2η Εργασία

ΕΠΑΜΕΙΝΩΝΔΑΣ ΙΩΑΝΝΟΥ(3140059)

ΑΛΕΞΑΝΔΡΟΣ ΚΟΝΟΜΗΣ(3140085)

ΑΝΑΦΟΡΑ ΠΑΡΑΔΟΣΗΣ

Η εργασία σχετίζεται με αλγορίθμους ταξινόμησης, τον ΑΤΔ της ουράς προτεραιότητας και προβλήματα βελτιστοποίησης σχετικά με την ανάθεση πόρων μνήμης για αποθήκευση δεδομένων μεγάλου όγκου. Συγκεκριμένα αποτελείται από 4 μέρη. Το Μέρος Α αποτελεί την αρχικοποίηση των ΑΤΔ που χρειάζονται για την εργασία, το Μέρος Β την υλοποίηση του αλγορίθμου Greedy, το Μέρος Γ την υλοποίηση του αλγορίθμου Greedy-Decreasing και τέλος στο Μέρος Δ παρουσιάζεται μια πειραματική αξιολόγηση. Σημειώνεται ότι τα αντίστοιχα προγράμματα που δημιουργήθηκαν για την ολοκλήρωση της εργασίας μεταγλωττίστηκαν κι εκτελέστηκαν μέσω κονσόλας(Windows cmd). Παρακάτω γίνεται επεξήγηση της λειτουργίας, των ιδεών και του κώδικα που χρησιμοποιήθηκε για το κάθε μέρος χωριστά.

Μέρος Α: Στο πρώτο μέρος όπως προαναφέρθηκε κατασκευάζονται οι ΑΤΔ που χρειάζονται για την εργασία, δηλαδή η ουρά προτεραιότητας(MaxPQ.java) και ο σκληρός δίσκος(Disk.java) που θα είναι το στοιχείο **<T>** που θα εισάγεται. Όσον αφορά την MaxPQ έγινε προσαρμογή στην υπάρχουσα του εργαστηρίου και χρήση **Generics <T>**, δηλαδή λειτουργεί για οποιονδήποτε τύπο δεδομένων. Αρχικά οι μεταβλητές της κλάσης είναι οι private T[] PQ στην οποία θα σώζονται οι δίσκοι με χρήση σωρού και κατά σύμβαση το root είναι στη θέση 1 , και η μεταβλητή private int size που αποτελεί το μέγεθος της ουράς προτεραιότητας και αρχικοποιείται σε 0 μέσω του κατασκευαστή. **Αφαιρέθηκε η μέθοδος grow**, καθώς δίνεται μέγιστο μέγεθος για τον πίνακα οπότε δεν υπάρχει περίπτωση υπέρβασης και έτσι ανάγκη για να φτιάξουμε μεγαλύτερο πίνακα. Οι μέθοδοι **isempty**(επιστρέφει true αν η ουρά προτεραιότητας δεν έχει στοιχεία), **insert(** τοποθετεί ένα στοιχείο στο τέλος της ουράς προτεραιότητας και το κάνει swim προς τα πάνω),**peek**(επιστρέφει το στοιχείο με τη μέγιστη προτεραιότητα αλλά δεν το αφαιρεί),**getMax**(αφαιρεί κι επιστρέφει το στοιχείο με τη μέγιστη προτεραιότητα)**,** **swim**(«ανεβάζει» προς τα πάνω το στοιχείο που προστίθεται στο τέλος του πίνακα, για να αποκατασταθεί η τάξη στο σωρό), **sink**(«βυθίζει» το στοιχείο προς τα κάτω) και η **swap**(κάνει ανταλλαγή των στοιχείων στα indexes I και j) **έχουν παραμείνει όπως ήταν στον κώδικα του εργαστηρίου**. Η μόνη μέθοδος που προστέθηκε είναι η public void refresh() { sink(1); } και ανανεώνει τη λίστα κάνοντας sink το root για να αποκατασταθεί ο η ισορροπία στο σωρό. Η συγκεκριμένη μέθοδος υπάρχει για χρήση στο αρχείο Greedy.java και καλείται μετά την εισαγωγή ενός φακέλου στο δίσκο με τον περισσότερο

ελεύθερο χώρο, καθώς πλέον ο συγκεκριμένος δίσκος πιθανότατα να μην έχει τη μέγιστη προτεραιότητα και πρέπει να αντικατασταθεί με τον επόμενο με τον περισσότερο ελεύθερο χώρο. Οπότε την καλούμε και τον κάνουμε sink.

Μέρος Β: Στο μέρος αυτό γίνεται η υλοποίηση του αλγορίθμου Greedy στο πρόγραμμα Greedy.java. Το πρόγραμμα αποτελείται από τη main, την **GreedyAlgorithm,** την **TextFileCheck** και την **Print**. Η main αρχικά αποθηκεύει σε μια μεταβλητή String file_path=args[0] το path του textfile που δίνεται μέσω κονσόλας. Έπειτα καλείται η **TextFileCheck(file_path**) με όρισμα το μονοπάτι του textfile για να εξετάσουμε αν το αρχείο βρίσκεται στη σωστή μορφή. Στο εσωτερικό της μεθόδου αυτής κατασκευάζουμε αντίστοιχα αντικείμενα για την επεξεργασία του textfile(**BufferedReader και Scanner**) και διαβάζουμε κάθε γραμμή του file για να δούμε αν είναι ακέραιος μεταξύ του 0 και 1.000.000. Αν βρεθεί γραμμή που δεν περιέχει ακέραιο τότε τυπώνεται η αντίστοιχη εξαίρεση και σταματά το πρόγραμμα. Αν κάποιος ακέραιος δε βρίσκεται στο επιθυμητό μέγεθος τυπώνει αντίστοιχο μήνυμα λάθους και τερματίζει το πρόγραμμα. Σε διαφορετική περίπτωση το text file είναι στη σωστή μορφή και φτιάχνεται ένας πίνακας ακεραίων μέσα στην TextFileCheck, ο οποίος περιέχει τους φακέλους με τη σειρά που βρίσκονται στο textfile. Ο πίνακας αυτός γίνεται return στη main. Αφού έχει γίνει επιστροφή στη main καλείται η μέθοδος **GreedyAlgorithm(table)** με όρισμα τον πίνακα αυτόν. Κατασκευάζεται ουρά προτεραιότητας τύπου Disk και της δίνεται μέγιστο μέγεθος το σύνολο των φακέλων για να μην υπάρχει περίπτωση υπερχείλισης**:MaxPQ<Disk> PQ=new MaxPQ<>(table.length+1).** Δημιουργείται ο πρώτος δίσκος με id 0 που θα γίνει insert στο MaxPQ Disk **disk0=new Disk(0)**. Αρχικοποιούνται επίσης μεταβλητές που θα φανούν χρήσιμες στη συνέχεια όπως η NumberOfDisks που κρατάει το πλήθος των δίσκων που χρησιμοποιήθηκαν στο text file και η folders_size που κρατάει το συνολικό μέγεθος όλων των φακέλων. Στη συνέχεια προσθέτουμε τον πρώτο φάκελο στον πρώτο δίσκο: **disk0.addFileToTheList(table[0])** μέσω της public μεθόδου της κλάσης Disk addFileToTheList(η οποία όταν καλείται καλεί και την private μέθοδο reduce_size για να μειώσει το μέγεθος του ελεύθερου χώρου του δίσκου). Ο πρώτος δίσκος προστίθεται ως πρώτο στοιχείο στην ουρά προτεραιότητας μέσω της μεθόδου insert της MaxPQ που χει περιγραφεί πιο πάνω **PQ.insert(disk0).** Μετά γίνεται χρήση for loop αρχίζοντας από το δεύτερο φάκελο(table[1]) και φτάνοντας στον τελευταίο για να προσθέσουμε τους φακέλους σε δίσκους. Σε κάθε επανάληψη αυξάνουμε την μεταβλητή folders_size προσθέτοντας το μέγεθος του τωρινού φακέλου και ελέγχουμε αν το current file(table[i]) είναι μικρότερο από το χώρο του δίσκου με τον περισσότερο ελεύθερο χώρο ως εξής: **if(table[i]<=PQ.peek().getFreeSpace()).** Αν ισχύει κάτι τέτοιο προσθέτουμε στο συγκεκριμένο δίσκο το current file **PQ.peek().addFileToTheList(table[i]).** Σε διαφορετική περίπτωση θα χρειαστεί να φτιάξουμε νέο δίσκο(οπότε αυξάνεται και η μεταβλητή NumberOfDisks κατά 1), να τον κάνουμε insert στο PQ και να προσθέσουμε εκεί το current file με τις εξής εντολές: **α) NumberOfDisks++; β) Disk disk=new Disk(NumberOfDisks-1); γ) PQ.insert(disk); και δ) disk.addFileToTheList(table[i]);.** Στο τέλος κάθε επανάληψης καλούμε τη public μέθοδο refresh της MaxPQ για να ανανεώσουμε τη λίστα, καθώς υπάρχει πιθανότητα μετά από εισαγωγή ενός φακέλου στο δίσκο με τη μέγιστη

προτεραιότητα να υπάρχει νέος δίσκος στην ουρά με περισσότερο ελεύθερο χώρο. Έξω από το for loop καλείται η μέθοδος Print που πραγματοποιεί εκτύπωση στη μορφή που ζητείται. Τέλος, η GreedyAlgorithm επιστρέφει το πλήθος των δίσκων που χρησιμοποιήθηκαν στο textfile.

Μέρος Γ: Όσον αφορά τον αλγόριθμο Greedy-Decreasing το μόνο που χρειάζεται είναι πρώτα να διαταχθούν οι φάκελοι με φθίνουσα σειρά και μετά να υλοποιηθεί ο Greedy. Στο σημείο αυτό θα περιγραφεί ο αλγόριθμος **MergeSort** που χρησιμοποιήθηκε στο Sort.java για να γίνει η συγκεκριμένη ταξινόμηση. Το πρόγραμμα έχει δύο static μεθόδους που επιστρέφουν ταξινομημένο πίνακα: την **public static int[] MergeSort(int [] array)** και την **private static int[] Merge(int[] right, int[]left).** Η Mergesort λειτουργεί με αναδρομή. Η βασική περίπτωση είναι αν το μέγεθος του πίνακα που δίνεται ως όρισμα είναι 1 ή 0 όπου επιστρέφεται ο πίνακας. Αρχικά χωρίζουμε τον πίνακα στη μέση βρίσκοντας το μεσαίο σημείο με int midpoint=array.length/2. Αρχικοποιούμε το μέγεθος του αριστερού υποπίνακα με μέγεθος midpoint int[]left=new int[midpoint]. Αν το μέγεθος του αρχικού array είναι άρτιο τότε ο δεξιός υποπίνακας έχει μέγεθος midpoint(int[] right=new int [midpoint]), αλλιώς midpoint+1(int[] right=new int[midpoint+1]). Έπειτα δίνουμε τις τιμές του πίνακα από το δείκτη 0 ως midpoint-1 στον αριστερό υποπίνακα και από το δείκτη midpoint ως το array.length-1 στο δεξιό υποπίνακα. Στη συνέχει μέσω αναδρομής στο δεξιό και αριστερό υποπίνακα, **right=MergeSort(right) και left=MergeSort(left)** ταξινομούνται οι δύο υποπίνακες και έπειτα τους συγχωνεύουμε σε έναν μέσω της **Merge.** Η Merge παίρνει ως όρισμα τους δύο υποπίνακες και επιστρέφει τον ταξινομημένο πίνακα που χει συγχωνευτεί από τους δύο υποπίνακες(που είναι ήδη ταξινομημένοι λόγω αναδρομής, δηλ. προηγουμένως οι υποπίνακες αυτών έχουν συγχωνευτεί μέσω της Merge Κλπ…). Στο εσωτερικό υπάρχει ένα while loop που έχει ως συνθήκη ότι είτε έχει ακόμα στοιχεία ο αριστερός υποπίνακας που δεν έχουν τοποθετηθεί στο νέο πίνακα είτε ο δεξής. Έπειτα συγκρίνουμε τα στοιχεία των δύο υποπινάκων και το μεγαλύτερο απ αυτά τοποθετείται στο νέο πίνακα ώστε να έχουμε φθίνουσα σειρά. Με την έξοδο απ το while loop η μέθοδος επιστρέφει το συγχωνευμένο ταξινομημένο πίνακα στη MergeSort και τον επιστρέφει κι αυτή με τη σειρά της. Αναλυτικές λεπτομέρειες σχετικά με τον κώδικα υπάρχουν στα σχόλια του Sort.java.

Μέρος Δ: Στο μέρος Δ γίνεται πειραματική αξιολόγηση για να διαπιστωθεί ποιος αλγόριθμος είναι πιο αποδοτικός στην πράξη. Γι αυτή την αξιολόγηση δημιουργήθηκαν 2 προγράμματα: το **RandomInput.java** που παράγει τα αρχεία με φακέλους τυχαίου μεγέθους και το **AlgorithmComp.java** που συγκρίνει την απόδοση του Greedy και του Greedy-Decreasing στα text files που παράχθηκαν προηγουμένως. Το πρόγραμμα RandomInput.java κάνει αρχικά import το java.util.Random και το java.io.* για χρήση της κλάσης Random και παραγωγή text files αντίστοιχα. Επίσης, το πρόγραμμα δέχεται 2

ορίσματα από κονσόλα, την τιμή N(πλήθος φακέλων σε κάθε text file) και το πλήθος των Text Files που θα δημιουργήσουμε. Αποθηκεύονται σε αντίστοιχες ακέραιες μεταβλητές ως εξής: **int N=Integer.parseInt(args[0]) και int Text_Files=Integer.parseInt(args[1])**. Έπειτα φτιάχνουμε ένα αντικείμενο Random καθώς το χρειαζόμαστε για παραγωγή τυχαίων αριθμών. Μετά αρχικοποιούμε μια μεταβλητή text_file_number=1 η οποία θα χρησιμοποιηθεί για τα ονόματα των text_files(το πρώτο το ονομάζουμε TEXTFILE(1), το δεύτερο TEXTFILE(2) κλπ). Στη συνέχεια υπάρχει ένα for loop που αρχίζει από το 1(1ος φάκελος) μέχρι την ακέραια μεταβλητή Text_Files(τελευταίος φάκελος). Σε κάθε επανάληψη αυξάνεται επίσης και η μεταβλητή text_file_number κατά 1 καθώς φτιάχνουμε φάκελο με την εξής μορφή**: File file=new File("TEXTFILE("+text_file_number+").txt")**, οπότε στην πρώτη επανάληψη φτιάχνει φάκελο με όνομα TEXTFILE(1) στη δεύτερη TEXTFILE(2) κ.οκ. Η επόμενη εντολή περιέχει τη δημιουργία FileWriter αντικειμένου για να μπορούμε να γράψουμε στο αρχείο που μόλις δημιουργήσαμε. Ύστερα έχουμε ακόμα μια for που αρχίζει απ το 1(πρώτος φάκελος) και φτάνει ως N-1(τελευταίος φάκελος) όπου N το πλήθος των φακέλων για το κάθε text file που χει δοθεί ως όρισμα args[0]. Σε κάθε επανάληψη(δηλαδή σε κάθε γραμμή του αντίστοιχου text file) θέλουμε μια τυχαία τιμή από 0 ως 1.000.000 για να αναπαριστούμε το μέγεθος σε MB του κάθε φακέλου. Αυτό το επιτυγχάνουμε ως εξής: **writer.write(r.nextInt(1000001)+"\n").** Στην περίπτωση που είμαστε στην τελευταία επανάληψη(δηλαδή στο N-1) έχουμε την εξής εντολή: **writer.write(r.nextInt(1000001)+"\n"+r.nextInt(1000001))** που παράγει τυχαία τιμή για τον N-1 φάκελο και τυχαία τιμή για τον τελευταίο φάκελο στην επόμενη(τελευταία) γραμμή του text file. Η ίδια διαδικασία εκτελείται για το επόμενο text file, μέχρι να φτάσουμε στο τελευταίο. Ο αριθμός επαναλήψεων δηλαδή της πρώτης for εξαρτάται από το όρισμα args[1](δηλαδή πόσα text files θέλουμε) και της δεύτερης for από το όρισμα args[0](πλήθος φακέλων σε κάθε text file). Εμείς δημιουργήσαμε 20 text files για κάθε τιμή N. Το δεύτερο πρόγραμμα που δημιουργήθηκε για το συγκεκριμένο μέρος είναι το AlgorithmComp.java όπως προαναφέρθηκε και συγκρίνει τους δύο αλγορίθμους. Στο πρόγραμμα αυτό αρχικοποιούμε 2 μεταβλητές που παριστάνουν τους συνολικούς δίσκους που χρησιμοποίησε ο Greedy και τους αντίστοιχους που χρησιμοποίησε ο Greedy-Decreasing ως εξής: int Total_Disks_Decreasing=0 και int Total_Disks_Greedy=0. Έπειτα αρχικοποιούμε 2 μεταβλητές που θα παριστάνουν τους δίσκους που χρησιμοποιούν οι 2 αλγόριθμοι σε κάθε text file ξεχωριστά ως εξής: int Disks_Decreasing=0 και int Disks_Greedy=0. Μετά έχουμε μια for η οποία θα «τρέξει» για 20 επαναλήψεις(καθώς δημιουργούμε 20 text files για κάθε τιμή N με τη RandomInput) και θα συγκρίνει την απόδοση των αλγορίθμων σε κάθε text file. Ελέγχουμε αν κάθε text file είναι σε σωστή μορφή ως εξής: **int[] table=Greedy.TextFileCheck("TEXTFILE("+text_file_number+").txt").**

Η TextFileCheck είναι static οπότε χρησιμοποιούμε το όνομα της κλάσης για να την καλέσουμε κι επιστρέφει και τον πίνακα που περιέχει τα μεγέθη των φακέλων αν το text file είναι σε σωστή μορφή. Σώζουμε τον πίνακα αυτόν στον table[]. Έπειτα θέλουμε να καταγράψουμε πόσους δίσκους χρησιμοποίησε ο Greedy και ο Greedy-Decreasing για το συγκεκριμένο text file. Για τον Greedy καλούμε την GreedyAlgorithm με όρισμα τον πίνακα table[] απευθείας και σώζουμε στην Disks_Greedy τους δίσκους που χρησιμοποιήθηκαν ως εξής: **Disks_Greedy=Greedy.GreedyAlgorithm(table).** Σημειώνεται ότι η GreedyAlgorithm

κάνει return τους δίσκους που χρησιμοποίησε για συγκεκριμένο text file οπότε αποθηκεύουμε την επιστρεφόμενη τιμή στην Disks_Greedy. Στη συνέχεια το μόνο που πρέπει να κάνουμε για να υλοποιηθεί ο Greedy_Decreasing είναι να ταξινομήσουμε σε φθίνουσα σειρά τους φακέλους και μετά να καλέσουμε την GreedyAlgorithm με όρισμα το νέο ταξινομημένο πίνακα. Αυτό το επιτυγχάνουμε ως εξής: **table=Sort.MergeSort(table).** Η MergeSort είναι static οπότε αντίστοιχα κι αυτή την καλούμε με το όνομα της κλάσης. Επιστρέφει τον ταξινομημένο πίνακα που τον αποθηκεύουμε στο table. Επομένως τώρα που έχουμε ταξινομημένους τους φακέλους σε φθίνουσα σειρά ξανακαλούμε την GreedyAlgorithm και αποθηκεύουμε και πάλι τους δίσκους που χρησιμοποιεί ως εξής:**Disks_Decreasing=Greedy.GreedyAlgorithm(table).** Τώρα έχουμε καταγράψει τους δίσκους που χρησιμοποίησε ο Greedy και ο Greedy Decreasing στο συγκεκριμένο text file και το μόνο που πρέπει να κάνουμε είναι να τους συγκρίνουμε αριθμητικά για να συμπεράνουμε ποιος αλγόριθμος είναι πιο αποδοτικός στην πράξη. Με τη βοηθητική static μέθοδο compare που δέχεται ορίσματα δύο ακεραίους και τους συγκρίνει ,δίνουμε ως ορίσματα τα 2 πλήθη δίσκων για το συγκεκριμένο αρχείο. Αν το πλήθος δίσκων που χρησιμοποίησε ο Greedy_Decreasing είναι μικρότερο από το αντίστοιχο του Greedy τότε προφανώς και είναι πιο αποδοτικός από τον Greedy, οπότε τυπώνεται και αντίστοιχο μήνυμα. Σε κάθε επανάληψη του for loop αυξάνουμε και τις μεταβλητές Total_Disks_Decreasing και Total_Disks_Greedy(που όπως προαναφέρθηκε είναι μεταβλητές για το συνολικό πλήθος των δίσκων για κάθε αλγόριθμο σε όλα τα text files) ως εξής: **Total_Disks_Greedy+=Disks_Greedy και Total_Disks_Decreasing+=Disks_Decreasing.** Μετά την έξοδο απ το for θέλουμε επίσης να συγκρίνουμε το μέσο όρο δίσκων που χρησιμοποίησε ο κάθε δίσκος για να δούμε τελικά συνολικά για όλα τα text files ποιος αλγόριθμος απ τους δύο είναι πιο αποδοτικός. Αρχικά εκτυπώνουμε για κάθε αλγόριθμο το μέσο όρο δίσκων που χρησιμοποίησε **διαιρώντας την Total_Disks_Greedy και την Total_Disks_Decreasing με 20**(αφού τα text files που έχουμε δημιουργήσει για κάθε τιμή του N είναι 20). Τέλος δίνουμε ως όρισμα και πάλι στη βοηθητική συνάρτηση compare τις 2 αυτές μεταβλητές για να βγάλουμε τελικό συμπέρασμα για όλα τα text files. Αν η Total_Disks_Decreasing<Total_Disks_Greedy τότε ο αλγόριθμος Greedy_Decreasing συνολικά για όλα τα text files είναι πιο αποδοτικός και τυπώνεται αντίστοιχο μήνυμα με το τελικό συμπέρασμα. Σε αντίθετη περίπτωση γίνεται το ίδιο για τον Greedy.

Ευρήματα: Παρακάτω παρουσιάζονται οι μέσοι όροι δίσκων που παρήγαγε κάθε αλγόριθμος για κάθε τιμή του N. Υπενθύμιση: Για την εργασία δημιουργήθηκαν 20 text files για κάθε τιμή του N (50,100,250,500,750,1000,2000,5000). Έγινε σύγκριση των δύο αλγορίθμων για κάθε text file χωριστά(πλήθος δίσκων) και στο τέλος παρουσιάζονται οι μέσοι όροι για όλα τα text files του κάθε αλγορίθμου. Στη συγκεκριμένη αναφορά εμφανίζεται μόνο το δεύτερο.

Για N=50

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 30.1
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 27.55
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=100

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 58.75
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 52.8
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=250

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 148.1
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 130.25
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=500

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 295.25
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 258.0
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=750

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 440.0
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 382.55
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=1000

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 586.75
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 507.7
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=2000

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 1172.0
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 1011.35
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Για N=5000

```
Comparison:Greedy-Decreasing Algorithm is more efficient than Greedy Algorithm in TEXTFILE(20)

Average of disk files used in all text files by Greedy Algorithm: 2925.8
Average of disk files used in all text files by Greedy-Decreasing Algorithm: 2515.9
Final Comparison: Greedy-Decreasing Algorithm is more efficient

C:\Users\nwntas\Documents\Δομές Δεδομένων\Εργασία 2\src>
```

Συμπέρασμα: Το τελικό και προφανές συμπέρασμα είναι ότι ο Greedy_Decreasing δίχως αμφιβολία είναι πιο αποδοτικός από τον Greedy. Παρατηρήθηκε επίσης ότι για μικρές τιμές του N ο Greedy Decreasing **παρήγαγε κατά 10 τοις εκατό λιγότερους δίσκους** από τον Greedy ενώ όσο αυξανόταν το N οι δίσκοι **που παρήγαγε ήταν κατά 15 τοις εκατό λιγότεροι, με το ποσοστό αυτό να αρχίζει να σταθεροποιείται.**