η απόδοση του χειριστή συνένωση είναι ένα set από πλειάδες, με ακρίβεια ένα multiset. Στο παρακάτω για απλούστευση ο όρος set θα αναφέρεται επίσης σε multisets. Δεν υπάρχει ενιαίο γενικά αποφεκτό μέτρο για την αξιολόγιση της ποιότητας μιας προσέγγισης σε ένα set-valued αποτέλεσμα ερωτήματος. Ένα καλά – γνωστό και ευρέως χρησιμοποιούμενο μέτρο είναι η συμμετρική διαφορά. Για τα δύο σύνολα X και Y υπολογίζεται σαν |(X–Y) U (Y–X)|. Για ισο-ενώσεις (equi-join) η απόρριψη πλειάδων πριν από τη λήξη φυσικά οδηγεί σε μια κατάσταση όπου η παραγόμενη απόδοση είναι ένα υποσύνολο του ακριβούς αποτελέσματος συνένωσης (δηλαδή, το αποτέλεσμα αν δεν υπήρχε έλλειψη πόρων). Στην περίπτωση αυτή η συμμετρική διαφορά απλοποιεί με τον αριθμό των αγχνοούμενων πλειάδων απόδοσης. Συνεπώς, θα αναφερθούμε σ'αυτό σαν μέτρο MAX-υποσύνολο. Αυτό το μέτρο θα είναι η κύρια εστίαση αυτής της εργασίας.

Στις κοινότητες εξόρυξης δεδομένων και ανάκτησης πληροφοριών κάμποσα συνολοθεωριτικά μέτρα ομοιότητας έχουν χρησιμοποιηθεί. Τα πιο διαδεδομένα μέτρα ομοιότητας ανάμεσα σε δύο σύνολα X και Y είναι αντίστοιχοι συντελεστές |X ⍰ Y|, dice συντελεστής $2\frac{X \cap Y}{|X|+|Y|}$, Jaccard συντελεστής $|\frac{X \cap Y}{X \cup Y}|$ και Cosine συντελεστής $\frac{|X \cap Y|}{\sqrt{|X|+|Y|}}$. Για X ⊆ Y όλα αυτά τα μέτρα μεγιστοποιούνται με την μεγιστοποίηση του μεγέθους του συνόλου X, ως εκ τούτου είναι ισοδύναμα με MAX – υποσύνολο. Ο συντελεστής επανάληψης $\frac{|X \cap Y|}{\min\{|X|,|Y|\}}$ ισούται με 1 για X ⊆ Y.

Η πρόσφατα εισαγόμενη Earth Mover's Distance (EDM) χρησιμοποιείται κυρίως σαν μέτρο ομοιότητας στην επεξεργασία εικόνας. Ορίζεται σαν ποσότητα εργασίας που αποκτείται για την μετατροπή ενός συνόλου X σε ένα άλλο σύνολο Y ίσης ή μεγαλύτερης μάζας (αριθμός πλειάδων). Και τα δύο σύνολα θεωρούνται σαν μάζες που κατανεμήθηκαν σε χώρο δεδομένων για τον οποίο υπάρχει μέτρο απόστασης. Η δουλειά που απαιτείται για την μετατροπή X σε Y

μετριέται ως το άθροισμα όλων των προϊόντων μάζας που μεταφέρονται από την απόσταση μεταφοράς. Η EMD είναι η ελάχιστη δουλειά πάνω σε όλες τις πιθανές μετατροπές του X σε Y. Αν X $\subseteq$ Y, τότε επιπόλαια αξιολογείται σε 0.

Το Match and Compare (MAC) set μέτρο ομοιότητας επίσης απαιτεί μια απόσταση μετρική ανάμεσα στις πλειάδες των δύο συνόλων. Πρώτα βρίσκεται ελάχιστη κάλυψη κόστους του πλήρους διμερούς γραφήματος του οποίου οι κόμβοι αντιστοιχούν στις πλειάδες και του οποίου τα άκρα έχουν το βάρος των αντίστοιχων αποστάσεων. Στη συνέχεια η συνολική απόσταση υπολογίζεται σαν συνάρτηση των βαρών άκρων στο κάλυμμα και του αριθμού των άκρων προσπίπτων σε κάθε κόμβο.

Εισάγουμε μια καινούργια μέτρηση λάθους, το Αρχείο-μετρικό (ArM) το οποίο έχει σημασία για την σημασιολογική εξομάλυνση φορτίου. Όπως αναφέραμε νωρίτερα, σε κάποιες εφαρμογές τα δεδομένα εισόδου αποθηκεύονται σε αρχεία και ως εκ τούτου κατά τη διάρκεια χαμηλού-φορτίου περιόδων μπορούν να χρησιμοποιηθούν για την βελτίωση κατά προσέγγιση αποτελεσμάτων τα οποία επιτεύχθηκαν κατά τη διάρκεια περιόδων υψηλού φορτίου. Δεδομένου ότι το τελικό αποτέλεσμα θα είναι ακριβές, το ArM δεν μετρά την ποιότητα προσέγγισης αλλά μάλλον το πόσο εργασία απαιτείται για την ολοκλήρωση ατελούς εργασίας. Δεδομένου ότι η πρόσβαση σε ένα αρχείο είναι πολύ πιο αργή από την πρόσβαση μνήμης, προσεγγίζουμε αυτό το μετά την επεξεργασία κόστος από τον αριθμό των πλειάδων οι οποίες δεν θα μπορούσαν να συνδιαστούν με όλες τις πλειάδες εταίρους τους εξαιτίας έλλειψης πόρων (κατά τη διάρκεια του φορτίου αιχμής). Αυτό το μέτρο ποιότητας είναι μια απλούστευση, καθόσον προϋποθέτει ότι το μετά-επεξεργασίας κόστος (διαβάζοντας ελλιπείς πλειάδες και εταίρους τους από το αρχείο και εντάσσοντας αυτές) εξαρτάται γραμμικά από τον αριθμό τέτοιων πλειάδων μόνο. Άλλα μέτρα όπως ο μέσος χρόνος επεξεργασίας ανά πλειάδα ή η μέση καθυστέρηση απόδοσης είναι λογικά αλλά δεν συζητιούνται λεπτομερώς εδώ λόγω της έλλειψης χώρου.

Το ArM επισήμως ορίζεται ως εξής. Έστω

$$\delta_R(i, j) = \begin{cases} 1, & \text{αν } r(i) \text{επέζησε για τουλάχιστον } j \text{ χρόνο στη μνήμη} \\ 0, & \text{διαφορετικά} \end{cases}$$

Αυτή η μεταβλητή δείχνει όταν μια πλειάδα που φθάνει σε χρόνο i είναι ακόμα στη μνήμη και ομοίως ορίζεται για ροή S.

Επιπλέον, χρησιμοποιούμε

$$S^{<}(i) = \{ j : j \in [i - w + 1,\ i - 1] \wedge s(j) = r(i)\}.$$

Αυτό το set δείκτη περιέχει τους χρόνους άφιξης όλων των εταίρων συνένωσης του r(i) στο S που έφθασαν πριν από αυτό. Ένα παρόμοιο set $R^{<}$ ορίζεται για κάθε S(i).

Σημειώστε ότι r(i) υπόκειτε σε απόλυτη επεξεργασία εάν όλα S(j) για $j \in S^{<}(i)$ είναι στη μνήμη τη χρονική στιγμή i και αν r(i) είναι στη μνήμη μέχρι τη χρονική στιγμή $j_{r(i)}$, όπου $j_{r(i)} = \max \{ j : j \in [i, i + w - 1] \wedge s(j) = r(i)\}$ (δηλαδή τη στιγμή που ο τελευταίος εταίρος συνένωσης του r(i) φτάνει στη ροή S). Αφού $\delta_R(i,j)$ είναι ίσο με 0 και μόνο αν r(i) δεν έχει επιζήσει για j χρόνο, ξέρουμε ότι r(i) είναι ατελές αν και μόνο αν $\delta_R(i , j_{r(i)} - i)\prod_{j \in S<(i)}\delta_{S(j , i-j)} = 0$. Έτσι ArM ορίζεται σαν

$$\Sigma(\delta_R(i, j_{r(i)} - i) \prod \delta_{S(j , j - i)} \quad + \quad \delta_{S(i, j_{S(i)} - i)} \prod \delta_{R(j , j - i)}).$$

3.4 Συμπέρασμα και Μελλοντική δουλειά

Συζητήσαμε το πρόβλημα του κατά προσέγγιση υπολογισμού των συνενώσεων σε συρόμενο παράθυρο για ροές δεδομένων. Ορίσαμε το πρόβλημα χώρου των προσεγγίσεων συνένωσης που βασίζεται σε λεπτούς κόκκους πλειάδων χρησιμοποιώντας διάφορα σετ μετρήσεων λάθους και εξετάσαμε τη μέτρηση υποσυνόλου MAX σε βάθος και δώσαμε άριστους αλγόριθμους για συνενώσεις σε συρόμενο παράθυρο. Πιστεύουμε ότι αυτή η δουλειά δείχνει ότι η σημασιολογική απόρριψη φορτίου, δηλαδή η προσαρμογή σε ελλείψεις πόρων απορρίπτοντας πλειάδες βασιζόμενοι στις αξίες τους, είναι σαφώς ανώτερη από την τυχαία απόρριψη φορτίου με μικρό κόστος γενικών εξόδων για την διατήρηση απλών στατιστικών ροής.

Επίσης, προτείναμε το καινούριο αρχείο-μετρικό σαν καινούργιο μέτρο αξιολόγησης της απόδοσης αλγορίθμων συνένωσης πάνω σε συρόμενα παράθυρα για συστήματα ροής δεδομένων με την υποστήριξη ενός αρχείου.

Αυτή η δουλειά εξέτασε μόνο μέρος του συνολικού προβλήματος χώρου και πολλά προβλήματα παραμένουν ανοικτά. Η ανάπτυξη αποδοτικών αλγόριθμων για το αρχείο-μετρικό είναι μέρος της μελλοντικής δουλειάς. Επίσης θα εξετάσουμε τα άλλα μοντέλα επεξεργασίας συνένωσης, ιδιαίτερα την περίπτωση της αργής CPU. Μια άλλη ενδιαφέρουσα κατεύθυνση μελλοντικής δουλειάς είναι να εξετάσουμε πως πολλαπλά ερωτήματα μπορούν αποτελεσματικά να μοιράζουν πόρους και πως να συνδιάζουν την σημασιολογική απόρριψη φορτίου με την επιλογή εφαρμογών συνένωσης.

Κεφάλαιο 4

Κύριο μέρος της Εφαρμογής

4.1 Ο αλγόριθμος FIFO (First In First Out)

4.1.1 Εισαγωγή

Ο Αλγόριθμος **FIFO** ή αλλιώς **First In First Out** είναι ο πρώτος από τους τέσσερις που θα μελετήσουμε σε αυτό το έγγραφο. Όπως αναφέραμε παραπάνω, ο κάθε αλγόριθμος διαφέρει με τους άλλους στον τρόπο διαγραφής μιας καταχώρησης για να μπεί μία νέα. Με άλλα λόγια τα προγράμματα διαφέρουν στις συνθήκες διαγραφής ενός αποθηκευμένου ακέραιου, είτε της μίας ροής είτε της άλλης, και αντικατάστασης αυτού.

Ο τρόπος διαγραφής του αλγόριθμου FIFO είναι ότι διαγράφει τον ακέραιο που καταχωρήθηκε/αποθηκεύτηκε πιο πρώτος από τους άλλους. Αυτός δηλαδή που μπήκε πρώτος, θα φύγει και πρώτος. Εξού και το όνομα First-In-First-Out.

4.1.2 Επεξήγηση αλγορίθμου

#include <stdio.h>

#include <stdlib.h>

Όταν ο προεπεξεργαστής συναντήσει μια οδηγία *#include*, κοιτάζει το όνομα αρχείου που ακολουθεί και το συμπεριλαμβάνει μέσα στο τρέχον αρχείο. Οι γωνιακές παρενθέσεις (< , >) λένε στον προεπεξεργαστή να ψάξει για το αρχείο σε ένα ή περισσότερους από τους τυποποιημένους καταλόγους αρχείων του συστήματος.

Ο λόγος που συμπεριλαμβάνουμε τα αρχεία αυτά είναι επειδή περιέχουν πληροφορίες που τις χρειαζόμαστε. Για παράδειγμα, το αρχείο **stdio.h** περιέχει συνήθως τους ορισμούς των *getchar(), putchar(), printf(), scanf()* κ.α. ενώ το αρχείο **stdlib.h** μας επιτρέπει να χρησιμοποιήσουμε μαθηματικές συναρτήσεις, όπως για παράδειγμα η συνάρτηση *rand()* την οποία θα δούμε παρακατω στον αλγόριθμο.

#define NUMBERS 240

Κατά την μεταγλώττιση του προγράμματος η τιμή 240 θα αντικαθιστά την μεταβλητή NUMBERS κάθε φορά που χρησιμοποιείται αυτή. Αυτό καλείται αντικατάσταση κατά την *ώρα της μεταγλώττισης*. Την ώρα που εμείς τρέχουμε το πρόγραμμα, έχουν ήδη γίνει όλες οι αντικαταστάσεις.

void getStart(int * , int * , int * , int * , int , int , int * , int *);

int insertA(int * , int * , int * , int , int , int , int *);

int insertB(int * , int * , int * , int , int , int , int *);

void results(int * , int * , int , int , int *);

Κάθε μία εντολή είναι μία δήλωση μιας συνάρτησης. Για παράδειγμα η εντολή «*int insertA(int * , int * , int * , int , int , int , int *);»* μας δηλώνει ότι η συνάρτηση θα έχει τιμή επιστροφής έναν int (ακέραιο), το όνομά της είναι insertA και έχει 7 ορίσματα απο τα οποία τα τέσσερα είναι ένας δείκτης σε ακέραιο (int *) και τα άλλα τρία είναι ένας ακέραιος (int). Έχει

σημασία η σειρά που είναι γραμμένα τα ορίσματα διότι με την ίδια σειρα πρέπει να σταλούν οι τιμές παρακάτω στον αλγόριθμο.

Αντίστοιχα και στις άλλες τρείς συναρτήσεις, με την διαφορα ότι η λέξη void δηλώνει ότι η συνάρτηση δεν θα έχει τιμή επιστροφής.

main(){

int i , x , y , PA , PB , acount , bcount , count1=0;

int *A , *A1 , *B , *B1;

Το **main** είναι αλήθεια ένα απλό όνομα, αλλά είναι η μόνη εκλογή που έχουμε. Η εκτέλεση ενός προγράμματος σε C αρχίζει πάντα με μια συνάρτηση που καλείται **main()**.

Οι επόμενες δύο προτάσεις, είναι προτάσεις δήλωσης. Η πρώτη πρόταση δήλωσης μας αναγγέλει ότι θα χρησιμοποιήσουμε οκτώ μεταβλητές με τα ονόματα *i, x , y , PA , PB , acount , bcount* και *count1* αντίστοιχα και ότι θα είναι τύπου **int** (ακέραια). Επίσης δίνει αρχική τιμή στη μεταβλητή *count1*, την τιμή 0. Οι υπόλοιπες μεταβλητές που δεν πήραν αρχική τιμή, το σύστημα τους έχει δώσει τυχαίες τιμές και παρακάτω στο πρόγραμμα εμείς δίνουμε στην κάθε μεταβλητή την τιμή που επιθυμούμε.

Την μεταβλητή **i** θα την χρησιμοποιήσουμε για μετρητή στους διάφορους βρόχους επανάληψης της συνάρτησης main. Οι ακέραιες μεταβλητές **x** , **y** θα πάρουν τιμές από τον χρήστη, κάτι που θα δούμε παρακάτω, και θα αποθηκεύσουν τον αριθμό των πλειάδων (ή αλλιώς θέσεων) των πινάκων Α και Β αντίστοιχα. Οι **PA** , **PB**, όπως και οι x,y, θα πάρουν τιμές από τον χρήστη και συμβολίζουν το πόσες φορές θα εκτελεστεί η συνάρτηση insertA και insertB αντίστοιχα, τις οποίες θα δούμε παρακάτω. Οι **acount** , **bcount** θα κρατούν την σειρά που μπαίνουν οι ακέραιοι στους πίνακες Α και Β αντίστοιχα. Όταν για παράδειγμα εισέλθει ο $35^{ος}$ ακέραιος στη $6^{η}$ θέση του πίνακα Α, τότε ο acount θα δώσει την τιμή 35 στην $6^{η}$ θέση του πίνακα Α1. Έπειτα, στην επόμενη εντολή έχουμε την δημιουργία τεσσάρων δεικτών. Παρακάτω θα δούμε ότι κάθε ένας από αυτούς θα δεσμεύσει χώρο στη μνήμη ανάλογα με τις τιμές των x,y. Όπως προαναφέραμε, η δουλειά των πινάκων **A1** και **B1** είναι να κρατάνε σε αντιστοιχία με τους πίνακες **A** και **B**, την σειρά που μπήκαν οι ακέραιοι έτσι ώστε όταν θα θέλουμε να διώξουμε τον ακέραιο που εισήλθε πιο πριν από τους άλλους, να μπορούμε να τον βρούμε εύκολα.

printf("Dose posa insertA theleis na exeis\n");

scanf("\n%d", &PA);

printf("Dose posa insertB theleis na exeis\n");

scanf("\n%d", &PB);

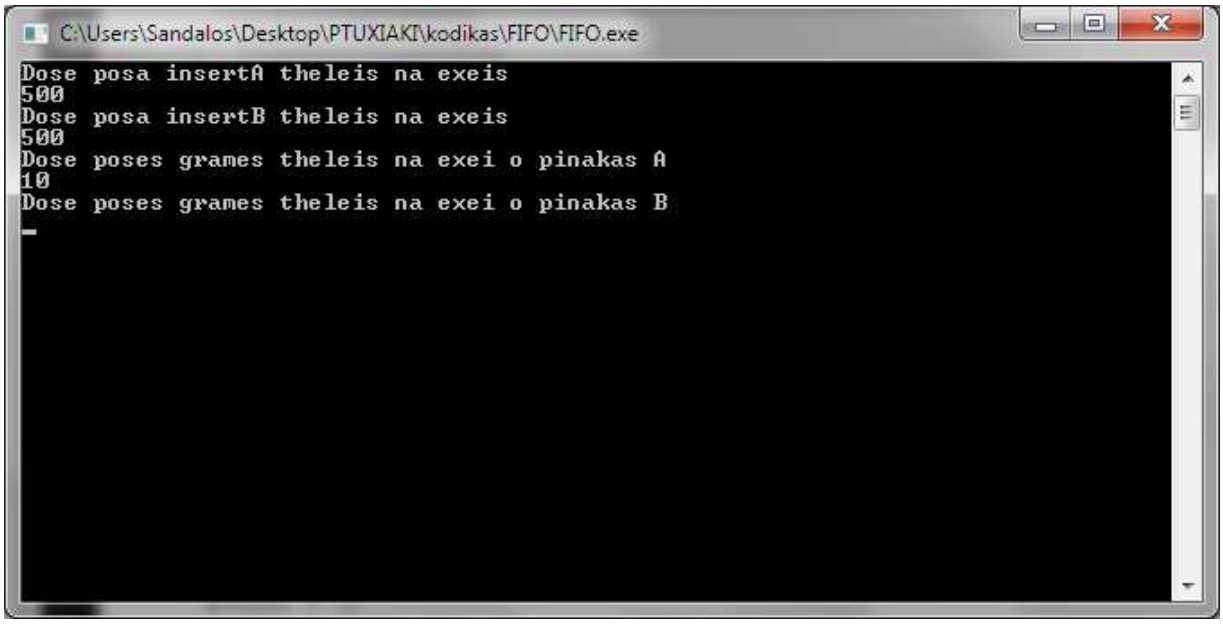
printf("Dose poses grames theleis na exei o pinakas A\n");

scanf("\n%d", &x);

printf("Dose poses grames theleis na exei o pinakas B\n");

scanf("\n%d", &y);

Εδώ διαβάζουμε τιμές από τον χρήστη και τις αποθηκεύουμε στις μεταβλητές PA, PB, x και y αντίστοιχα. Στην παρακάτω εικόνα βλέπουμε το παράθυρο το οποίο εμφανίζεται κατά την διάρκεια εκτέλεσης του προγράμματος, στο οποίο έχουμε δώσει την τιμή 500 στις μεταβλητές PA και PB, ενώ η μεταβλητή x αποθήκευσε την τιμή 10 και για τη μεταβλητή y δεν έχουμε δώσει ακόμα τιμή.



Εικόνα 1

A = (int *)malloc(x * sizeof(int));

A1 = (int *)malloc(x * sizeof(int));

B = (int *)malloc(y * sizeof(int));

B1 = (int *)malloc(y * sizeof(int));

acount = 1;

bcount = 1;

Στις παραπάνω εντολές έχουμε αρχικοποιήσεις μεταβλητών. Συγκεκριμένα, οι τέσσερις πρώτες εντολές με την βοήθεια της συνάρτησης malloc() δεσμεύουν (συνεχόμενο) χώρο στην μνήμη. Οι **A** και **A1** δεσμεύουν όσο x * τα byte που πιάνει ένας integer στην μνήμη. Αν για παράδειγμα ο ένας integer πιάνει 2 bytes, και έχουμε δώσει τιμή 10 στη μεταβλητή x, τότε οι A και A1 θα δεσμεύσουν 10*2 = 20 bytes ο καθένας, ή αλλιώς χώρο για 10 ακέραιους. Αντίστοιχα και οι B, B1. Στις μεταβλητές **acount** και **bcount** δίνουμε αρχική τιμή 1, της οποίας την χρησιμότητα θα δούμε παρακάτω.

getStart(A , A1 , B , B1 , x , y , &acount , &bcount);

Εδώ καλούμε την συνάρτηση *getStart* η οποία έχει ως ορίσματα τους δείκτες *A, A1, B, B1, &acount, &bcount* καθώς και τους ακέραιους *x, y* με την σειρά που φαίνεται παραπάνω.

Ο δείκτης **A** «δείχνει» στο πρώτο στοιχείο του πίνακα A, αντίστοιχα και οι A1, B, B1, και οι δείκτες **&acount** και **&bcount** «δείχνουν» τους ακέραιους account και bcount αντίστοιχα. Οι τέσσερις πρώτοι δείκτες είναι μεταβλητοί δείκτες, αφού τους δημιουργήσαμε εμείς παραπάνω στις δηλώσεις μεταβλητών. Παράλληλα οι δείκτες των ακεραίων είναι σταθεροί δείκτες, αφού δημιουργήθηκαν από μόνοι τους όταν εμείς δημιουργήσαμε τις αντίστοιχες μεταβλητές στις οποίες δείχνουν (το ότι είναι σταθεροί δείκτες το καταλαβαίνουμε άλλωστε με το & το οποίο βρίσκεται μπροστά από τους σταθερούς δείκτες μόνο).

Τιμή επιστροφής η συνάρτηση *getStart* δεν έχει. Όταν έγινε η δήλωση της συνάρτησης πρίν το όνομα αυτής, βάλαμε την λέξη-κλειδί **void** η οποία δηλώνει ότι η συνάρτηση δεν επιστρέφει τίποτα.

Η υλοποίηση της συνάρτησης αυτής φαίνεται παρακάτω, αφού τελειώσει η main(), όπως και σε όλες τις συναρτήσεις που καλούμε. Εδώ είναι απλά το κάλεσμά της, καθώς και τα ορίσματα. Παρακάτω θα δούμε ότι η δουλειά της είναι να δώσει αρχικές (τυχαίες) τιμές για να γεμίσει αρχικά τους δύο πίνακες (A, B) που κρατούν τις τιμές των δύο τυχαίων ροών, καθώς

και τους δύο πίνακες (A1, B1) οι οποίοι συγκρατούν για κάθε τιμή του πίνακα Α και του πίνακα Β αντίστοιχα την σειρά την οποία εισήλθαν, έτσι ώστε να γνωρίζουμε την σειρά την οποία ήρθε η κάθε τιμή της κάθε ροής.

```
for(i=x; i<PA; i++){
    acount = insertA(A , A1 , B , x , y , acount , &count1);
}
for(i=y; i<PB; i++){
    bcount = insertB(B , B1 , A , x , y , bcount , &count1);
}
```

Κάθε φορά που εκτελούνται οι εντολές που βρίσκονται μέσα στο block εντολών της for, καλείται η συνάρτηση **insertA**, η οποία έχει τιμή επιστροφής έναν **int** (ακέραιο) και ορίσματα τους δείκτες *A, A1, B, &count1* καθώς και τους ακέραιους *x, y, acount* όπως φαίνεται παραπάνω.

Η τιμή επιστροφής της συνάρτησης αποθηκεύεται στην ακέραια μεταβλητή **acount**, η οποία στέλνεται ως ορισμά σε κάθε κάλεσμα της συνάρτησης όπως και τα υπόλοιπα ορίσματα. Η συνάρτηση θα καλεσθεί όσες φορές ισχύει η συνθήκη της for.

Ο μετρητής **i** ξεκινά από την θέση **x** διότι έχουμε ήδη δώσει x τιμές στους πίνακες (από την συνάρτηση *getStart*), άρα οι τιμές που μένουν να δώσουμε, αφού συνολικά είναι **PA** τιμές, είναι PA-x, ή αλλιώς από το x μέχρι το PA.

Αντίστοιχα λειτουργεί και η επόμενη for.

```
    results(A , B , x , y , &count1);
}
```

Στην παραπάνω εντολή, καλούμε την συνάρτηση **results**, η οποία έχει ως ορίσματα τους δείκτες *A, B, &count1* καθώς και τους ακέραιους *x, y* με την σειρά που φαίνεται παραπάνω.

Τιμή επιστροφής δεν έχει, και η δουλειά της είναι να κάνει τις μετρήσεις των εκάστοτε join που έχουν οι δύο πίνακες (A,B) των τυχαίων ροών και να τις εμφανίσει στον χρήστη.

Το άγκιστρο κλεισίματος (}) αναφέρεται στην συνάρτηση main(), και μας δηλώνει ότι σε εκείνο το σημείο σταματάνε οι εντολές της.

Στη συνέχεια θα αναλύσουμε τις συναρτήσεις τις οποίες καλέσαμε κατα την διάρκεια εκτέλεσης της main().

void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y , int *acount , int *bcount){

int i , a , b , c=0 , i2;

A[0] = rand()% NUMBERS+1;

B[0] = rand()% NUMBERS+1;

A1[0] = 0;

B1[0] = 0;

Εδώ ξεκινά η υλοποίηση της συνάρτησης **getStart.** Τιμή επιστροφής δεν έχει, όπως προαναφέραμε, και αυτό δηλώνεται με την λέξη-κλειδί **void** πριν το όνομα της συνάρτησης. Αμέσως μετά, μέσα στις παρενθέσεις, βλέπουμε ότι η συνάρτηση δημιουργεί, και δηλώνει ταυτόχρονα, αντίγραφα των μεταβλητών που ήταν ως ορίσματα. Για παράδειγμα, οι ακέραιες μεταβλητές *x* και *y* έχουν τοπική εμβέλεια. Δηλαδή, επειδή δημιουργήθηκαν στην συνάρτηση *getStart,* τις αναγνωρίζει μόνο η ίδια. Όταν δηλαδή τελειώσει η getStart και γυρίσει ο compiler στην συνάρτηση main(), δεν θα γνωρίζει καμία μεταβλητή την οποία έχει δημιουργήσει η getStart. Αντίστοιχα, έχουν δημιουργηθεί οι δείκτες *A, A1, B, B1, acount, bcount* οι οποίοι έχουν πάρει τις αντίστοιχες τιμές των ορισμάτων, όπως και οι ακέραιοι επίσης.

Στη συνέχεια, δημιουργούμε τις ακέραιες μεταβλητές *i, a, b, c, i2* και δίνουμε αρχική τιμή στην μεταβλητή c την τιμή 0.

Η μεταβλητή **i** θα λειτουργήσει ως μετρητής στους βρόχους επαναλήψεων, όπως και η μεταβλητή **i2**. Στη μεταβλητή **a** θα δίνουμε μία τυχαία τιμή κάθε φορά, και αφού περάσει τους κατάλληλους ελέγχους και δούμε ότι είναι κατάλληλη για να εισαχθεί στον πίνακα A τότε θα δίνει την τιμή της στην εκάστοτε θέση του πίνακα A. Αντίστοιχα θα λειτουργεί και η μεταβλητή **b** για τον πίνακα B. Η μεταβλητή **c** παίρνει τιμές στο προγραμμά μας 0 και 1. Την χρησιμοποιούμε για να καταλαβαίνουμε αν ο compiler έχει μπεί σε block εντολών της if.

Αμέσως μετά, δίνουμε αρχικές τιμές στις πρώτες θέσεις των πινάκων A και B αντίστοιχα. Οι τιμές που δίνουμε προέρχονται από την συνάρτηση **rand()**, η οποία σε συνδιασμό με το υπόλοιπο της εντολής (δηλαδή, η εντολή rand()% NUMBERS+1;) θα μας

δώσει αποτέλεσμα έναν (τυχαίο) ακέραιο αριθμό στην γκάμα 0-240 (η γκάμα είναι έως 240 διότι η σταθερά NUMBERS έχει την τιμή αυτή). Στις πρώτες θέσεις των πινάκων A1 και B1, οι οποίοι κρατούν την σειρά του αντίστοιχου ακέραιου που μπήκε στον πίνακα Α, όπως και στον πίνακα Β, δίνουμε την τιμή 0 για να εκφράσουμε ότι είναι οι πρώτοι που εισέρχονται.

```
for(i=1; i<x; i++){
    a = rand()% NUMBERS+1;
    for(i2=0; i2<i; i2++){
        if(A[i2] == a){
            i--;
            c = 1;
            break;
        }
    }
```

Στον παραπάνω αλγόριθμο έχουμε μια εμφωλευμένη for μέσα σε μία άλλη. Καθώς έχουμε δώσει μια πρώτη τιμή στον πίνακα Α, βάζουμε σε μια μεταβλητή (**a**) έναν τυχαίο ακέραιο, στην αντίστοιχη γκάμα, και στη συνέχεια ελέγχουμε αν αυτός ο ακέραιος υπάρχει στον πίνακα Α. Αυτή η διαδικασία γίνεται διότι θέλουμε το κάθε στοιχείο του εκάστοτε πίνακα (A,B) να είναι μοναδικό. Αν η τιμή που διαβάσαμε υπάρχει στον πίνακα Α, τότε μπαίνει στο block εντολών της if και **μειώνει τον μετρητή** (**i**) κατά μία μονάδα (έτσι ώστε την επόμενη φορά που θα ξαναέρθουμε σε αυτό το σημείο να θέλουμε να βρούμε τον κατάλληλο τυχαίο αριθμό για την ίδια θέση και όχι για μια θέση παρακάτω), **δίνει στην ακέραια μεταβλητή c την τιμή 1** (από την οποία παρακάτω θα καταλάβουμε ότι μπήκε μέσα ο compiler,δηλαδή ότι η ακέραια τιμή που ήταν στην μεταβλητή a υπάρχει στον πίνακα Α). Αμέσως μετά έρχεται η εντολή **break** η οποία μας βγάζει απο την επανάληψη την οποία βρισκόμαστε και πάει τον compiler στον τελευταίο άγκιστρο που φαίνεται στον παραπάνω αλγόριθμο. Η εντολή break χρησιμοποιείται διότι αν βρούμε έστω και μία φορά ότι η τιμή που θέλουμε να δώσουμε στην εκάστοτε θέση του πίνακα υπάρχει σε αυτόν, τότε πρέπει να διαβάσουμε άλλη τιμή και είναι άσκοπο να συνεχίσουμε να ψάχνουμε τον πίνακα Α. Η break λοιπον σταματά το ψάξιμο αυτό.

Προσοχή, αν φτάσουμε στην εντολή break, βγαίνουμε από την εμφωλευμένη for και όχι από την αρχική.

```
        if(c == 0){
            A[i] = a;
            A1[i] = *acount;
            *acount += 1;
        }
        c = 0;
    }
    c = 0;
```

Στη συνέχεια, ελέγχουμε αν η ακέραια μεταβλητή c έχει τιμή 0 (έχει αυτή την τιμή μόνο εάν δεν μπήκε στην if, δηλαδή, μόνο αν η τιμή που είχε η ακέραια μεταβλητή a δεν προϋπήρχε στον πινακα Α). Αν ισχύει η συνθήκη της if, τότε σημαίνει ότι η τιμή που αντιπροσωπεύει η ακέραια μεταβλητή a είναι έτοιμη να εισαχθεί στον πίνακα. Αφού καταχωρηθεί (με την εντολή **A[i]=a;**) δίνουμε τα **περιεχόμενα του δείκτη** acount (με την εντολή ***acount**) στην i θέση του πίνακα Α1. Ο πίνακας Α1 κρατά την σειρά που εισήλθε ο ακέραιος που βρίσκεται στην αντίστοιχη θέση του πίνακα Α. Βγαίνοντας από την if, μηδενίζουμε την ακέραια μεταβλητή c, για να έχει μηδενική τιμή στην επόμενη επανάληψη, καθώς την ξανά μηδενίζουμε όταν βγαίνουμε από την for για να έχει μηδενική τιμή παρακάτω που θα κάνουμε την αντίστοιχη δουλειά για τον πίνακα Β.

```
for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(B[i2] == b){
                i--;
                c = 1;
                break;
            }
```

```
        }
        if(c == 0){
            B[i] = b;
            B1[i] = *bcount;
            *bcount += 1;
        }
        c = 0;
    }
}
```

Εδώ βλέπουμε ότι γίνεται η αντίστοιχη δουλειά για τον πίνακα Β, όπως και για τον πίνακα Α που είδαμε στο προηγούμενο κομμάτι.

```
int insertA(int *A , int *A1 , int *B , int x , int y , int acount , int *count1){
   int i , ak , pos , a;
```

Στον παραπάνω κομμάτι του αλγορίθμου έχουμε την λέξη-κλειδί **int** που μας δηλώνει ότι η τιμή επιστροφής είναι ένας ακέραιος. Το όνομα της συνάρτησης είναι **insertA** και έχει ως ορίσματα τους δείκτες *A, A1, B, count1* καθώς και τους ακέραιους *x, y, acount*. Αμέσως μετά δηλώνουμε τις μεταβλητές που θα χρησιμοποιήσουμε παρακάτω.

Η μεταβλητή **i** έχει τον ρόλο και πάλι του μετρητή. Την μεταβλητή **ak** θα την έχουμε ως μέτρο σύγκρισης για να βρίσκουμε κάθε φορά την ελάχιστη ακέραια τιμή στον πίνακα Α1. Στην μεταβλητή **a** θα δίνουμε μια τυχαία ακέραια τιμή στην αντίστοιχη γκάμα που θα βρισκόμαστε κάθε φορά για να γίνει η νέα καταχώρηση. Ο ακέραιος **pos** θα αποθηκεύει την θέση στην οποία βρήκαμε την ελάχιστη τιμή του πίνακα Α1.

```
a = rand()% NUMBERS+1;
```

Εδώ δίνουμε μια αρχική τιμή στην μεταβλητή a, η οποία είναι ένας τυχαίος ακέραιος στην γκάμα που βρίσκεται το εκάστοτε παράδειγμα, στην περίπτωσή μας έχουμε την γκάμα από 0 έως 240.

```
for(i=0; i<x; i++){
    if(A[i] == a){
        for(i=0; i<y; i++){
            if (B[i] == a){
              *count1 += 1;
              break;
            }
        }
        a = rand()% NUMBERS+1;
        i = -1;
    }
}
```

Όπως και στη συνάρτηση *getStart* έτσι και εδώ ελέγχουμε αν η τυχαία ακέραια τιμή που έχει εισέλθει στην μεταβλητή a υπάρχει ήδη στον πίνακα. Αυτή τη δουλειά την κάνει σε κάθε επανάληψη, της εξωτερικής for, η if, της οποίας αν ισχύει η συνθήκη τότε σημαίνει ότι η τιμή που έχει η μεταβλητή a υπάρχει ήδη στον πίνακα A. Τότε ο compiler εκτελεί τις εντολές της if, η οποία ελέγχει όλα τα στοιχεία του πίνακα B για να δεί αν κάνει έστω και με ένα στοιχείο join. Αν κάνει join, τότε θα το συμπεριλάβουμε στα αποτελέσματα κι ας μην εισέλθει η τιμή του a στον πίνακα. Λέμε έστω και ένα, διότι μόλις η συνθήκη της δεύτερης if γίνει αληθές (**B[i] == a**), τότε αφού αυξήσουμε την μεταβλητή **count1** (πιο σωστά αυξάνουμε τα περιεχόμενα του δείκτη count1) έχουμε την εντολή **break** η οποία μας βγάζει από την βρόχο της εμφωλευμένης for. Αμέσως μετά, αφού έχουμε δεί ότι η τυχαία ακέραια τιμή που διαβάσαμε και αποθηκεύσαμε στην μεταβλητή a δεν είναι κατάλληλη για να εισαχθεί στον πίνακα, δίνουμε μια άλλη τυχαία ακέραια τιμή στην μεταβλητή a. Το βήμα μας παίρνει την

τιμή -1, διότι τώρα η ακέραια μεταβλητή (**a**) έχει άλλη τιμή, και πρέπει να ψάξουμε από την αρχή τον πίνακά μας να δούμε αν υπάρχει αυτή η τιμή ήδη σε αυτόν.

```
ak = A1[0];
pos = 0;
for(i=1; i<x; i++){
    if (A1[i]<ak){
      ak = A1[i];
      pos = i;
    }
}
```

Σε αυτό το σημείο του αλγορίθμου κάνουμε μία αναζήτηση γιατί αφού στο προηγούμενο κομμάτι βρέθηκε ο αριθμός ο οποίος θα εισαχθεί, τώρα πρέπει να δούμε ποιος ακέραιος θα αποχαιρετίσει τον πίνακα. Αυτός που θα διαγραφεί, για να καταχωρηθεί ο νέος, είναι ο ακέραιος ο οποίος μπήκε πιο πρίν από τους άλλους στον πίνακα που βρίσκεται. Η σειρά του καθένα ακέραιου κρατείται σε ένα παράλληλο πίνακα, τον **A1**. Η θέση, λοιπόν, όπου ο A1 έχει την μικρότερη τιμή είναι και η αντίστοιχη του πίνακα A όπου πρέπει να αντικατασταθεί. Έτσι, λοιπόν, δίνουμε στην μεταβλητή **ak** την τιμή του πρώτου στοιχείου του πίνακα A1 (**A1[0]**) και στην μεταβλητή **pos** την τιμή 0. Υπενθυμίζουμε ότι η μεταβλητή pos θα μας κρατά την θέση που θα βρούμε τον μικρότερο ακέραιο. Στη συνέχεια, με τη βοήθεια του βρόχου for, ελέγχουμε ένα-ένα όλα τα στοιχεία του πίνακα A1, και μόλις βρούμε κάποια τιμή του που να είναι μικρότερη από την τιμή του ak τότε ο compiler μπαίνει στο block εντολών της if. Δίνουμε στην μεταβλητή ak την μικρότερη τιμή που βρίκαμε και η μεταβλητή pos κρατά την θέση του πίνακα που βρίσκεται αυτή. Στο τέλος της επανάληψης, η μεταβλητή ak έχει την μικρότερη τιμή του πίνακα A1 και η μεταβλητή pos συγκρατεί την θέση της στον πίνακα A1.

```
A[pos] = a;
A1[pos] = acount;
acount++;
```

```
   return acount;
}
```

Στον παραπάνω μέρος, βλέπουμε ότι τελικά καταχωρούμε την τυχαία ακέραιη τιμή που έχει η μεταβλητή a στην pos θέση του πίνακα A και η αντίστοιχη θέση στον πίνακα A1 παίρνει την τιμή της μεταβλητής **acount**, η οποία λειτουργεί ως μετρητής. Έπειτα, η μεταβλητή acount παίρνει την αύξηση κατά μία μονάδα (έτσι ώστε την επόμενη φορά που θα εισέλθει ένας ακέραιος στον A, η αντίστοιχη θέση στο πίνακα A1 θα έχει αυξηθεί) και η συνάρτηση επιστρέφει την τιμή της με την λέξη-κλειδί **return**. Το άγκιστρο κλεισίματος είναι της συνάρτησης *insertA*.

```
int insertB(int *B , int *B1 , int *A , int x , int y , int bcount , int *count1){
   int i , ak , pos , b;
   b = rand()% NUMBERS+1;
   for(i=0; i<y; i++){
        if(B[i] == b){
             for(i=0; i<x; i++){
                  if (A[i] == b){
                    *count1 += 1;
                    break;
                  }
             }
             b = rand()% NUMBERS+1;
             i = -1;
        }
   }
   ak = B1[0];
```

```
   pos = 0;
   for(i=1; i<y; i++){
         if (B1[i]<ak){
           ak = B1[i];
           pos = i;
         }
   }
   B[pos] = b;
   B1[pos] = bcount;
   bcount++;
   return bcount;
}
```

Παραπάνω έχουμε τον αλγόριθμο της συνάρτησης *insertB*. Η υλοποίηση είναι ίδια με την συνάρτηση *insertA* με τη διαφορά ότι τώρα κάνουμε την ίδια δουλειά στον πίνακα Β, αντί για του πίνακα Α.

```
void results(int *A ,  int *B , int x , int y , int *count1){
    int i , i2 , count;
    count = 0;
```

Σε αυτό το σημείο βλέπουμε τον ορισμό της συνάρτησης **results**. Όπως έχουμε πει και παραπάνω, η λέξη-κλειδί **void** μας δηλώνει ότι η συνάρτηση δεν έχει τιμή επιστροφής, καθώς και ότι τα ορίσματα είναι αυτά που φαίνονται στις παρενθέσεις. Αμέσως μετά, ακολουθούν οι δηλώσεις των μεταβλητών *i, i2, count* καθώς και η αρχικοποίηση της ακέραιας μεταβλητής *count*, η οποία παίρνει την τιμή 0. Οι μεταβλητές **i,i2** λειτουργούν ως μετρητές, ενώ η μεταβλητή **count** θα κρατά τα αποτελέσματά μας, δηλαδή το πόσα join γίνονται στους δύο πίνακες έχοντας συμπεριλάβει και τα join που έκαναν οι μεταβλητές που ήδη υπήρχαν όταν έχουν διαβαστεί.

```
for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
                if(A[i] == B[i2]){
                        count++;
                }
        }
}
```

Ο παραπάνω αλγόριθμος κάνει αναζήτηση των πινάκων Α και Β για να βρεί πόσες φορές σχετίζεται (είναι ίσο) οποιοδήποτε στοιχείο του πίνακα Α με οποιοδήποτε στοιχείο του πίνακα Β. Ο μετρητής μας εδώ εκφράζεται από την ακέραια μεταβλητή **count**.

```
count += *count1;
printf("Ta apotelesmata sou einai: %d join\n", count);
scanf("%d\n", &i);
}
```

Εδώ βλέπουμε ότι αυξάνουμε την μεταβλητή **count**, η οποία μετρούσε πόσες φορές σχετίζονται τα στοιχεία του πίνακα Α με τα στοιχεία του πίνακα Β, κατά την ακέραια τιμή που έχει η μεταβλητή **count1**. Η μεταβλητή count1, όπως είπαμε παραπάνω έχει την ακέραια τιμή που εκφράζει το πόσες φορές διαβάστηκε ένας ακέραιος, είτε για τον πίνακα Α είτε για τον πίνακα Β, δεν αποθηκεύτηκε και παράλληλα υπήρχε ο ίδιος ακέραιος στον άλλο πίνακα (δηλαδή, αν διαβάζαμε για τον πίνακα Α μία τιμή που ήδη υπήρχε στον πίνακα αυτόν και παράλληλα αυτή η τιμή υπήρχε στον πίνακα Β, τότε η ακέραια μεταβλητή count1 αυξανόταν κατά μία μονάδα). Στην ακριβώς επόμενη εντολή τυπόνονται τα αποτελέσματά μας. Η εντολή *scanf* την έχουμε βάλει διότι μόνο έτσι το πρόγραμμα **DEV C++** μας δείχνει τα αποτελέσματα σε κάθε πείραμα.

4.2 Ο αλγόριθμος TimeOfJoin

4.2.1 Εισαγωγή

Ο Αλγόριθμος **TimeOfJoin** είναι ο δεύτερος από τους τέσσερις που θα μελετήσουμε σε αυτό το έγγραφο. Όπως αναφέραμε παραπάνω, ο κάθε αλγόριθμος διαφέρει με τους άλλους στον τρόπο διαγραφής μιας καταχώρησης για να μπεί μία νέα. Έτσι και εδώ ο αλγόριθμος αυτός διαλέγει διαφορετικά τις ακέραιες τιμές, που έχουν καταχωρηθεί στον ένα ή στον άλλο πίνακα, για να αποχωρίσουν τον πίνακα και να εισαχθεί η νέα ακέραια τυχαία τιμή.

Κάθε φορά που εισέρχεται ένας νέος ακέραιος, ελέγχουμε αν κάνει join (αν υπάρχει δηλαδή η ίδια τιμή στον άλλο πίνακα). Για τον κάθε ακέραιο κρατάμε μία τιμή που εκφράζει χρονικά πότε έκανε join. Αυτός που έχει την μικρότερη τιμή, δηλαδή έκανε πιο παλιά join σε σχεση με τους άλλους, είναι αυτός που διαλέγει ο αλγόριθμος TimeOfJoin να διαγράψει για να εισαχθεί η νέα ακέραια τιμή.

4.2.2 Επεξήγηση αλγορίθμου

#include <stdio.h>

#include <stdlib.h>

#define NUMBERS 240

Όπως στον αλγόριθμο FIFO, έτσι και εδώ χρησιμοποιούμε τις τρεις παραπάνω εντολές για να έχουμε πρόσβαση στις βιβλιοθήκες **stdio.h** και **stdlib.h**, και επίσης δίνουμε στην σταθερά **NUMBERS** την τιμή 240.

void getStart(int * , int * , int * , int * , int , int , int * , int *);

int insertA(int * , int * , int * , int , int , int , int *);

int insertB(int * , int * , int * , int , int , int , int *);

void results(int * , int * , int , int , int *);

Εδώ έχουμε τις δηλώσεις των συναρτήσεων **getStart**, **insertA**, **insertB** και **results**. Η συνάρτηση getStart δεν έχει τιμή επιστροφής (μας το δηλώνει η λέξη-κλειδί **void**) και έχει οκτώ ορίσματα από τα οποία τα έξι είναι δείκτης σε ακέραιο (**int ***) και τα υπόλοιπα δύο είναι ακέραιος (**int**). Αντίστοιχα η συνάρτηση insertA και insertB επιστρέφουν ένα ακέραιο και έχουν επτά ορίσματα από τα οποία τα τέσσερα είναι δείκτης σε ακέραιο και τα υπόλοιπα τρία είναι ακέραιος. Τέλος, η συνάρτηση results δεν επιστρέφει τίποτα και τα ορίσματά του είναι πέντε, τα οποία απαρτίζονται από τρείς δείκτες σε ακέραιους και δύο ακέραιους.

main(){

int i , x , y , ta , tb , PA , PB , count1=0;

int *A , *A1 , *B , *B1;

ta = 1;

tb = 1;

Εδώ ξεκινά η συνάρτηση main. Αμέσως μετά δηλώνουμε τις μεταβλητές που θα χρησιμοποιήσουμε παρακάτω στη συνάρτηση αυτή, όπως φαίνεται παραπάνω στον αλγόριθμο.

Όπως και στον αλγόριθμο του αλγόριθμου FIFO, έτσι και εδώ η μεταβλητή **i** θα είναι χρήσιμη ως μετρητής στους διάφορους βρόχους επανάληψης της συνάρτησης main. Οι ακέραιες μεταβλητές **x** , **y** θα πάρουν και πάλι τιμές από τον χρήστη, κάτι που θα δούμε παρακάτω, και θα αποθηκεύσουν τον αριθμό των πλειάδων (ή αλλιώς θέσεων) των πινάκων Α και Β αντίστοιχα. Οι **PA** , **PB**, όπως και οι x,y, θα πάρουν τιμές από τον χρήστη και συμβολίζουν το πόσες φορές θα εκτελεστεί η συνάρτηση insertA και insertB αντίστοιχα, τις οποίες θα δούμε παρακάτω. Οι μεταβλητές **ta** , **tb** λειτουργούν ως «ρολόι» για τους πίνακες Α και Β αντίστοιχα. Και πάλι εδώ οι πίνακες **A1** και **B1** έχουν αντιστοιχία οι θέσεις τους με τις θέσεις των πινάκων **A** και **B**, όμως τώρα στην αρχή θα δούμε οτι οι πίνακες Α1 και Β1 θα πάρουν σε όλες τις θέσεις τους την τιμή -1. Έτσι για τον πίνακα Α, η μεταβλητή ta έχει αρχική τιμή 1 και ο πρώτος ακέραιος του πίνακα Α που κάνει join, η αντίστοιχη θέση στον πίνακα Α1 θα έχει την τιμή που έχει η μεταβλητή ta, και η μεταβλητή ta θα αυξάνεται κατά μία μονάδα. Ανα πάσα στιγμή δηλαδή θα ξέρουμε ποιος έκανε πιο παλιά join από τους άλλους, και είναι και αυτός που θα ψάχνουμε. Αντίστοιχα, για την μεταβλητή tb που αναφέρεται στον πίνακα Β. Έπειτα, στην επόμενη εντολή έχουμε την δημιουργία τεσσάρων δεικτών. Παρακάτω θα δούμε ότι κάθε ένας από αυτούς θα δεσμεύσει χώρο στη μνήμη ανάλογα με τις τιμές των x,y. Η μεταβλητή **count1** παίρνει την τιμή 0 και η δουλειά της είναι να μετρά τους αριθμούς που διαβάστηκαν και εκάναν join αλλά δεν ήταν κατάλληλοι για να εισαχθούν είτε στον πίνακα Α είτε στον πίνακα Β.

printf("Dose posa insertA theleis na exeis\n");

scanf("\n%d", &PA);

printf("Dose posa insertB theleis na exeis\n");

scanf("\n%d", &PB);

printf("Dose poses grames theleis na exei o pinakas A\n");

scanf("\n%d", &x);

printf("Dose poses grames theleis na exei o pinakas B\n");

scanf("\n%d", &y);

Σε αυτό το σημείο του αλγόριθμο ο χρήστης μας δίνει τιμές για να τις δώσουμε στις μεταβλητές PA, PB, x και y αντίστοιχα.

A = (int *)malloc(x * sizeof(int));

A1 = (int *)malloc(x * sizeof(int));

B = (int *)malloc(y * sizeof(int));

B1 = (int *)malloc(y * sizeof(int));

Με τη βοήθεια της συνάρτησης malloc στις παραπάνω τέσσερις εντολές δεσμεύουμε χώρο στη μνήμη. Στους Α και Α1 δεσμεύουμε χώρο όσο x ακέραιοι και στους πίνακες Β και Β1 όσο y ακέραιοι.

getStart(A , A1 , B , B1 , x , y , &ta , &tb);

Εδώ καλούμε την συνάρτηση getStart με ορίσματα τους δείκτες Α, Α1, Β, Β1, &ta και &tb καθώς και τους ακέραιους x,y.

```
for(i=x; i<PA; i++){
    ta = insertA(A , A1 , B , x , y , ta , &count1);
}
for(i=y; i<PB; i++){
    tb = insertB(B , B1 , A , x , y , tb , &count1);
}
```

Όπως και στον αλγόριθμο FIFO, έτσι και εδώ κάθε φορά που εκτελούνται οι εντολές που βρίσκονται μέσα στο block εντολών της for, καλείται η συνάρτηση **insertA**, η οποία έχει τιμή επιστροφής έναν **int** (ακέραιο) και ορίσματα τους δείκτες *A, A1, B, &count1* καθώς και τους ακέραιους *x, y, ta* όπως φαίνεται παραπάνω.

Η τιμή επιστροφής της συνάρτησης αποθηκεύεται στην ακέραια μεταβλητή **ta**, η οποία στέλνεται ως ορισμά σε κάθε κάλεσμα της συνάρτησης όπως και τα υπόλοιπα ορίσματα. Η συνάρτηση θα καλεσθεί όσες φορές ισχύει η συνθήκη της for.

Ο μετρητής **i** ξεκινά από την θέση **x** διότι έχουμε ήδη δώσει x τιμές στους πίνακες (από την συνάρτηση *getStart*), άρα οι τιμές που μένουν να δώσουμε, αφού συνολικά είναι **PA** τιμές, είναι PA-x, ή αλλιώς από το x μέχρι το PA.

Αντίστοιχα λειτουργεί και η επόμενη for.

results(A , B , x , y , &count1);

}

Στην παραπάνω εντολή, καλούμε την συνάρτηση **results**, η οποία έχει ως ορίσματα τους δείκτες *A, B, &count1* καθώς και τους ακέραιους *x, y* με την σειρά που φαίνεται παραπάνω. Τιμή επιστροφής δεν έχει, και η δουλειά της είναι να κάνει τις μετρήσεις των εκάστοτε join που έχουν οι δύο πίνακες (A,B) των τυχαίων ροών και να τις εμφανίσει στον χρήστη. Το άγκιστρο κλεισίματος (}) αναφέρεται στην συνάρτηση main(), και μας δηλώνει ότι σε εκείνο το σημείο σταματάνε οι εντολές της.

Στη συνέχεια θα αναλύσουμε τις συναρτήσεις τις οποίες καλέσαμε κατα την διάρκεια εκτέλεσης της main().

void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y , int *ta , int *tb){

int i , i2 , a , b , c=0;

A[0] = rand()% NUMBERS+1;

B[0] = rand()% NUMBERS+1;

Εδώ ξεκινά η υλοποίηση της συνάρτησης **getStart.** Τιμή επιστροφής δεν έχει, όπως προαναφέραμε, και αυτό δηλώνεται με την λέξη-κλειδί **void** πριν το όνομα της συνάρτησης. Αμέσως μετά, μέσα στις παρενθέσεις, βλέπουμε ότι η συνάρτηση δημιουργεί, και δηλώνει ταυτόχρονα, μεταβλητές για να αποθηκεύσει τις τιμές των ορισμάτων. Στη συνέχεια, δημιουργούμε τις ακέραιες μεταβλητές *i, a, b, c, i2* και δίνουμε αρχική τιμή στην μεταβλητή c την τιμή 0.

Η μεταβλητή **i** θα λειτουργήσει ως μετρητής στους βρόχους επαναλήψεων, όπως και η μεταβλητή **i2**. Στη μεταβλητή **a** θα δίνουμε μία τυχαία τιμή κάθε φορά, και αφού περάσει τους κατάλληλους ελέγχους και δούμε ότι είναι κατάλληλη για να εισαχθεί στον πίνακα A τότε θα δίνει την τιμή της στην εκάστοτε θέση του πίνακα A. Αντίστοιχα θα λειτουργεί και η μεταβλητή **b** για τον πίνακα B. Η μεταβλητή **c** παίρνει τιμές στο προγραμμά μας 0 και 1. Την χρησιμοποιούμε για να καταλαβαίνουμε αν ο compiler έχει μπεί σε block εντολών της if.

Αμέσως μετά, δίνουμε αρχικές τιμές στις πρώτες θέσεις των πινάκων Α και Β αντίστοιχα. Τις τιμές τις δίνουμε με την βοήθεια της συνάρτησης **rand()** η οποία σε συνδιασμό με το υπόλοιπο της εντολής (δηλαδή, η εντολή rand()% NUMBERS+1;) θα μας δώσει αποτέλεσμα έναν (τυχαίο) ακέραιο αριθμό στην γκάμα 0-240 (η γκάμα είναι έως 240 διότι η σταθερά NUMBERS έχει την τιμή αυτή).

```
for(i=1; i<x; i++){
    a = rand()% NUMBERS+1;
    for(i2=0; i2<i; i2++){
        if(A[i2] == a){
            i--;
            c=1;
            break;
        }
    }
```

Στον παραπάνω αλγόριθμο ελέγχουμε για κάθε τυχαία τιμή που διαβάζουμε και αποθηκεύουμε στην μεταβλητή **a** αν είναι κατάλληλη για να εισαχθεί στον πίνακα Α. Όπως έχουμε πει και σε άλλο σημείο του εγγράφου αυτού, θέλουμε τα στοιχεία του πίνακα Α, καθώς και του πίνακα Β, να είναι μοναδικά. Έτσι, για κάθε τυχαία τιμή που διαβάζουμε ψάχνουμε ολόκληρο τον πίνακα Α να δούμε αν υπάρχει ξανά αυτή η τιμή. Αν υπάρχει (**if(A[i2] == a)**) τότε μειώνουμε τον μετρητή έτσι ώστε την επόμενη φορά να ψάχνουμε ακέραιο για την ίδια θέση, και δίνουμε την τιμή 1 στην μεταβλητή **c**. Αμέσως μετά φτάνουμε στην εντολή **break** που θα μας βγάλει από την εμφωλευμένη for και θα σταματίσουμε να ψάχνουμε τον πίνακα Α καθώς θα βάλουμε στην επόμενη επανάληψη της εξωτερικής for άλλη τιμή στην μεταβλητή a.

Προσοχή, αν φτάσουμε στην εντολή break, βγαίνουμε από την εμφωλευμένη for και όχι από την αρχική.

```
        if(c == 0){
            A[i] = a;
        }
        c = 0;
    }
    c = 0;
```

Αμέσως μετά, ελέγχουμε αν η μεταβλητή c έχει την τιμή 0 (**if(c == 0)**). Υπενθυμίζουμε ότι η μεταβλητή c έχει τιμή 1 αν έχει γίνει αληθές η προηγούμενη συνθήκη της if. Με άλλα λόγια η ακέραια μεταβλητή c έχει τιμή 0 αν η τιμή που έχει η μεταβλητή a δεν υπάρχει στον υπάρχον πίνακα A, και έχει τιμή 1 αν ισχύει το αντίθετο. Αν λοιπόν η μεταβλητή c έχει τιμή 0, ή αλλιώς αν η τιμή της μεταβλητής a είναι κατάλληλή για να εισέλθει στον πίνακα A, τότε δίνουμε την τιμή που αντιπροσωπεύει η ακέραια μεταβλητή a στην εκάστοτε θέση του πίνακα A (**A[i]=a**). Στη συνέχεια, μηδενίζουμε την μεταβλητή c για την επόμενη επανάληψη διότι θέλουμε να έχει default τιμή 0. Βγαίνοντας από την εξωτερική for μηδενίζουμε και πάλι την μεταβλητή c διότι παρακάτω θα μας χρειαστεί και, όπως προαναφέραμε, θέλουμε να έχει default τιμή 0.

```
    for(i=0; i<x; i++){
        A1[i] = -1;
    }
    for(i=0; i<y; i++){
        B1[i] = -1;
    }
```

Στο αρχικό στάδιο επεξήγησης του αλγόριθμου FIFO αναφερθήκαμε στο ότι οι πίνακες A1 και B1 θα έχουν τιμές -1 σε όλες τις θέσεις τους αρχικά. Στις δύο παραπάνω επαναλήψεις πραγματοποιείται αυτό. Σε αυτό το σημείο θέλουμε να σας αναφέρουμε ότι αν κάποια τιμή μπεί στον πίνακα A ή B και δεν κάνει join τότε η αντίστοιχη θέση στον πίνακα A1 ή B1 θα παραμείνει η τιμή, δηλαδή θα έχει τιμή -1, μικρότερη δηλαδή από οποιαδήποτε άλλη μεταβλητή κάνει join.

```
for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
                if(B[i2] == b){
                        i--;
                        c=1;
                        break;
                }
        }
        if(c == 0){
            B[i] = b;
        }
        c = 0;
}
```

Στον παραπάνω αλγόριθμο γεμίζουμε τον πίνακα Β, με τους κατάλληλους τυχαίους ακέραιους αριθμούς στην αντίστοιχη γκάμα, ακριβώς με τον ίδιο τρόπο που γεμίσαμε και τον πίνακα Α.

```
for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
                if(A[i] == B[i2]){
                        A1[i] = *ta;
                        *ta += 1;
                        break;
                }
```

```
        }
    }
```

Έπειτα, ελέγχουμε κάθε στοιχείο του πίνακα Α με όλα τα στοιχεία του πίνακα Β, με την βοήθεια των δύο for. Αν κάποιο στοιχείο κάνει join τότε σταματάμε να ψάχνουμε τα υπόλοιπα στοιχεία του Β (με την εντολή **break**) και δίνουμε στην αντίστοιχη θέση του στον πίνακα Α1 τα περιεχόμενα του δείκτη **ta** (ή αλλιώς την τιμή της μεταβλητής ta) η οποία αμέσως μετά αυξάνεται. Στο τέλος των δύο for αυτών ο πίνακας Α1 έχει κρατήσει την σειρά όπου οι τιμές του πίνακα Α έκαναν join. Αν κάποια τιμή δεν έκανε join τότε ο πίνακας Α1 έχει στην αντίστοιχη θέση του την τιμή -1.

```
    for(i2=0; i2<y; i2++){
        for(i=0; i<x; i++){
            if(B[i2] == A[i]){
                B1[i2] = *tb;
                *tb += 1;
                break;
            }
        }
    }
}
```

Εδώ γίνεται ακριβώς η ίδια δουλειά με πριν, αλλά για τον πίνακα Β. Το τελευταίο άγκιστρο κλεισίματος είναι της συνάρτησης getStart και μας δηλώνει ότι εδώ τελειώνουν οι εντολές της.

```
int insertA(int *A , int *A1 , int *B , int x , int y , int ta , int *count1){
    int i , i2 , ak , pos , a;
```

Η τιμή επιστροφής της συνάρτησης είναι ένας ακέραιος. Το όνομα της συνάρτησης είναι **insertA** και έχει ως ορίσματα τους δείκτες *A, A1, B, count1* καθώς και τους ακέραιους *x, y, ta*. Αμέσως μετά δηλώνουμε τις μεταβλητές που θα χρησιμοποιήσουμε παρακάτω.

Οι μεταβλητές **i**,**i2** έχουν τον ρόλο του μετρητή. Την μεταβλητή **ak** θα την έχουμε ως μέτρο σύγκρισης για να βρίσκουμε κάθε φορά την ελάχιστη ακέραια τιμή στον πίνακα Α1. Στην μεταβλητή **a** θα δίνουμε μια τυχαία ακέραια τιμή στην αντίστοιχη γκάμα που θα βρισκόμαστε κάθε φορά για να γίνει η νέα καταχώρηση. Ο ακέραιος **pos** θα αποθηκεύει την θέση στην οποία βρήκαμε την ελάχιστη τιμή του πίνακα Α1.

```
a = rand()% NUMBERS+1;
```

Στην παραπάνω εντολή δίνουμε μια τυχαία ακέραια τιμή στην γκάμα 0-240 στην μεταβλητή a.

```
for(i=0; i<x; i++){
    if(A[i] == a){
        for(i=0; i<y; i++){
            if (B[i] == a){
              *count1 += 1;
              break;
            }
        }
        a = rand()% NUMBERS+1;
        i = -1;
    }
}
```

Έπειτα, ελέγχουμε ένα-ένα όλα τα στοιχεία του πίνακα Α για να δούμε αν η τιμή που έχουμε στην μεταβλητή a υπάρχει ήδη στον πίνακα Α (**if(A[i] == a**). Αν υπάρχει σημαίνει ότι δεν

θα καταχωρηθεί στον πίνακα Α. Όμως μας ενδιαφέρει για τα αποτελέσματά μας να δούμε αν η ήδη υπάρχουσα τιμή στον πίνακα a κάνει join με κάποιο στοιχείο του πίνακα Β. Παρόλο που δεν θα καταχωρηθεί, αν κάνει join θα μετρήσει στα αποτελέσματά μας καθώς η μεταβλητή count1 (η οποία μετρά πόσες τιμές που δεν καταχωρήθηκαν έκαναν join) θα προστεθεί στα αποτελέσματα. Είτε κάνει join είτε όχι, η μεταβλητή a παίρνει μία νέα τυχαία ακέραια τιμή στην γκάμα που βρισκόμαστε και ο μετρητής i παίρνει την τιμή -1. Έτσι ο compiler γυρνόντας στην αρχική for, αυξάνει τον μετρητή i και γίνεται 0 και ξεκινά η διαδικασία αναζήτησης του κατάλληλου τυχαίου ακεραίου από την αρχή.

Αν η αρχική τυχαία ακέραια τιμή που δώσαμε στην μεταβλητή a δεν υπάρχει στον πίνακα Α, τότε ο compiler μετά από x φορές που θα ελένξει την συνθήκη της πρώτης if, προχορά χωρίς να κάνει την παραπάνω διαδικασία.

```
ak = A1[0];
pos = 0;
for(i=1; i<x; i++){
    if (A1[i]<ak){
      ak = A1[i];
      pos = i;
    }
}
```

Στον αλγόριθμο που βλέπουμε παραπάνω, δίνουμε στην μεταβλητή ak την τιμή που έχει η πρώτη θέση του πίνακα Α1 (**ak=A1[0]**), και στην μεταβλητή **pos** την τιμή 0. Στη συνέχεια ελέγχουμε όλα τα υπόλοιπα στοιχεία του πίνακα Α1 (εκτός το πρώτο στοιχείο, αφού και η αρχική τιμή του μετρητή i η τιμή 1) αν είναι μικρότερα της μεταβλητής ak. Αν είναι κάποιο, τότε την τιμή αυτή την δίνουμε στην μεταβλητή ak και η μεταβλητή pos κρατά την θέση που βρίκαμε την τιμή αυτή. Στο τέλος της for η μεταβλητή ak έχει την μικρότερη τιμή του πίνακα Α1 και η μεταβλητή pos κρατά την θέση που βρέθηκε. Άρα στην pos θέση του πίνακα Α πρέπει να καταχωρήσουμε την τιμή που έχει η μεταβλητή a, ή αλλιώς η τιμή που υπάρχει στην pos θέση του πίνακα Α είναι αυτή που θα αντικατασταθεί από την νέα τυχαία ακέραια τιμή που βρίσκεται στην μεταβλητή a.

```
    A[pos] = a;
    A1[pos] = -1;
    for(i=0; i<y; i++){
        if(A[pos] == B[i]){
            A1[pos] = ta;
            ta += 1;
            break;
        }
    }
    return ta;
}
```

Είμαστε έτοιμοι λοιπόν να δώσουμε στην θέση που βρέθηκε ο ακέραιος, ο οποίος πιο παλιά έκανε join από τους άλλους, την νέα τυχαία ακέραια τιμή και αυτό γίνεται με την εντολή **A[pos]=a**. Στην αντίστοιχη θέση στον πίνακα A1 δίνουμε την τιμή -1 καθώς δεν γνωρίζουμε αν η νέα τιμή κάνει join με κάποιο στοιχείο του πίνακα B. Στην βρόχο for ελέγχουμε όλα τα στοιχεία του πίνακα B με τη νέα τιμή και άν υπάρχει σύνδεση (δηλαδή αν η νέα τιμή υπάρχει στον πίνακα B) τότε η αντίστοιχη θέση στον πίνακα A1 (**A1[pos]**) παίρνει την τιμή που έχει η μεταβλητή ta, η οποία μας δηλώνει την χρονική σειρά των join που έχουν κάνει τα αντίστοιχα στοιχεία του πίνακα A. Αμέσως μετά, η μεταβλητή ta παίρνει την καθιερωμένη αύξηση και σταματάμε να ψάχνουμε τον πίνακα B με την βοήθεια της μεταβλητής **break**. Τέλος, η συνάρτηση επιστρέφει την τιμή της μεταβλητής ta.

```
int insertB(int *B , int *B1 , int *A , int x , int y , int tb , int *count1){
    int i , i2 , ak , pos , b;
    b = rand()% NUMBERS+1;
    for(i=0; i<y; i++){
        if(B[i] == b){
```

```
            for(i=0; i<x; i++){
                  if (A[i] == b){
                    *count1 += 1;
                    break;
                  }
            }
            b = rand()% NUMBERS+1;
            i = -1;
      }
}
ak = B1[0];
pos = 0;
for(i=1; i<y; i++){
          if (B1[i]<ak){
            ak = B1[i];
            pos = i;
          }
}
B[pos] = b;
B1[pos] = -1;
for(i=0; i<x; i++){
      if(B[pos] == A[i]){
              B1[pos] = tb;
              tb += 1;
```

```
            break;
        }
    }
    return tb;
}
```

Παραπάνω βλέπουμε την συνάρτηση insertB, η οποία λειτουργεί με το ίδιο σκεπτικό όπως η συνάρτηση insertA, για να αντικαταστήσει όσες φορές καλεστεί με νέα κατάλληλα στοιχεία τον πίνακα B και B1.

```
void results(int *A ,  int *B , int x , int y , int *count1){
    int i , i2 , count;
    count = 0;
```

Σε αυτό το σημείο του αλγόριθμο ξεκινά ο ορισμός της συνάρτησης results, η οποία δεν έχει τιμή επιστροφής και τα ορίσματά της τα βλέπουμε στις παρενθέσεις. Στη συνέχεια δημιουργούμε τις μεταβλητές *i,i2,count*. Οι μεταβλητές i,i2 θα λειτουργήσουν ως μετρητές, ενώ η μεταβλητή count, η οποία στη συνέχεια παίρνει αρχική τιμή 0, θα μετρήσει τα αποτελέσματά μας, δηλαδή πόσες φορές κάνουν join τα στοιχεία του πίνακα A με τα στοιχεία του πίνακα B.

```
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if(A[i] == B[i2]){
                count++;
            }
        }
    }
```

Στον παραπάνω αλγόριθμο, ελέγχουμε όλα τα στοιχεία του πίνακα Α με όλα τα στοιχεία του πίνακα Β και κάθε φορά που βλέπουμε ότι μία τιμή του ενός πίνακα υπάρχει στον άλλο, η μεταβλητή count αυξάνεται κατά μία μονάδα. Στο τέλος των επαναλήψεων η μεταβλητή count εκφράζει τα join που γίνονται μεταξύ των δύο πινάκων.

```
    count += *count1;

    printf("Ta apotelesmata sou einai: %d join\n", count);

    scanf("%d\n", &i);

}
```

Υπενθυμίζουμε ότι η μεταβλητή count1 συγκρατεί τον αριθμό των ακεραίων που έχουν διαβαστεί, υπήρχαν οι τιμές τους στον πίνακα για τον οποίο διαβάστηκαν (είτε για τον Α είτε για τον Β), και η τιμή αυτών υπήρχε στον άλλο πίνακα, δηλαδή έκαναν join. Έτσι στα τελικά μας αποτελέσματα συμπεριλαμβάνουμε όχι μόνο τα join που κάνουν οι τελικοί πίνακες, αλλά και τα στοιχεία που έχουν διαβαστεί και θα έκαναν join αλλά δεν καταχωρήθηκαν. Για αυτόν τον λόγο στην μεταβλητή count προσθέτουμε την τιμή της μεταβλητής count1 (πιο σωστά προσθέτουμε τα περιεχόμενα του δείκτη count1) με την βοήθεια της εντολής **count+=*count1**. Αμέσως μετά εμφανίζουμε τα αποτελέσματά μας στον χρήστη. Η εντολή *scanf* την έχουμε βάλει διότι μόνο έτσι το πρόγραμμα **DEV C++** μας δείχνει τα αποτελέσματα σε κάθε πείραμα.

4.3 Ο αλγόριθμος LFU (Less Frequency Used)

4.3.1 Εισαγωγή

Ο Αλγόριθμος **LFU** είναι ο τρίτος από τους τέσσερις που θα μελετήσουμε σε αυτό το έγγραφο. Όπως αναφέραμε παραπάνω, ο κάθε αλγόριθμος διαφέρει με τους άλλους στον τρόπο διαγραφής μιας καταχώρησης για να μπεί μία νέα. Έτσι και εδώ ο αλγόριθμος αυτός διαλέγει διαφορετικά τις ακέραιες τιμές, που έχουν καταχωρηθεί στον ένα ή στον άλλο πίνακα, για να αποχωρίσουν τον πίνακα και να εισαχθεί η νέα ακέραια τυχαία τιμή.

Κάθε φορά που εισέρχεται ένας νέος ακέραιος, ελέγχουμε αν κάνει join (αν υπάρχει δηλαδή η ίδια τιμή στον άλλο πίνακα). Για τον κάθε ακέραιο κρατάμε πόσα join κάνει. Αυτός που έχει την μικρότερη τιμή, δηλαδή αυτός που έχει κάνει τα λιγότερα join, είναι αυτός που διαλέγει ο αλγόριθμος LFU να διαγράψει για να εισαχθεί η νέα ακέραια τιμή.

4.3.2 Επεξήγηση αλγορίθμου

#include <stdio.h>

#include <stdlib.h>

#define NUMBERS 240

Ο αλγόριθμος εδώ μας δηλώνει ότι θα έχουμε στη διάθεσή μας τις βιβλιοθήκες **stdio.h** και **stdlib.h** καθώς και ότι η σταθερά **NUMBERS** θα αντιπροσωπεύει την τιμή 240.

void getStart(int * , int * , int * , int * , int , int);

void insertA(int * , int * , int * , int * , int , int , int *);

void insertB(int * , int * , int * , int * , int , int , int *);

void results(int * , int , int *);

Στο αμέσως επόμενο κομμάτι του αλγόριθμο, βλέπουμε τις δηλώσεις των συναρτήσεων **getStart**, **insertA**, **insertB** και **results**, οι οποίες δεν επιστρέφουν τίποτα και έχουν τα ορίσματα όπως φαίνονται παραπάνω.

main(){

int i , x , y , PA , PB , count1=0;

int *A , *B , *A1 , *B1;

Έπειτα, ξεκινά η συνάρτηση main() της οποίας οι πρώτες εντολές είναι δηλώσεις μεταβλητών. Στην πρώτη γραμμή αλγόριθμο της συνάρτησης έχουμε την δημιουργία των *i,x,y,PA,PB* και *count1*. Η μεταβλητή **i** θα λειτουργεί ως μετρητής στους διάφορους βρόχους που θα εκτελέσουμε παρακάτω. Οι μεταβλητές **x** , **y** θα πάρουν τιμές από τον χρήστη και θα αντιπροσωπεύουν τον αριθμό των θέσεων που θέλουμε να έχουν οι πίνακες Α και Β αντίστοιχα. Επίσης, οι μεταβλητές **PA** , **PB** θα πάρουν και αυτές τιμές από τον χρήστη, και θα συγκρατούν τον αριθμό που θα εκτελεστούν οι συναρτήσεις insertA και insertB αντίστοιχα. Οι συναρτήσεις insertA και insertB καθορίζουν πόσες φορές θα διαβάσουμε ένα στοιχείο για τον πίνακα Α ή Β, αντίστοιχα, για να το εισάγουμε σε αυτούς. Η μεταβλητή **count1** παίρνει την τιμή

0 και η δουλειά της είναι να μετρά τους αριθμούς που διαβάστηκαν και εκάναν join αλλά δεν ήταν κατάλληλοι για να εισαχθούν είτε στον πίνακα Α είτε στον πίνακα Β.

```
printf("Dose posa insertA theleis na exeis\n");

scanf("\n%d", &PA);

printf("Dose posa insertB theleis na exeis\n");

scanf("\n%d", &PB);

printf("Dose poses grames theleis na exei o pinakas A\n");

scanf("\n%d", &x);

printf("Dose poses grames theleis na exei o pinakas B\n");

scanf("\n%d", &y);
```

Εδώ διαβάζουμε τιμές από τον χρήστη και τις δίνουμε στις μεταβλητές PA, PB, x και y αντίστοιχα, με την βοήθεια των εντολών *printf* και *scanf*.

```
A = (int *)malloc(x * sizeof(int));

A1 = (int *)malloc(x * sizeof(int));

B = (int *)malloc(y * sizeof(int));

B1 = (int *)malloc(y * sizeof(int));

getStart(A , A1 , B , B1 , x , y);
```

Αμέσως μετά με τη συνάρτηση malloc στις παραπάνω πρώτες τέσσερις εντολές δεσμεύουμε χώρο στη μνήμη. Στους Α και Α1 δεσμεύουμε χώρο όσο x ακέραιοι και στους πίνακες Β και Β1 όσο y ακέραιους. Στην τελευταία εντολή καλούμε την συνάρτηση getStart με τα ορίσματα όπως φαίνεται στις παρενθέσεις.

```
    for(i=x; i<PA; i++){
        insertA(A , A1 , B , B1 , x , y , &count1);
    }
    for(i=y; i<PB; i++){
        insertB(B , B1 , A , A1 , x , y , &count1);
    }
    results(A1 , x , &count1);
}
```

Τέλος, η main() καλεί, με την βοήθεια δύο for, PA-x φορές τη συνάρτηση insertA και PB-y φορές την συνάρτηση insertB. Δεν καλεί τις συναρτήσεις PA και PB φορές αντίστοιχα διότι ο πίνακας A έχει πάρει ήδη x στοιχεία από την συνάρτηση getStart, όπως και ο πίνακας B, y στοιχεία. Αμέσως μετά καλείται η συνάρτηση results η οποία θα μας εμφανίσει τα αποτελέσματά μας.

```
void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y){
    int i , i2 , c=0 , a , b;
```

Εδώ ξεκινά ο ορισμός της συνάρτησης getStart. Η λέξη-κλειδί **void** μας δηλώνει ότι δεν επιστρέφει τίποτα η συνάρτηση. Τα ορίσματα της συνάρτησης αποτελούνται από τέσσερις δείκτες (A, A1, B, B1) και δύο ακέραιους (x, y). Έπειτα, δημιουργούμε τις μεταβλητές i, i2, c, a, b.

Οι μεταβλητές **i**,**i2** θα λειτουργήσουν ως μετρητές στους διάφορους βρόχους που θα χρησιμοποιήσουμε, οι μεταβλητές **a**,**b** θα αποθηκεύουν τις εκάστοτε τυχαίες ακέραιες τιμές που θα διαβάζουμε και η μεταβλητή **c** θα παίρνει τιμές 0 ή 1 και θα μας βοηθάει στο να καταλαβαίνουμε αν ο compiler μπήκε σε διάφορα block εντολών. Επίσης, δίνουμε αρχική τιμή στην μεταβλητή c την τιμή 0.

```
A[0] = rand()% NUMBERS+1;

B[0] = rand()% NUMBERS+1;
```

Εδώ δίνουμε αρχικές τιμές στις πρώτες θέσεις των πινάκων Α και Β αντίστοιχα. Τις τιμές τις δίνουμε με την βοήθεια της συνάρτησης **rand()** η οποία σε συνδιασμό με το υπόλοιπο της εντολής (δηλαδή, η εντολή rand()% NUMBERS+1;) θα μας δώσει αποτέλεσμα έναν (τυχαίο) ακέραιο αριθμό στην γκάμα 0-240 (η γκάμα είναι έως 240 διότι η σταθερά NUMBERS έχει την τιμή αυτή).

```
for(i=1; i<x; i++){
    a = rand()% NUMBERS+1;
    for(i2=0; i2<i; i2++){
        if(A[i2] == a){
            i--;
            c=1;
            break;
        }
    }
```

Όπως και στους προηγούμενους αλγόριθμους, έτσι και εδώ ελέγχουμε για κάθε τυχαία τιμή που διαβάζουμε και αποθηκεύουμε στην μεταβλητή **a** αν είναι κατάλληλη για να εισαχθεί στον πίνακα Α. Όπως έχουμε πει και παραπάνω, θέλουμε τα στοιχεία του πίνακα Α, καθώς και του πίνακα Β, να είναι μοναδικά. Έτσι για κάθε τυχαία τιμή που διαβάζουμε ψάχνουμε ολόκληρο τον πίνακα Α να δούμε αν υπάρχει ξανά αυτή η τιμή. Αν υπάρχει (**if(A[i2] == a)**) τότε μειώνουμε τον μετρητή έτσι ώστε την επόμενη φορά να ψάχνουμε ακέραιο για την ίδια θέση, και δίνουμε την τιμή 1 στην μεταβλητή **c**. Αμέσως μετά φτάνουμε στην εντολή **break** που θα μας βγάλει από την εμφωλευμένη for και θα σταματίσουμε να ψάχνουμε τον πίνακα Α καθώς θα βάλουμε στην επόμενη επανάληψη της εξωτερικής for άλλη τιμή στην μεταβλητή a.

Προσοχή, αν φτάσουμε στην εντολή break, βγαίνουμε από την εμφωλευμένη for και όχι από την αρχική.

```
        if(c == 0){
            A[i] = a;
        }
        c = 0;
    }
    c = 0;
```

Επίσης, ελέγχουμε αν η μεταβλητή c έχει την τιμή 0 (**if(c == 0)**). Υπενθυμίζουμε ότι η μεταβλητή c έχει τιμή 1 αν έχει γίνει αληθές η προηγούμενη συνθήκη της if. Με άλλα λόγια, η μεταβλητή c έχει τιμή 0 αν η τιμή που έχει η μεταβλητή a δεν υπάρχει στον υπάρχον πίνακα A, και έχει τιμή 1 αν ισχύει το αντίθετο. Αν λοιπόν η μεταβλητή c έχει τιμή 0, ή αλλιώς αν η τιμή της μεταβλητής a είναι κατάλληλή για να εισέλθει στον πίνακα A, τότε δίνουμε την τιμή που αντιπροσωπεύει η ακέραια μεταβλητή a στην εκάστοτε θέση του πίνακα A (**A[i]=a**). Στη συνέχεια, μηδενίζουμε την μεταβλητή c για την επόμενη επανάληψη διότι θέλουμε να έχει default τιμή 0. Βγαίνοντας από την εξωτερική for μηδενίζουμε και πάλι την μεταβλητή c διότι παρακάτω θα μας χρειαστεί και, όπως προαναφέραμε, θέλουμε να έχει default τιμή 0.

```
    for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(B[i2] == b){
                i--;
                c=1;
                break;
            }
        }
        if(c == 0){
            B[i] = b;
```

```
        }
        c = 0;
    }
```

Εδώ γίνεται αντίστοιχη δουλειά για τον πίνακα B.

```
    for(i=0; i<x; i++){
        A1[i] = 0;
    }
    for(i=0; i<y; i++){
        B1[i] = 0;
    }
```

Όπως έχουμε πεί και πιο πρίν, οι πίνακες A1 και B1 κρατούν σε αντιστοιχία στοιχεία για τις τιμές των πινάκων A και B. Συγκεκριμένα, κρατούν το πόσα join κάνει η αντίστοιχη θέση στον αντίστοιχο πίνακα. Για παράδειγμα, αν η τιμή που βρίσκεται στην $1^{η}$ θέση του πίνακα A (A[0]) κάνει 1 join, τότε A1[0]=1. Για αυτό το λόγο, σε αυτό το σημείο του αλγόριθμο μηδενίζουμε όλα τα στοιχεία των πινάκων A1 και B1 έτσι ώστε να βρούμε παρακάτω τον αριθμό των join της κάθε θέσης και να τον αποθηκεύσουμε στην εκάστοτε θέση στους πίνακες A1 και B1.

```
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if(A[i] == B[i2]){
                A1[i] += 1;
                B1[i2] += 1;
            }
        }
```

```
    }
}
```

Στον παραπάνω αλγόριθμο ελέγχουμε όλα τα στοιχεία του πίνακα Α με όλα τα στοιχεία του πίνακα Β. Όποτε βρίσκουμε κοινά στοιχεία, αυξάνουμε την αντίστοιχη θέση στους πίνακες Α1 και Β1 έτσι ώστε στο τέλος των επαναλήψεων οι θέσεις των πινάκων Α1 και Β1 να συγκρατούν τα join της κάθε θέσης και των δύο πινάκων (Α και Β).

```
void insertA(int *A , int *A1 , int *B , int *B1 , int x, int y , int *count1){
    int i , i2 , ak , pos , a;
    a = rand()% NUMBERS+1;
```

Εδώ ξεκινά ο ορισμός της συνάρτησης insertA. Τιμή επιστροφής δεν έχει και τα ορίσματά της είναι όπως φαίνονται στις παρενθέσεις. Αμέσως μετά δηλώνουμε τις μεταβλητές i,i2,ak,pos και a.

Όπως και στους υπόλοιπους αλγόριθμους, έτσι και εδώ οι μεταβλητές **i**,**i2** έχουν τον ρόλο του μετρητή. Την μεταβλητή **ak** θα την έχουμε ως μέτρο σύγκρισης για να βρίσκουμε κάθε φορά την ελάχιστη ακέραια τιμή στον πίνακα Α1. Στην μεταβλητή **a** θα δίνουμε μια τυχαία ακέραια τιμή στην αντίστοιχη γκάμα που θα βρισκόμαστε κάθε φορά για να γίνει η νέα καταχώρηση. Ο ακέραιος **pos** θα αποθηκεύει την θέση στην οποία βρήκαμε την ελάχιστη τιμή του πίνακα Α1.

Η μεταβλητή a, λοιπόν, παίρνει μια τυχαία ακέραια τιμή στην γκάμα που βρισκόμαστε, 0-240.

```
for(i=0; i<x; i++){
        if(A[i] == a){
            for(i=0; i<y; i++){
                if (B[i] == a){
                    *count1 += 1;
                    break;
```

```
            }
        }
        a = rand()% NUMBERS+1;
        i = -1;
    }
}
```

Αμέσως μετά, ψάχνουμε όλο τον πίνακα Α για να δούμε αν υπάρχει η τιμή της μεταβλητής a σε αυτόν. Αν υπάρχει (**if(A[i]==a**) τότε ψάχνουμε ένα-ένα τα στοιχεία του πίνακα Β για να δούμε αν κάνει join. Αν κάνει (**if(B[i]==a**) τότε αυξάνουμε την μεταβλητή count1 (η οποία αθροίζεται στο τέλος με τα συνολικά join) και σταματάμε το ψάξιμο του πίνακα Β (με τη βοήθεια της εντολής **break**). Μετά δίνουμε στην μεταβλητή a μία νέα τυχαία ακέραια τιμή στην γκάμα μας και δίνουμε την τιμή -1 στην μεταβλητή i, έτσι ώστε μόλις γυρίσει ο compiler στη for και πάρει ο μετρητής την αύξηση, θα γίνει 0 και θα ξανα ξεκινήσει το ψάξιμο του πίνακα Α να δεί αν υπάρχει σε αυτόν η νέα τιμή. Στο τέλος των επαναλήψεων αυτών, η μεταβλητή a έχει μία τυχαία ακέραια τιμή η οποία δεν υπάρχει σε καμία θέση του πίνακα Α.

```
ak = A1[0];
pos = 0;
for(i=1; i<x; i++){
    if (A1[i]<ak){
      ak = A1[i];
      pos = i;
    }
}
```

Έπειτα, δίνουμε στην μεταβλητή ak την τιμή που έχει η πρώτη θέση του πίνακα Α1 (**ak=A1[0]**), και στην μεταβλητή **pos** την τιμή 0. Στη συνέχεια, ελέγχουμε όλα τα υπόλοιπα στοιχεία του πίνακα Α1 (εκτός το πρώτο στοιχείο, αφού και η αρχική τιμή του μετρητή i η τιμή 1) αν είναι μικρότερα της μεταβλητής ak. Αν είναι κάποιο, τότε την τιμή αυτή την δίνουμε στην μεταβλητή ak και η μεταβλητή pos κρατά την θέση που βρίκαμε την τιμή αυτή. Στο τέλος της

for η μεταβλητή ak έχει την μικρότερη τιμή του πίνακα A1 και η μεταβλητή pos κρατά την θέση που βρέθηκε. Άρα, η τιμή που υπάρχει στην pos θέση του πίνακα A είναι αυτή που θα αντικατασταθεί από την νέα τυχαία ακέραια τιμή που βρίσκεται στην μεταβλητή a.

```
A[pos] = a;
A1[pos] = 0;
for(i=0; i<y; i++){
    if (A[pos] == B[i]){
      A1[pos] += 1;
    }
}
```

Ήρθε η ώρα λοιπόν της αντικατάστασης. Με την πρώτη εντολή, **A[pos]=a**, δίνουμε την τιμή της μεταβλητής a στην θέση που βρέθηκε ο ακέραιος με λιγότερα join. Όμως, δεν γνωρίζουμε αν και πόσα join κάνει ο νέος ακέραιος. Έτσι, μηδενίζουμε την αντιστοιχη θέση στον πίνακα A1 (**A1[pos]=0**) και ελέγχουμε, με τη βοήθεια του βρόχου for, τον αριθμό των join που κάνει και τον αποθηκεύουμε στην αντίστοιχη αυτή θέση.

```
for(i=0; i<y; i++){
    B1[i] = 0;
}
for(i2=0; i2<y; i2++){
    for(i=0; i<x; i++){
        if (B[i2] == A[i]){
          B1[i2] += 1;
        }
    }
```

```
   }
}
```

Επειδή άλλαξε ένα στοιχείο στον πίνακα Α, τώρα δεν είναι έγκυρες οι μετρήσεις των join του πίνακα Β. Έτσι, μηδενίζουμε όλες τις μετρήσεις αυτές (κάνουμε όλα τα στοιχεία του πίνακα Β1 ίσα με 0) και κάνουμε ξανά την καταμέτρηση των join που κάνουν τα στοιχεία του πίνακα Β.

```
void insertB(int *B , int *B1 , int *A , int *A1 , int x, int y , int *count1){
   int i , i2 , ak , pos , b;
   b = rand()% NUMBERS+1;
   for(i=0; i<y; i++){
        if(B[i] == b){
             for(i=0; i<x; i++){
                  if (A[i] == b){
                    *count1 += 1;
                    break;
                  }
             }
             b = rand()% NUMBERS+1;
             i = -1;
        }
   }
   ak = B1[0];
   pos = 0;
   for(i=1; i<y; i++){
```

```
        if (B1[i]<ak){
          ak = B1[i];
          pos = i;
        }
    }
    B[pos] = b;
    B1[pos] = 0;
    for(i=0; i<x; i++){
        if (B[pos] == A[i]){
          B1[pos] += 1;
        }
    }
    for(i=0; i<x; i++){
        A1[i] = 0;
    }
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if (A[i] == B[i2]){
              A1[i] += 1;
            }
        }
    }
}
```

Εδώ γίνεται αντίστοιχη δουλειά για τον πίνακα Β.

```
void results(int *A1 , int x , int *count1){
   int i , count=0;
   for(i=0; i<x; i++){
        count += A1[i];
   }
   count += *count1;
   printf("Ta apotelesmata sou einai: %d join\n", count);
   scanf("%d\n", &i);
}
```

Τέλος για τον αλγόριθμο LFU, έχουμε τον ορισμό της συνάρτησης results, η οποία δεν επιστρέφει τίποτα και τα ορίσματά της φαίνονται στις παρενθέσεις.

Την μεταβλητή **i** θα την χρησιμοποιήσουμε για μετρητή, ενώ η μεταβλητή **count** θα καταγράψει τα συνολικά join που γίνονται ανάμεσα στους δύο πίνακες, αθροίζοντας όλα τα στοιχεία του πίνακα A1 (αν αθροίζαμε τα στοιχεία του πίνακα B1 θα βγάζαμε το ίδιο αποτέλεσμα). Στη συνέχεια προσθέτουμε στη μεταβλητή αυτή, την ακέραια τιμή που κρατά η μεταβλητή **count1**, η οποία έχει καταγράψει τα join που έκαναν οι ακέραιοι που διαβάστηκαν αλλά δεν καταχωρήθηκαν. Αμέσως μετά, τυπώνουμε τα αποτελέσματά μας. Υπενθυμίζουμε ότι η εντολή *scanf* την χρησιμοποιούμε για να μπορέσει το λογισμικό μας (DEV C++) να μας εμφανίσει τα αποτελέσματά μας.

4.4 Ο αλγόριθμος LRU (Less Recently Used)

4.4.1 Εισαγωγή

Ο Αλγόριθμος **LRU** είναι τελευταίος που θα μελετήσουμε σε αυτό το έγγραφο. Υπενθυμίζουμε ότι ο κάθε αλγόριθμος διαφέρει με τους άλλους στον τρόπο διαγραφής μιας καταχώρησης για να μπεί μία νέα, όμως θα μπορούσαμε να πούμε ότι ο αλγόριθμος LRU διαφέρει αρκετά από τους άλλους.

Κάθε φορά που εισέρχεται ένας νέος ακέραιος, ελέγχουμε αν υπάρχει στον πίνακα για τον οποίο διαβάστηκε. Αν υπάρχει τότε μεταφέρεται από την θέση που βρέθηκε, στην τελευταία θέση. Αν βρεθεί η ίδια τιμή στον άλλο πίνακα, μεταφέρεται και εκεί στην τελευταία θέση του. Αν όμως δεν βρεθεί η ίδια τιμή στον πίνακα για τον οποίο διαβάστηκε, τότε όλα τα στοιχεία του πίνακα κάνουν *shift* προς την πρώτη θέση (δηλαδή, η τελευταία γίνεται προτελευταία.. η $5^{η}$ γίνεται $4^{η}$.. η $2^{η}$ γίνεται $1^{η}$) και η καινούργια γίνεται τελευταία τιμή του πίνακα. Στο δια ταύτα θα χαθεί η τιμή του πρώτου στοιχείου του πίνακα.

4.4.2 Επεξήγηση αλγορίθμου

#include <stdio.h>

#include <stdlib.h>

#define NUMBERS 240

Ο αλγόριθμος εδώ μας δηλώνει ότι θα έχουμε στη διάθεσή μας τις βιβλιοθήκες **stdio.h** και **stdlib.h** καθώς και ότι η σταθερά **NUMBERS** θα αντιπροσωπεύει την τιμή 240.

void getStart(int * , int * , int , int);

void insertA(int * , int * , int , int , int *);

void insertB(int * , int * , int , int , int *);

void results(int * , int * , int , int , int *);

Στο αμέσως επόμενο κομμάτι του αλγόριθμου, βλέπουμε τις δηλώσεις των συναρτήσεων **getStart**, **insertA**, **insertB** και **results**, οι οποίες δεν επιστρέφουν τίποτα και έχουν τα ορίσματα όπως φαίνονται παραπάνω.

main(){

int i , x , y , PA , PB , count1=0;

int *A , *B;

Εδώ ξεκινά η συνάρτηση main(), η οποία, όπως και στους άλλους αλγόριθμους έτσι και εδώ, δημιουργεί τη μεταβλητή **i** που θα την χρησιμοποιήσουμε σαν μετρητή. Επίσης τις μεταβλητές **PA** , **PB** οι οποίες θα αντιπροσωπεύουν το πόσες φορές θα καλεστούν οι συναρτήσεις insertA και insertB αντίστοιχα και οι τιμές τους θα παρθούν από τον χρήστη. Τέλος, δημιουργεί τις μεταβλητές **x** , **y** οι οποίες και αυτές θα πάρουν τιμές από τον χρήστη και δουλειά τους είναι να συγκρατούν το μέγεθος των πινάκων A και B αντίστοιχα. Η μεταβλητή **count1** παίρνει αρχική τιμή 0 και θα μετρά τις φορές που έχει διαβαστεί ένας ακέραιος που υπήρχε στον πίνακα για τον οποίο διαβάστηκε, αλλά υπήρχε και στον άλλο πίνακα. Η

καταμέτρηση που θα κάνει η μεταβλητή count1 θα προστεθεί στα συνολικά αποτελέσματα. Αμέσως μετά δημιουργούμε τους δείκτες **A** και **B**.

```
printf("Dose posa insertA theleis na exeis\n");
scanf("\n%d", &PA);
printf("Dose posa insertB theleis na exeis\n");
scanf("\n%d", &PB);
printf("Dose poses grames theleis na exei o pinakas A\n");
scanf("\n%d", &x);
printf("Dose poses grames theleis na exei o pinakas B\n");
scanf("\n%d", &y);
```

Εδώ διαβάζουμε τιμές από τον χρήστη και τις δίνουμε στις μεταβλητές PA, PB, x και y αντίστοιχα, με την βοήθεια των εντολών *printf* και *scanf*.

```
A = (int *)malloc(x * sizeof(int));
B = (int *)malloc(y * sizeof(int));
getStart(A , B , x , y);
```

Σε αυτό το σημείου του αλγόριθμο, δεσμεύουμε χώρο στη μνήμη για τους δείκτες A και B, όσο x ακέραιοι και y ακέραιοι αντίστοιχα και στη συνέχεια καλούμε τη συνάρτηση getStart με τα ορίσματα που φαίνονται παραπάνω.

```
for(i=x; i<PA; i++){
    insertA(A , B , x , y , &count1);
}
for(i=y; i<PB; i++){
    insertB(B , A , x , y , &count1);
```

```
    }
    results(A , B , x , y , &count1);
}
```

Αμέσως μετά, καλούμε PA-x φορές την συνάρτηση insertA και PB-y φορές την συνάρτηση insertB και στη συνέχεια καλούμε την συνάρτηση results, με τα ορίσματα που φαίνονται παραπάνω στην κάθε συνάρτηση.

```
void getStart(int *A , int *B , int x , int y){
    int i , i2 , a , b , c=0;
    A[0] = rand()% NUMBERS+1;
    B[0] = rand()% NUMBERS+1;
```

Μόλις έχει τελειώσει η συνάρτηση main(), και σειρά έχει ο ορισμός της συνάρτησης getStart, η οποία δεν επιστρέφει τίποτα και δέχεται τα ορίσματα όπως αναγράφονται στον αλγόριθμο.

Έπειτα, δημιουργούμε τις μεταβλητές **i** και **i2** οι οποίες θα παίξουν τον ρόλο του μετρητή, καθώς και οι μεταβλητές **a** και **b** οι οποίες θα διαβάζουν τους εκάστοτε πιθανούς τυχαίους ακέραιους αριθμούς για καταχώρηση στους πίνακες Α και Β αντίστοιχα. Καθώς δημιουργούμε την μεταβλητή **c** και της δίνουμε αρχική τιμή 0, σας υπενθυμίζουμε ότι παίρνει τιμές 0 ή 1 και θα μας βοηθάει στο να καταλαβαίνουμε αν ο compiler μπήκε σε διάφορα block εντολών. Στη συνέχεια, οι πρώτες θέσεις των πινάκων Α και Β (**A[0]**,**B[0]**) παίρνουν τυχαίες ακέραιες τιμές στην γκάμα που βρισκόμαστε, δηλαδή 0-240.

```
    for(i=1; i<x; i++){
        a = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(A[i2] == a){
                for( ; i2<i; i2++){
                    A[i2] = A[i2+1];
```

```
                }
                A[i-1] = a;
                i--;
                c = 1;
                break;
            }
        }
        if(c == 0){
            A[i] = a;
        }
        c = 0;
    }
```

Στη συνέχεια μπαίνουμε στη διαδικασία εισαγωγής του νέου ακεραίου. Θέλουμε να γεμίσουμε τις υπόλοιπες θέσεις του πίνακα Α, γι'αυτό και δίνουμε αρχική τιμή στο μετρητή 1 και όχι 0. Έπειτα, δίνουμε στην μεταβλητή a μία νέα τυχαία ακέραια τιμή στην γκάμα που βρισκόμαστε, και με τη βοήθεια του βρόχου for και του μετρητή i2, ψάχνουμε όλες τις θέσεις του πίνακα Α, που έχουμε βάλει τιμή (0 έως i), για να δούμε αν υπάρχει η τιμή της a στον υπάρχον πίνακα Α. Αν βρούμε σε κάποια θέση του πίνακα Α, την ίδια τιμή με την μεταβλητή a (**if(A[i2] == a)**) τότε αντιμεταθέτει όλα τα υπόλοιπα στοιχεία που έχει καταχωρήσει κατά μία θέση μπροστά (**A[i2] = A[i2+1]**). Στη συνέχεια, στη τελευταία θέση του πίνακα Α (τελευταία από τις ήδη καταχωρημένες θέσεις, και όχι του συνολικού πίνακα, εξού A[i-1] και όχι A[x-1]) δίνουμε την τιμή της μεταβλητής a,και ο μετρητής i παίρνει μείωση κατά μία μονάδα. Αυτό γίνεται ουσιαστικά διότι χάθηκε η τιμή της πρώτης θέσης, και εισήλθε η νέα τιμή στο τέλος, με αποτέλεσμα ο αριθμός των καταχωρημένων θέσεων να μείνει ίδιος. Άρα, αν για παράδειγμα πριν είχε 5 καταχωρημένα στοιχεία ο πίνακας Α, πάλι 5 έχει και τώρα. Δηλαδή, η τιμή του i δεν πρέπει να αλλάξει στην επόμενη επανάληψη. Αμέσως μετά, δίνουμε την τιμή 1 στην μεταβλητή c και με τη βοήθεια της μεταβλητής **break** σταματάμε το ψάξιμο του πίνακα Α. Έπειτα, αν η μεταβλητή c έχει τιμή 0, ή αλλιώς αν η τιμή της μεταβλητής a δεν υπήρχε ήδη στον υπάρχον πίνακα Α, τότε απλά δίνουμε την τιμή της στην i θέση του πίνακα Α. Σε αυτήν την περίπτωση ο μετρητής μας δεν παίρνει μείωση διότι ο αριθμός των καταχωρημένων

θέσεων του πίνακα Α αυξήθηκε κατά μία μονάδα. Στη συνέχεια, μηδενίζουμε την μεταβλητή c για την επόμενη επανάληψη (**c=0**).

```
    c = 0;
    for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(B[i2] == b){
                for( ; i2<i; i2++){
                    B[i2] = B[i2+1];
                }
                B[i-1] = b;
                i--;
                c = 1;
                break;
            }
        }
        if(c == 0){
            B[i] = b;
        }
        c = 0;
    }
}
```

Μηδενίζοντας την μεταβλητή c, συνεχίζουμε και κάνουμε την ίδια δουλειά για τον πίνακα Β.

```
void insertA(int *A , int *B , int x , int y , int *count1){
   int i , c=0 , a;
   a = rand()% NUMBERS+1;
```

Εδώ ξεκινά η συνάρτηση insertA, η οποία δεν επιστρέφει τίποτα και έχει ως ορίσματα τρείς δείκτες, τους *A,B,count1* και δύο ακέραιους, τους *x,y*. Αμέσως μετά, δημιουργούμε την μεταβλητή **i**, η οποία και πάλι παίζει τον ρόλο του μετρητή. Η μεταβλητή **c** παίρνει αρχική τιμή 0 και θα μας βοηθά να καταλάβουμε αν ο compiler μπήκε σε διάφορα block εντολών, και η μεταβλητή **a** που θα παίρνει τις εκάστοτε τυχαίες ακέραιες τιμές στην αντίστοιχη γκάμα που θα βρισκόμαστε. Στη συνέχεια η μεταβλητή a παίρνει αρχική τυχαία ακέραια τιμή στην γκάμα 0-240.

```
   for(i=0; i<x; i++){
      if(A[i] == a){
         for ( ; i<x; i++){
            A[i] = A[i+1];
         }
         for(i=0; i<y; i++){
            if (B[i] == a){
               *count1 += 1;
               break;
            }
         }
         A[x-1] = a;
         c = 1;
         break;
      }
```

}

Έπειτα, με τη βοήθεια του βρόχου for ψάχνουμε όλα τα στοιχεία του πίνακα Α να δούμε αν υπάρχει η τιμή της μεταβλητής a στον υπάρχον πίνακα. Αν την βρούμε σε κάποιο σημείο του πίνακα Α (**if(A[i] == a)**) τότε όλα τα στοιχεία από εκείνη τη θέση και έπειτα τα αντιμεταθέτουμε με μία θέση μπροστά. Αν για παράδειγμα ο πίνακας έχει 10 θέσεις, και την βρούμε στην $7^η$ θέση, τότε με τη βοήθεια της δεύτερης for η $8^η$ θέση θα δώσει την τιμή της στην $7^η$, η $9^η$ στην $8^η$ και η $10^η$ στην $9^η$. Αμέσως μετά, ψάχνουμε τον πίνακα Β για να δούμε αν η τιμή αυτή υπάρχει σε αυτόν, διότι αν υπάρχει τότε η μεταβλητή **count1** θα αυξηθεί και θα μετρηθεί στα αποτελέσματα (πιο σωστα θα αυξηθεί κατά μία μονάδα τα περιεχόμενα του δείκτη count1). Στη συνέχεια, για το παράδειγμα που λέγαμε παραπάνω, η τιμή της μεταβλητής a θα δωθεί στην $10^η$ θέση του πίνακα Α και αφού η μεταβλητή **c** πάρει την τιμή 0, σταματάμε το ψάξιμο του πίνακα Α με τη βοήθεια της μεταβλητής **break**.

Στο δια ταύτα, αν βρούμε την τιμή της μεταβλητής a στον πίνακα Α, τότε η τιμή μεταφέρεται στην τελευταία θέση του πίνακα και όλα τα στοιχεία από εκεί που βρέθηκε μέχρι το τέλος μετακινήθηκαν μία θέση μπροστά. Επίσης, ελέγξαμε αν η τιμή αυτή υπάρχει στον πίνακα Β.

```
if(c == 0){
    for(i=0; i<x; i++){
        A[i] = A[i+1];
    }
    A[x-1] = a;
}
```

Αν η τιμή c έχει την τιμή 0, ή αλλιώς αν η τιμή της μεταβλητής a δεν βρέθηκε στον πίνακα Α, τότε έχουμε μετακίνηση όλων των στοιχείων του πίνακα Α και η τιμή της μεταβλητής a δίνεται στην τελευταία θέση του πίνακα. Με άλλα λόγια, αν δεν βρεθεί η τιμή της a στον πίνακα, τότε διαγράφεται το πρώτο στοιχείο του, όλα τα υπόλοιπα στοιχεία πάνε μία θέση μπροστά και η νέα τιμή πάει στο τέλος.

```
for(i=0; i<y-1; i++){
```

```
        if(B[i] == a){
            for( ; i<y; i++){
                B[i] = B[i+1];
            }
            B[y-1] = a;
            break;
        }
    }
}
```

Έπειτα, ελέγχουμε αν υπάρχει η τιμή της a στον πίνακα B, και αν υπάρχει τότε η τιμή αυτή πάει στην τελευταία θέση του πίνακα και τα στοιχεία από εκεί που βρέθηκε μέχρι το τέλος πάνε μία θέση μπροστά. Αν βρεθεί η τιμή της a στον πίνακα B, τότε θα γίνει η παραπάνω δουλειά μία φορά και αυτό το μας το διασφαλίζει η εντολή break.

```
void insertB(int *B , int *A , int x , int y , int *count1){
    int i , c=0 , b;
    b = rand()% NUMBERS+1;
    for(i=0; i<y; i++){
        if(B[i] == b){
            for ( ; i<y; i++){
                B[i] = B[i+1];
            }
            for(i=0; i<x; i++){
                if (A[i] == b){
                    *count1 += 1;
```

```
                    break;
                }
            }
            B[y-1] = b;
            c = 1;
            break;
        }
    }
    if(c == 0){
        for(i=0; i<y; i++){
            B[i] = B[i+1];
        }
        B[y-1] = b;
    }
    for(i=0; i<x; i++){
        if(A[i] == b){
            for( ; i<x; i++){
                A[i] = A[i+1];
            }
            A[x-1] = b;
            break;
        }
    }
}
```

Στον παραπάνω αλγόριθμο γίνεται η ίδια δουλειά για τον πίνακα Β.

```
void results(int *A ,  int *B , int x , int y , int *count1){
   int i , i2 , count;
   count = 0;
```

Στη συνέχεια, ξεκινά ο ορισμός της συνάρτησης results, η οποία δεν επιστρέφει τίποτα και δέχεται τα ορίσματα όπως φαίνονται στις παρενθέσεις. Οι μεταβλητές **i**,**i2** θα παίξουν τον ρόλο του μετρητή ενώ η μεταβλητή **count** θα μετρά τα join που κάνουν οι δύο πίνακες.

```
   for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
             if(A[i] == B[i2]){
                  count++;
             }
        }
   }
```

Σε αυτό το σημείο του αλγόριθμου ελέγχουμε όλα τα στοιχεία του πίνακα Α με όλα τα στοιχεία του πίνακα Β για δούμε τα join τα οποία επιτυγχάνουν. Ο αριθμός των join εκφράζεται από την μεταβλητή **count** η οποία αυξάνεται κατά μία μονάδα μόλις βρεθεί ομοιότητα στις τιμές των δύο πινάκων.

```
   count += *count1;
   printf("Ta apotelesmata sou einai: %d join\n", count);
   scanf("%d\n", &i);
}
```

Στη συνέχεια, προσθέτουμε στην μεταβλητή count τα περιεχόμενα του δείκτη **count1**. Υπενθυμίζουμε ότι η μεταβλητή count1 συγκρατεί τον αριθμό των join που έγιναν των τιμών των οποίων διαβάστηκαν, έκαναν join αλλά υπήρχαν ήδη στον πίνακα για τον οποίο διαβάστηκαν. Τέλος, τυπώνουμε τα αποτελέσματά μας με τη βοήθεια της εντολής printf και η εντολή scanf μας επιτρέπει να βλέπουμε τα αποτελέσματά μας στο λογισμικό που έχουμε (DEV C++).

Κεφάλαιο 5

Πειραματικά αποτελέσματα

5.1 Εισαγωγή

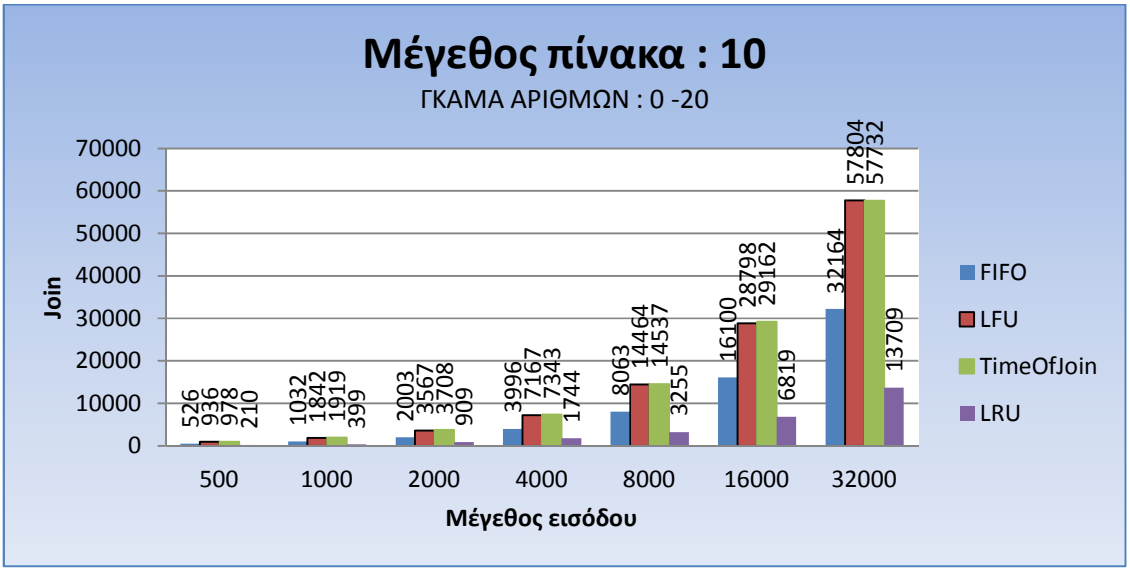
Για κάθε αλγόριθμο έχουμε πάρει μετρήσεις λαμβάνοντας υπόψιν τέσσερα διαφορετικά μεγέθη των πινάκων που αποθηκεύουν τις τυχαίες ροές (πίνακες Α και Β). Τα μεγέθη είναι τα εξής: 10, 20, 30, 40. Επίσης, για κάθε μέγεθος πίνακα έχουμε πάρει αποτελέσματα σε τρείς διαφορετικές γκάμες αριθμών κάθε φορά, x2, x4 και x6 το μέγεθος του πίνακα. Για παράδειγμα, οι μετρήσεις με μέγεθος πίνακα 10 έχουν πειραματικά αποτελέσματα στις γκάμες 0-20, 0-40 και 0-60. Παράλληλα, για κάθε γκάμα έχουμε επτά διαφορετικά μεγέθη εισόδου (PA και PB, ή αλλιώς διαφορετικές φορές θα καλεστεί η συνάρτηση insertA και insertB), τις τιμές 500, 1000, 2000, 4000, 8000, 16000 και 32000. Παρακάτω θα δούμε τους πίνακες των πειραματικών αποτελεσμάτων, καθώς και τις γραφικές παραστάσεις.

5.2 Μετρήσεις – Γραφικές παραστάσεις

Για μέγεθος πίνακα 10 έχουμε:

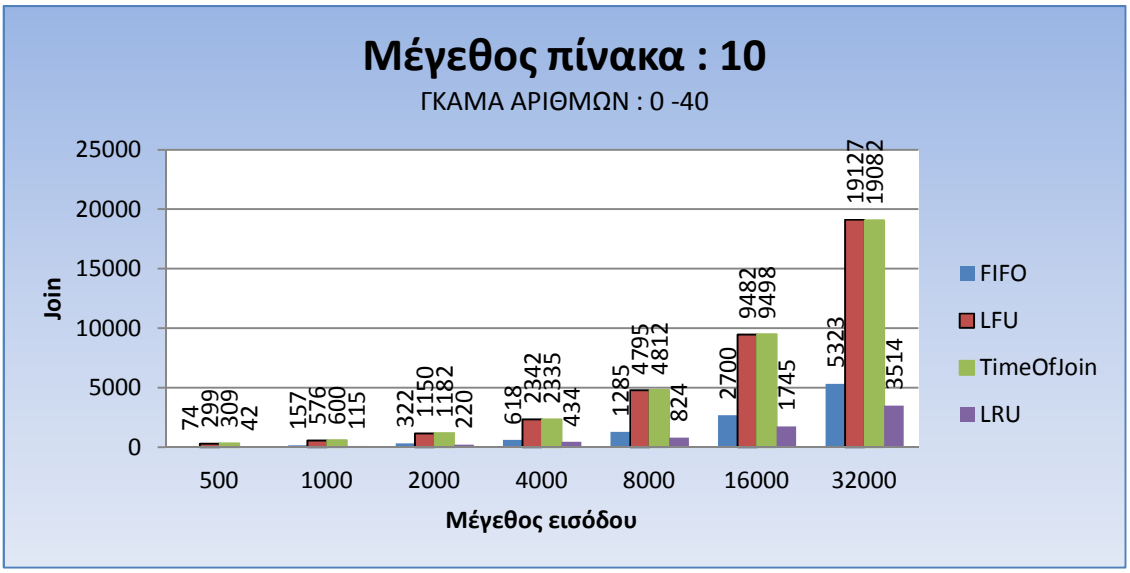
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -20

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	526	936	978	210
1000	1032	1842	1919	399
2000	2003	3567	3708	909
4000	3996	7167	7343	1744
8000	8063	14464	14537	3255
16000	16100	28798	29162	6819
32000	32164	57804	57732	13709



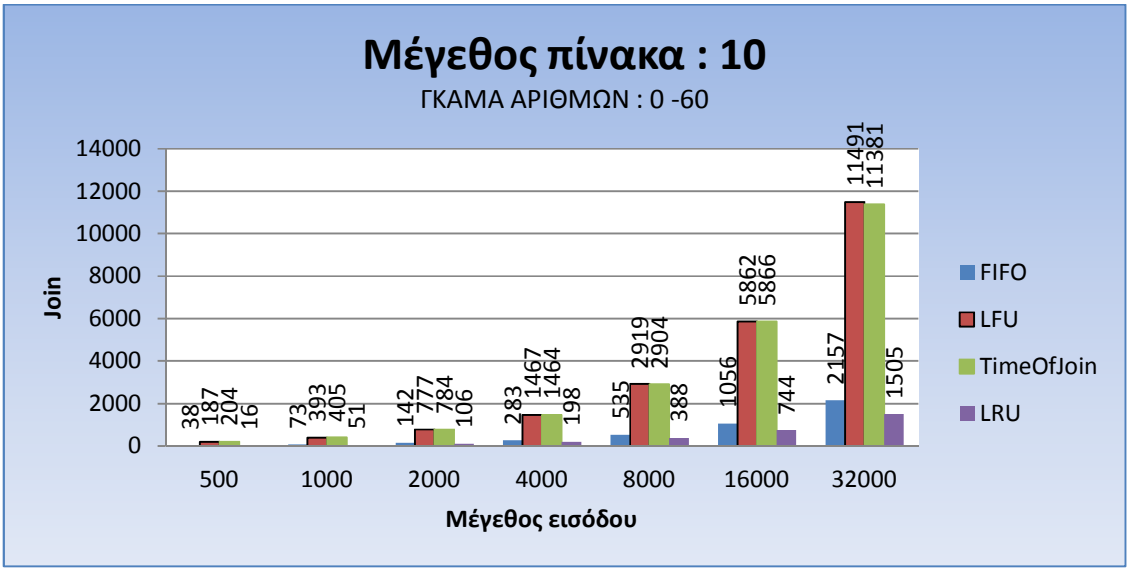
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -40

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	74	299	309	42
1000	157	576	600	115
2000	322	1150	1182	220
4000	618	2342	2335	434
8000	1285	4795	4812	824
16000	2700	9482	9498	1745
32000	5323	19127	19082	3514



ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -60

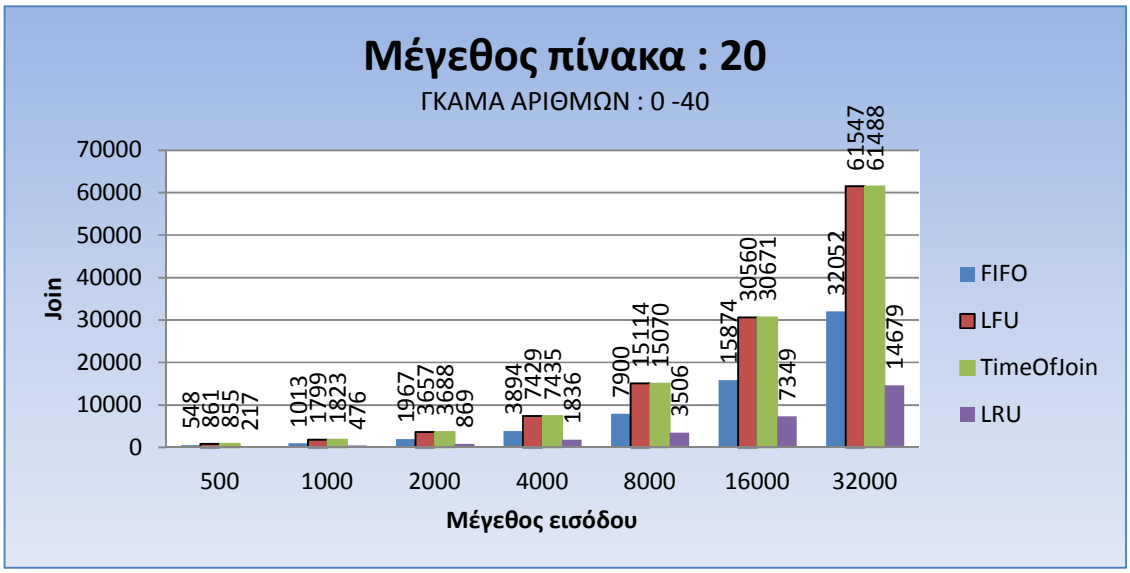
Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	38	187	204	16
1000	73	393	405	51
2000	142	777	784	106
4000	283	1467	1464	198
8000	535	2919	2904	388
16000	1056	5862	5866	744
32000	2157	11491	11381	1505



Για μέγεθος πίνακα 20 έχουμε:

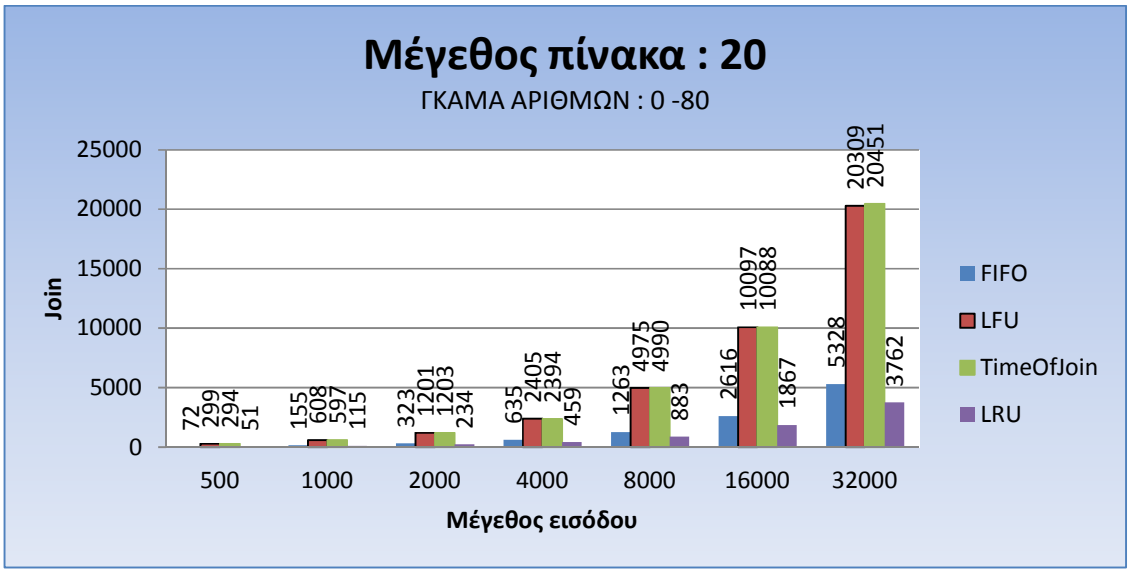
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -40

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	548	861	855	217
1000	1013	1799	1823	476
2000	1967	3657	3688	869
4000	3894	7429	7435	1836
8000	7900	15114	15070	3506
16000	15874	30560	30671	7349
32000	32052	61547	61488	14679



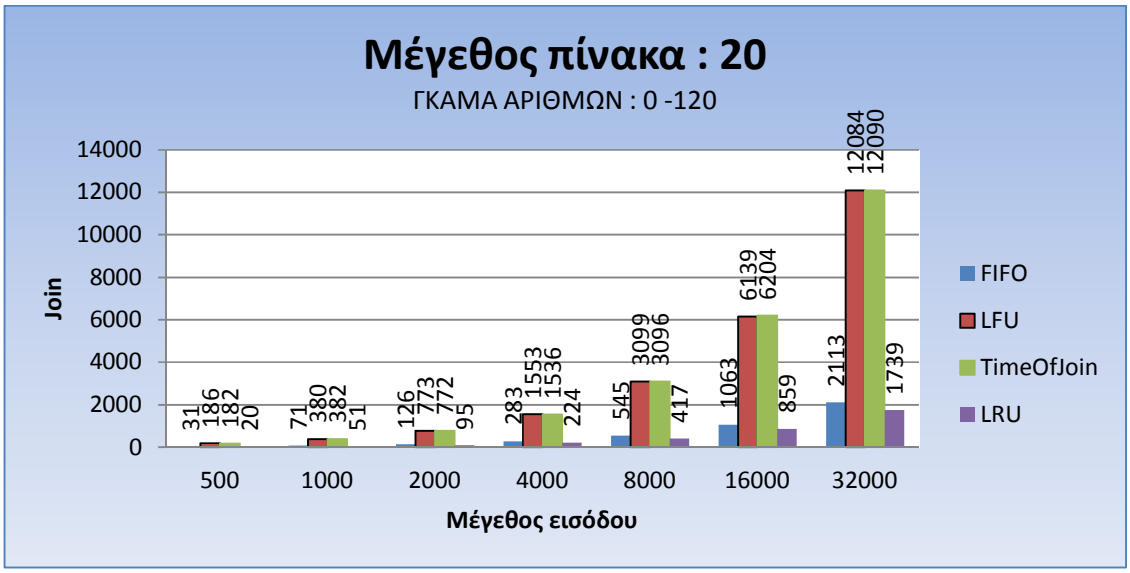
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -80

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	72	299	294	51
1000	155	608	597	115
2000	323	1201	1203	234
4000	635	2405	2394	459
8000	1263	4975	4990	883
16000	2616	10097	10088	1867
32000	5328	20309	20451	3762



ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 - 120

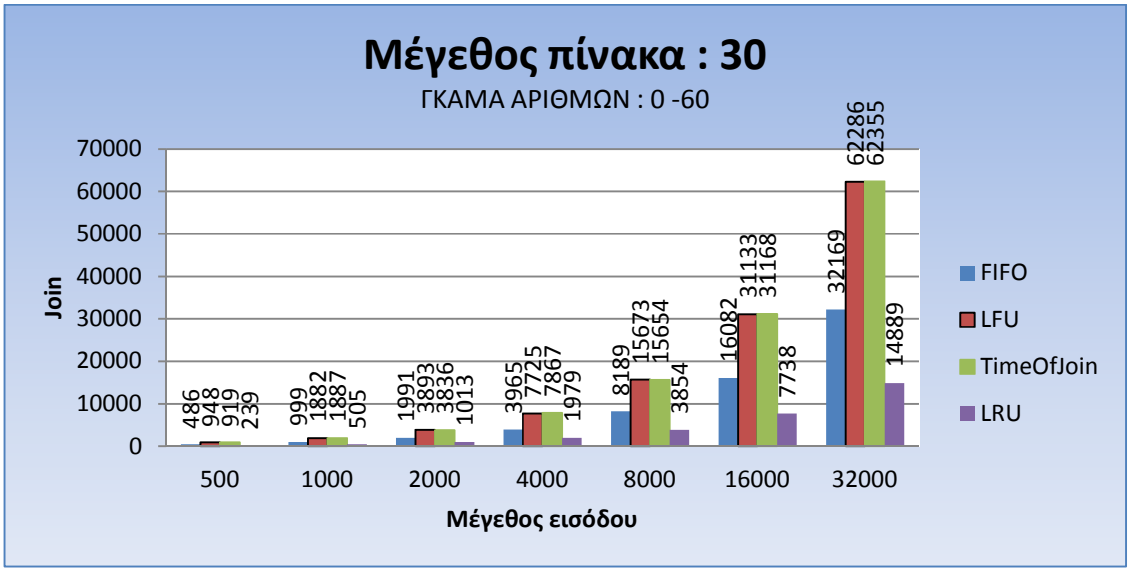
Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	31	186	182	20
1000	71	380	382	51
2000	126	773	772	95
4000	283	1553	1536	224
8000	545	3099	3096	417
16000	1063	6139	6204	859
32000	2113	12084	12090	1739



Για μέγεθος πίνακα 30 έχουμε:

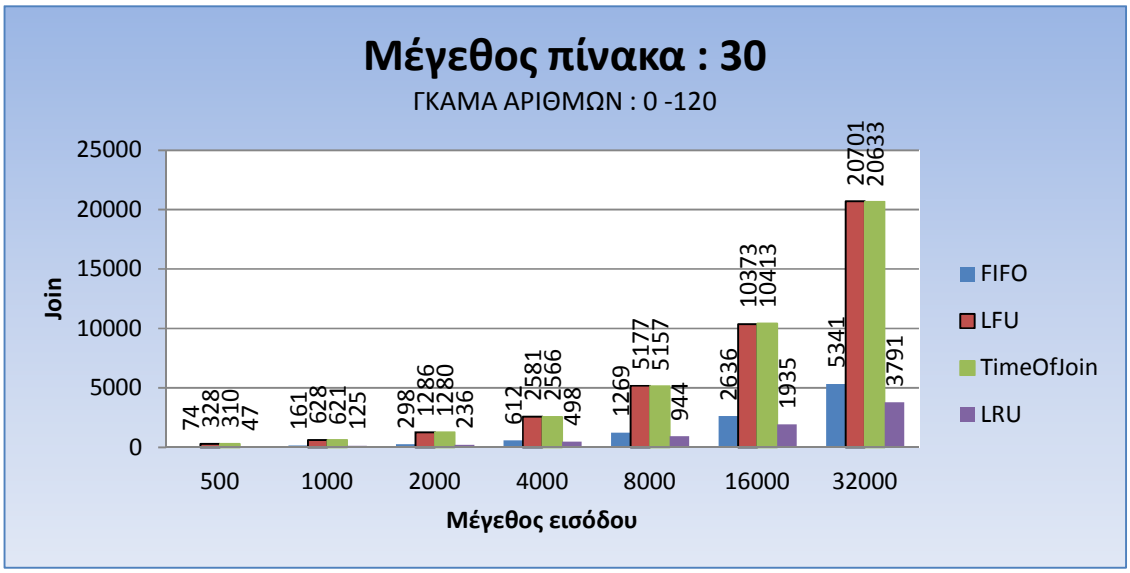
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -60

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	486	948	919	239
1000	999	1882	1887	505
2000	1991	3893	3836	1013
4000	3965	7725	7867	1979
8000	8189	15673	15654	3854
16000	16082	31133	31168	7738
32000	32169	62286	62355	14889



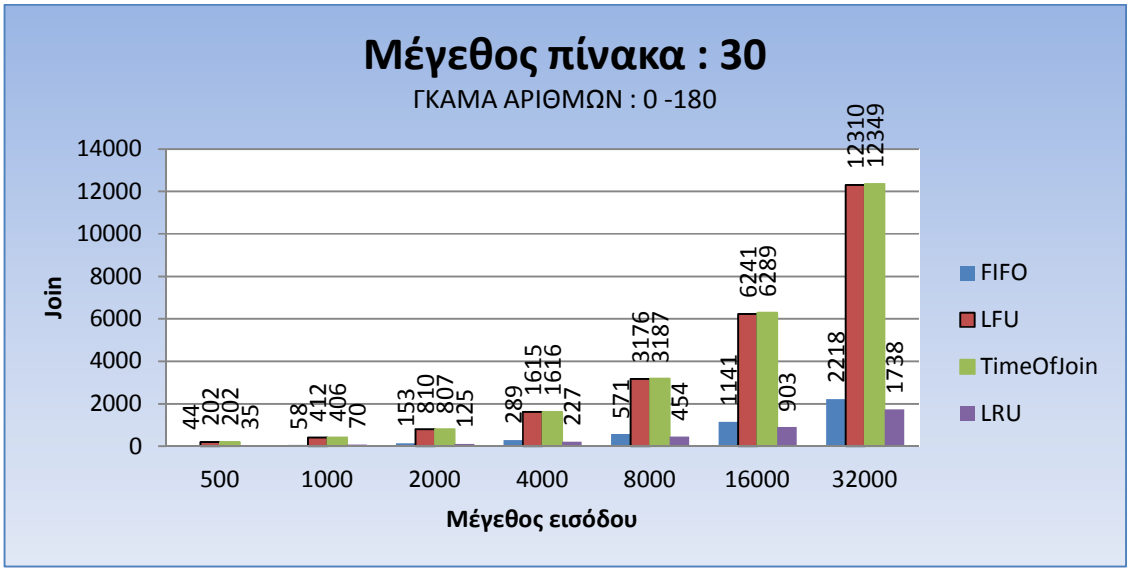
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 - 120

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	74	328	310	47
1000	161	628	621	125
2000	298	1286	1280	236
4000	612	2581	2566	498
8000	1269	5177	5157	944
16000	2636	10373	10413	1935
32000	5341	20701	20633	3791



ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 - 180

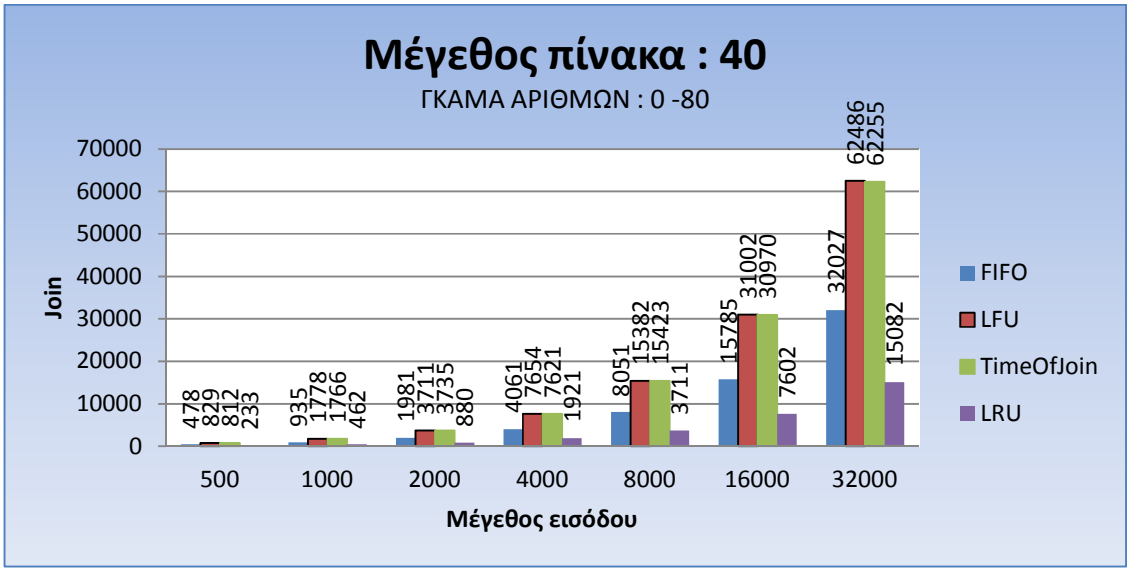
Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	44	202	202	35
1000	58	412	406	70
2000	153	810	807	125
4000	289	1615	1616	227
8000	571	3176	3187	454
16000	1141	6241	6289	903
32000	2218	12310	12349	1738



Για μέγεθος πίνακα 40 έχουμε:

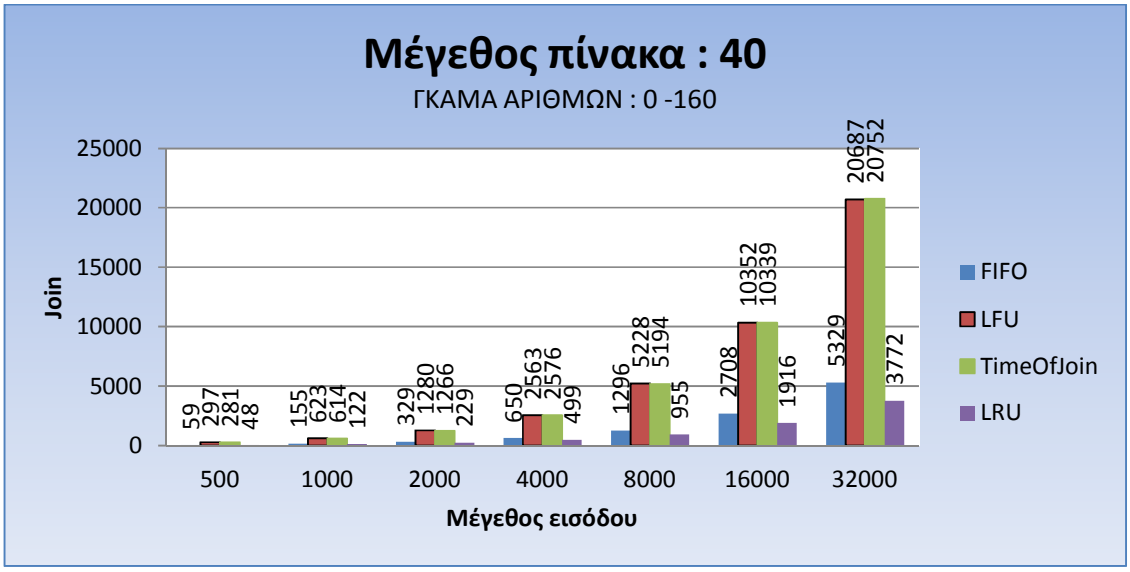
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 -80

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	478	829	812	233
1000	935	1778	1766	462
2000	1981	3711	3735	880
4000	4061	7654	7621	1921
8000	8051	15382	15423	3711
16000	15785	31002	30970	7602
32000	32027	62486	62255	15082



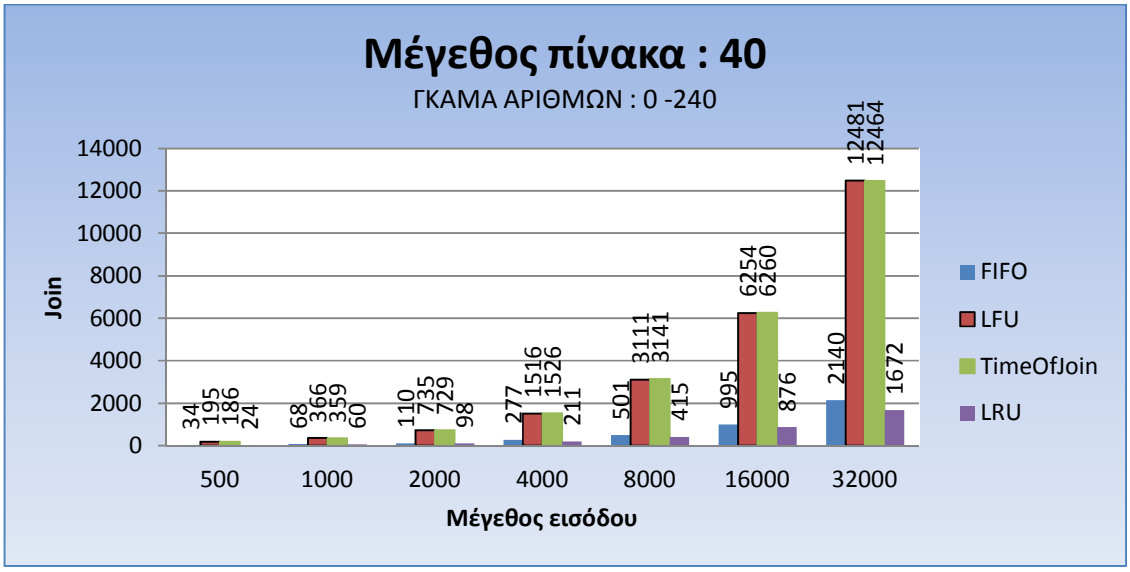
ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 - 160

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	59	297	281	48
1000	155	623	614	122
2000	329	1280	1266	229
4000	650	2563	2576	499
8000	1296	5228	5194	955
16000	2708	10352	10339	1916
32000	5329	20687	20752	3772



ΓΚΑΜΑ ΑΡΙΘΜΩΝ : 0 - 240

Μέγεθος Εισόδου	FIFO	LFU	TimeOfJoin	LRU
500	34	195	186	24
1000	68	366	359	60
2000	110	735	729	98
4000	277	1516	1526	211
8000	501	3111	3141	415
16000	995	6254	6260	876
32000	2140	12481	12464	1672



Για τις παραπάνω μετρήσεις, όπως προαναφέραμε, παίζουν ρόλο τρείς παράμετροι. Η γκάμα αριθμών που βρισκόμαστε στο εκάστοτε παράδειγμα, το μέγεθος των πινάκων Α και Β, και το μέγεθος της τυχαίας ροής εισόδου. Για κάθε μέγεθος πίνακα έχουμε τρείς διαφορετικές γκάμες, άρα και τρία διαφορετικά σχεδιαγράμματα. Σε κάθε σχεδιάγραμμα έχουμε πάρει τιμές για επτά διαφορετικά μεγέθοι εισόδου, τα οποία φαίνονται στον οριζόντιο άξονα του κάθε γραφήματος. Στον κάθετο άξονα αντικρίζουμε τα join που πετυχαίνει ο κάθε αλγόριθμος στην εκάστοτε μέτρηση. Όπως μπορείτε να δείτε, σε κάθε μέγεθος εισόδου έχουμε μία μέτρηση για τον κάθε αλγόριθμο. Για παράδειγμα, στο μέγεθος πίνακα 40 και στην γκάμα 0-240, για μέγεθος εισόδου 2000 βλέπουμε τον αλγόριθμο FIFO να πετυχαίνει 110 join, ο αλγόριθμος LRU 98 και οι αλγόριθμοι TimeOfJoin και LFU να έχουν 729 και 735 join αντίστοιχα. Αν παρατηρήσετε τις μετρήσεις μας θα δείτε ότι οι αλγόριθμοι FIFO και LRU καταφέρνουν τα λιγότερα join σε σχέση με τους υπόλοιπους δύο, ενώ είναι δύσκολο να ξεχωρίσουμε κάποιον από τους TimeOfJoin και LFU καθώς τα αποτελέσματα του ενός συμβαδίζουν με του άλλου. Κάποιες φορές ο αλγόριθμος TimeOfJoin καταφέρνει να έχει περισσότερες διασυνδέσεις, και κάποιες άλλες ο αλγόριθμος LFU παίρνει την πρωτιά. Στο δια ταύτα, θα μπορούσαμε να πούμε ότι έχουμε πιο επιθυμητά αποτελέσματα όταν έχουμε σημασιολογική περικοπή φορτίου από όταν έχουμε τυχαίες ροές που αντικαθαστόνται χωρίς επιλογή.

Παράρτημα Α

Στο παράρτημα αυτό θα παραθέσουμε τους τέσσερις αλγόριθμους που αναλύσαμε στο κεφάλαιο 4.

Α.1 Ο αλγόριθμος FIFO

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBERS 240
void getStart(int * , int * , int * , int * , int , int , int * , int *);
int insertA(int * , int * , int * , int , int , int , int *);
int insertB(int * , int * , int * , int , int , int , int *);
void results(int * , int * , int , int , int *);

main(){
    int i , x , y , PA , PB , acount , bcount , count1=0;
    int *A , *A1 , *B , *B1;
    printf("Dose posa insertA theleis na exeis\n");
    scanf("\n%d", &PA);
    printf("Dose posa insertB theleis na exeis\n");
    scanf("\n%d", &PB);
    printf("Dose poses grames theleis na exei o pinakas A\n");
```

```
    scanf("\n%d", &x);
    printf("Dose poses grames theleis na exei o pinakas B\n");
    scanf("\n%d", &y);
    A = (int *)malloc(x * sizeof(int));
    A1 = (int *)malloc(x * sizeof(int));
    B = (int *)malloc(y * sizeof(int));
    B1 = (int *)malloc(y * sizeof(int));
    acount = 1;
    bcount = 1;
    getStart(A , A1 , B , B1 , x , y , &acount , &bcount);
    for(i=x; i<PA; i++){
        acount = insertA(A , A1 , B , x , y , acount , &count1);
    }
    for(i=y; i<PB; i++){
        bcount = insertB(B , B1 , A , x , y , bcount , &count1);
    }
    results(A , B , x , y , &count1);
}

void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y , int *acount , int *bcount){
    int i , a , b , c=0 , i2;
    A[0] = rand()% NUMBERS+1;
    B[0] = rand()% NUMBERS+1;
    A1[0] = 0;
```

```
B1[0] = 0;
for(i=1; i<x; i++){
	a = rand()% NUMBERS+1;
	for(i2=0; i2<i; i2++){
		if(A[i2] == a){
			i--;
			c = 1;
			break;
		}
	}
	if(c == 0){
	    A[i] = a;
	    A1[i] = *acount;
	    *acount += 1;
	}
	c = 0;
}
c = 0;
for(i=1; i<y; i++){
	b = rand()% NUMBERS+1;
	for(i2=0; i2<i; i2++){
		if(B[i2] == b){
			i--;
			c = 1;
```

```
                    break;
                }
            }
            if(c == 0){
                B[i] = b;
                B1[i] = *bcount;
                *bcount += 1;
            }
            c = 0;
        }
}

int insertA(int *A , int *A1 , int *B , int x , int y , int acount , int *count1){
    int i , ak , pos , a;
    a = rand()% NUMBERS+1;
    for(i=0; i<x; i++){
        if(A[i] == a){
            for(i=0; i<y; i++){
                if (B[i] == a){
                  *count1 += 1;
                  break;
                }
            }
            a = rand()% NUMBERS+1;
```

```
                i = -1;
            }
        }
        ak = A1[0];
        pos = 0;
        for(i=1; i<x; i++){
            if (A1[i]<ak){
              ak = A1[i];
              pos = i;
            }
        }
        A[pos] = a;
        A1[pos] = acount;
        acount++;
        return acount;
}

int insertB(int *B , int *B1 , int *A , int x , int y , int bcount , int *count1){
        int i , ak , pos , b;
        b = rand()% NUMBERS+1;
        for(i=0; i<y; i++){
            if(B[i] == b){
                for(i=0; i<x; i++){
                    if (A[i] == b){
```

```
                    *count1 += 1;
                    break;
                  }
              }
              b = rand()% NUMBERS+1;
              i = -1;
          }
    }
    ak = B1[0];
    pos = 0;
    for(i=1; i<y; i++){
          if (B1[i]<ak){
            ak = B1[i];
            pos = i;
          }
    }
    B[pos] = b;
    B1[pos] = bcount;
    bcount++;
    return bcount;
}

void results(int *A ,  int *B , int x , int y , int *count1){
     int i , i2 , count;
```

```
    count = 0;
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if(A[i] == B[i2]){
                count++;
            }
        }
    }
    count += *count1;
    printf("Ta apotelesmata sou einai: %d join\n", count);
    scanf("%d\n", &i);
}
```

A.2 Ο αλγόριθμος TimeOfJoin

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBERS 240
void getStart(int * , int * , int * , int * , int , int , int * , int *);
int insertA(int * , int * , int * , int , int , int , int *);
int insertB(int * , int * , int * , int , int , int , int *);
void results(int * , int * , int , int , int *);

main(){
    int i , x , y , ta , tb , PA , PB , count1=0;
    int *A , *A1 , *B , *B1;
    ta = 1;
    tb = 1;
    printf("Dose posa insertA theleis na exeis\n");
    scanf("\n%d", &PA);
    printf("Dose posa insertB theleis na exeis\n");
    scanf("\n%d", &PB);
    printf("Dose poses grames theleis na exei o pinakas A\n");
    scanf("\n%d", &x);
    printf("Dose poses grames theleis na exei o pinakas B\n");
    scanf("\n%d", &y);
    A = (int *)malloc(x * sizeof(int));
```

```
    A1 = (int *)malloc(x * sizeof(int));
    B = (int *)malloc(y * sizeof(int));
    B1 = (int *)malloc(y * sizeof(int));
    getStart(A , A1 , B , B1 , x , y , &ta , &tb);
    for(i=x; i<PA; i++){
        ta = insertA(A , A1 , B , x , y , ta , &count1);
    }
    for(i=y; i<PB; i++){
        tb = insertB(B , B1 , A , x , y , tb , &count1);
    }
    results(A , B , x , y , &count1);
}

void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y , int *ta , int *tb){
    int i , i2 , a , b , c=0;
    A[0] = rand()% NUMBERS+1;
    B[0] = rand()% NUMBERS+1;
    for(i=1; i<x; i++){
        a = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(A[i2] == a){
                i--;
                c=1;
                break;
```

```
            }
        }
        if(c == 0){
            A[i] = a;
        }
        c = 0;
}
c = 0;
for(i=0; i<x; i++){
        A1[i] = -1;
}
for(i=0; i<y; i++){
        B1[i] = -1;
}
for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(B[i2] == b){
                i--;
                c=1;
                break;
            }
        }
        if(c == 0){
```

```
            B[i] = b;
        }
        c = 0;
    }
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if(A[i] == B[i2]){
                A1[i] = *ta;
                *ta += 1;
                break;
            }
        }
    }
    for(i2=0; i2<y; i2++){
        for(i=0; i<x; i++){
            if(B[i2] == A[i]){
                B1[i2] = *tb;
                *tb += 1;
                break;
            }
        }
    }
}
```

```
int insertA(int *A , int *A1 , int *B , int x , int y , int ta , int *count1){
   int i , i2 , ak , pos , a;
   a = rand()% NUMBERS+1;
   for(i=0; i<x; i++){
         if(A[i] == a){
               for(i=0; i<y; i++){
                     if (B[i] == a){
                       *count1 += 1;
                       break;
                     }
               }
               a = rand()% NUMBERS+1;
               i = -1;
         }
   }
   ak = A1[0];
   pos = 0;
   for(i=1; i<x; i++){
         if (A1[i]<ak){
           ak = A1[i];
           pos = i;
         }
   }
   A[pos] = a;
```

```
    A1[pos] = -1;
    for(i=0; i<y; i++){
        if(A[pos] == B[i]){
            A1[pos] = ta;
            ta += 1;
            break;
        }
    }
    return ta;
}

int insertB(int *B , int *B1 , int *A , int x , int y , int tb , int *count1){
    int i , i2 , ak , pos , b;
    b = rand()% NUMBERS+1;
    for(i=0; i<y; i++){
        if(B[i] == b){
            for(i=0; i<x; i++){
                if (A[i] == b){
                  *count1 += 1;
                  break;
                }
            }
            b = rand()% NUMBERS+1;
            i = -1;
```

```
            }
        }
        ak = B1[0];
        pos = 0;
        for(i=1; i<y; i++){
                if (B1[i]<ak){
                    ak = B1[i];
                    pos = i;
                }
        }
        B[pos] = b;
        B1[pos] = -1;
        for(i=0; i<x; i++){
            if(B[pos] == A[i]){
                    B1[pos] = tb;
                    tb += 1;
                    break;
            }
        }
        return tb;
}

void results(int *A ,  int *B , int x , int y , int *count1){
     int i , i2 , count;
```

```
    count = 0;
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if(A[i] == B[i2]){
                count++;
            }
        }
    }
    count += *count1;
    printf("Ta apotelesmata sou einai: %d join\n", count);
    scanf("%d\n", &i);
}
```

A.3 Ο αλγόριθμος LFU (Less Frequency Used)

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBERS 40
void getStart(int * , int * , int * , int * , int , int);
void insertA(int * , int * , int * , int * , int , int , int *);
void insertB(int * , int * , int * , int * , int , int , int *);
void results(int * , int , int *);

main(){
    int i , x , y , PA , PB , count1=0;
    int *A , *B , *A1 , *B1;
    printf("Dose posa insertA theleis na exeis\n");
    scanf("\n%d", &PA);
    printf("Dose posa insertB theleis na exeis\n");
    scanf("\n%d", &PB);
    printf("Dose poses grames theleis na exei o pinakas A\n");
    scanf("\n%d", &x);
    printf("Dose poses grames theleis na exei o pinakas B\n");
    scanf("\n%d", &y);
    A = (int *)malloc(x * sizeof(int));
    A1 = (int *)malloc(x * sizeof(int));
    B = (int *)malloc(y * sizeof(int));
```

```
    B1 = (int *)malloc(y * sizeof(int));
    getStart(A , A1 , B , B1 , x , y);
    for(i=x; i<PA; i++){
        insertA(A , A1 , B , B1 , x , y , &count1);
    }
    for(i=y; i<PB; i++){
        insertB(B , B1 , A , A1 , x , y , &count1);
    }
    results(A1 , x , &count1);
}

void getStart(int *A , int *A1 , int *B , int *B1 , int x , int y){
    int i , i2 , c=0 , a , b;
    A[0] = rand()% NUMBERS+1;
    B[0] = rand()% NUMBERS+1;
    for(i=1; i<x; i++){
        a = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(A[i2] == a){
                i--;
                c=1;
                break;
            }
        }
```

```
        if(c == 0){
            A[i] = a;
        }
        c = 0;
}
c = 0;
for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
                if(B[i2] == b){
                        i--;
                        c=1;
                        break;
                }
        }
        if(c == 0){
            B[i] = b;
        }
        c = 0;
}
for(i=0; i<x; i++){
        A1[i] = 0;
}
for(i=0; i<y; i++){
```

```
            B1[i] = 0;
        }
        for(i=0; i<x; i++){
            for(i2=0; i2<y; i2++){
                if(A[i] == B[i2]){
                    A1[i] += 1;
                    B1[i2] += 1;
                }
            }
        }
}

void insertA(int *A , int *A1 , int *B , int *B1 , int x, int y , int *count1){
    int i , i2 , ak , pos , a;
    a = rand()% NUMBERS+1;
    for(i=0; i<x; i++){
        if(A[i] == a){
            for(i=0; i<y; i++){
                if (B[i] == a){
                  *count1 += 1;
                  break;
                }
            }
            a = rand()% NUMBERS+1;
```

```
            i = -1;
        }
}
ak = A1[0];
pos = 0;
for(i=1; i<x; i++){
        if (A1[i]<ak){
          ak = A1[i];
          pos = i;
        }
}
A[pos] = a;
A1[pos] = 0;
for(i=0; i<y; i++){
        if (A[pos] == B[i]){
          A1[pos] += 1;
        }
}
for(i=0; i<y; i++){
        B1[i] = 0;
}
for(i2=0; i2<y; i2++){
        for(i=0; i<x; i++){
              if (B[i2] == A[i]){
```

```
                B1[i2] += 1;
            }
        }
    }
}

void insertB(int *B , int *B1 , int *A , int *A1 , int x, int y , int *count1){
    int i , i2 , ak , pos , b;
    b = rand()% NUMBERS+1;
    for(i=0; i<y; i++){
        if(B[i] == b){
            for(i=0; i<x; i++){
                if (A[i] == b){
                  *count1 += 1;
                  break;
                }
            }
            b = rand()% NUMBERS+1;
            i = -1;
        }
    }
    ak = B1[0];
    pos = 0;
    for(i=1; i<y; i++){
```

```
        if (B1[i]<ak){
          ak = B1[i];
          pos = i;
        }
    }
    B[pos] = b;
    B1[pos] = 0;
    for(i=0; i<x; i++){
        if (B[pos] == A[i]){
          B1[pos] += 1;
        }
    }
    for(i=0; i<x; i++){
        A1[i] = 0;
    }
    for(i=0; i<x; i++){
        for(i2=0; i2<y; i2++){
            if (A[i] == B[i2]){
              A1[i] += 1;
            }
        }
    }
}
```

```
void results(int *A1 , int x , int *count1){
    int i , count=0;
    for(i=0; i<x; i++){
            count += A1[i];
    }
    count += *count1;
    printf("Ta apotelesmata sou einai: %d join\n", count);
    scanf("%d\n", &i);
}
```

A.4 Ο αλγόριθμος LRU (Less Recently Used)

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBERS 240
void getStart(int * , int * , int , int);
void insertA(int * , int * , int , int , int *);
void insertB(int * , int * , int , int , int *);
void results(int * , int * , int , int , int *);

main(){
    int i , x , y , PA , PB , count1=0;
    int *A , *B;
    printf("Dose posa insertA theleis na exeis\n");
    scanf("\n%d", &PA);
    printf("Dose posa insertB theleis na exeis\n");
    scanf("\n%d", &PB);
    printf("Dose poses grames theleis na exei o pinakas A\n");
    scanf("\n%d", &x);
    printf("Dose poses grames theleis na exei o pinakas B\n");
    scanf("\n%d", &y);
    A = (int *)malloc(x * sizeof(int));
    B = (int *)malloc(y * sizeof(int));
```

```
    getStart(A , B , x , y);
    for(i=x; i<PA; i++){
        insertA(A , B , x , y , &count1);
    }
    for(i=y; i<PB; i++){
        insertB(B , A , x , y , &count1);
    }
    results(A , B , x , y , &count1);
}

void getStart(int *A , int *B , int x , int y){
    int i , i2 , a , b , c=0;
    A[0] = rand()% NUMBERS+1;
    B[0] = rand()% NUMBERS+1;
    for(i=1; i<x; i++){
        a = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
            if(A[i2] == a){
                for( ; i2<i; i2++){
                    A[i2] = A[i2+1];
                }
                A[i-1] = a;
                i--;
                c = 1;
```

```
                    break;
                }
        }
        if(c == 0){
            A[i] = a;
        }
        c = 0;
}
c = 0;
for(i=1; i<y; i++){
        b = rand()% NUMBERS+1;
        for(i2=0; i2<i; i2++){
                if(B[i2] == b){
                        for( ; i2<i; i2++){
                            B[i2] = B[i2+1];
                        }
                        B[i-1] = b;
                        i--;
                        c = 1;
                        break;
                }
        }
        if(c == 0){
            B[i] = b;
```

```
            }
            c = 0;
     }
}

void insertA(int *A , int *B , int x , int y , int *count1){
   int i , c=0 , a;
   a = rand()% NUMBERS+1;
   for(i=0; i<x; i++){
         if(A[i] == a){
               for ( ; i<x; i++){
                  A[i] = A[i+1];
               }
               for(i=0; i<y; i++){
                     if (B[i] == a){
                       *count1 += 1;
                       break;
                     }
               }
               A[x-1] = a;
               c = 1;
               break;
         }
   }
```

```
    if(c == 0){
        for(i=0; i<x; i++){
                A[i] = A[i+1];
        }
        A[x-1] = a;
    }
    for(i=0; i<y; i++){
            if(B[i] == a){
                for( ; i<y; i++){
                    B[i] = B[i+1];
                }
                B[y-1] = a;
                break;
            }
    }
}

void insertB(int *B , int *A , int x , int y , int *count1){
    int i , c=0 , b;
    b = rand()% NUMBERS+1;
    for(i=0; i<y; i++){
            if(B[i] == b){
                for ( ; i<y; i++){
                    B[i] = B[i+1];
```

```
            }
            for(i=0; i<x; i++){
                if (A[i] == b){
                  *count1 += 1;
                  break;
                }
            }
            B[y-1] = b;
            c = 1;
            break;
        }
}
if(c == 0){
    for(i=0; i<y; i++){
            B[i] = B[i+1];
    }
    B[y-1] = b;
}
for(i=0; i<x; i++){
        if(A[i] == b){
            for( ; i<x; i++){
                A[i] = A[i+1];
            }
            A[x-1] = b;
```

```
                break;
            }
        }
    }

void results(int *A ,  int *B , int x , int y , int *count1){
    int i , i2 , count;
    count = 0;
    for(i=0; i<x; i++){
            for(i2=0; i2<y; i2++){
                    if(A[i] == B[i2]){
                            count++;
                    }
            }
    }
    count += *count1;
    printf("Ta apotelesmata sou einai: %d join\n", count);
    scanf("%d\n", &i);
}
```

Βιβλιογραφία

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 221–232, 2002.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[3] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, 2001.

[4] D. Barbar´a, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. Int. Conf. on Mobile Data Management (MDM)*, pages 3–14, 2001.

[6] D. Carney, U. C¸ etintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams — a new class of data management applications. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.

[7] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.

[8] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274, 1999.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 379 390, 2000.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 635–644, 2002.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[12] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 79–88, 2001.

[13] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 58–65, 2001.

[15] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. ACM Symp. on the Theory of Computing (STOC)*, pages 471–475, 2001.

[16] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. http://cdiac.esd.ornl.gov/ftp/ndp026b, 1996.

[17] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.

[18] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.
[19] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, pages 174–185, 1999.
[20] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–310, 1999.
[21] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2003.
[22] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *Proc. Int. Conf. on Very Large Databases (VLDB)*, 2002.
[23] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 2002.
[24] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.
[25] R. T. Rockafellar. *Network flows and monotropic optimization*. John Wiley & Sons, 1984.
[26] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databaases. In *Proc. Int. Conf. on Computer Vision (ICCV)*, pages 207–214, 1998.
[27] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.
[28] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 2 edition, 1979.