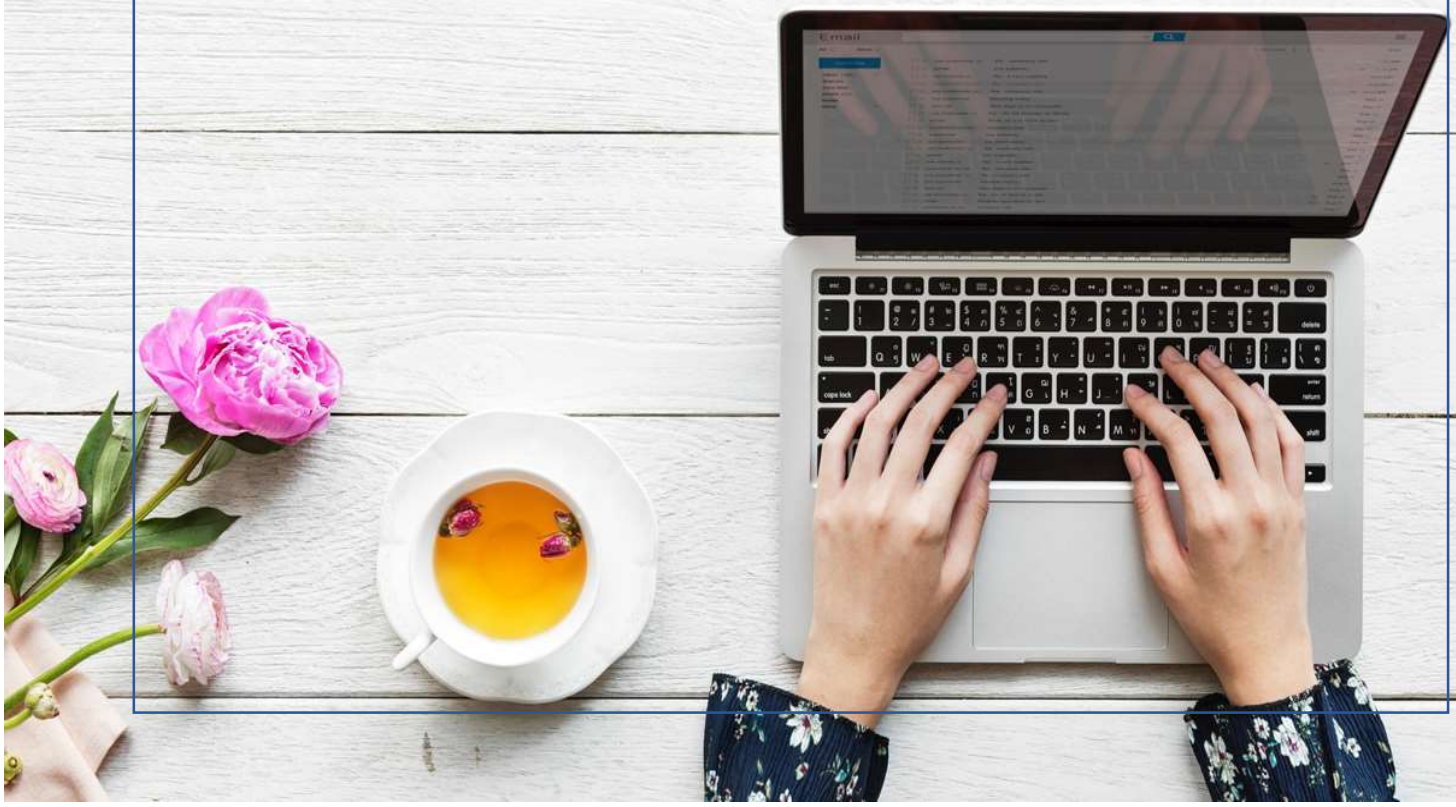
Πέτσα Γεωργία 3200155
Παναγιώτης Τριανταφυλλίδης 3200199

Δεύτερη εργασία στις Δομές Δεδομένων 2021-2022



ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΜΕΡΟΣ Α

Κλάση Processor:

Ιδιότητες

- Κάθε επεξεργαστής αρχικοποιείται με ένα μοναδικό id
- Έχει μια μεταβλητή sumtime η οποία περιέχει το σύνολο του χρόνου που έχει ξοδέψει κάθε επεξεργαστής
- Και τέλος μια λίστα ProcessedTasks η οποία περιέχει τα Tasks του συγκεκριμένου επεξεργαστή.

Μεθόδους

- ✓ Περιέχει έναν constructor ο οποίος αρχικοποιεί με ένα μοναδικό id τον κάθε processor
- ✓ Περιέχει την getActiceTime που επιστρέφει το sumtime του επεξεργαστή
- ✓ Υλοποιεί τη μέθοδο compareTo της διεπαφης Comparable, η οποία επιστρέφει τον αριθμό 1 εάν ο επεξεργαστής A έχει μικρότερο Active Time (Συνολικό χρόνο) από τον επεξεργαστή που παίρνει ως όρισμα η compareTo. (Σε περίπτωση ισότητας επιστρέφεται 1 ή -1 ανάλογα με τον αν ο επεξεργαστής A έχει μικρότερο id ή μεγαλύτερο id αντίστοιχα).
- ✓ Υλοποιεί την intertAtLists(Task t) η οποία προσθέτει στο τέλος της λίστας ProcessedTasks το καινούριο Task, και ενημερώνει το sumtime με τον αντίστοιχο χρόνο του Task.
- ✓ Την printProcessor() η οποία τυπώνει το id του επεξεργαστή, τα δεδομένα του Task και τον αντίστοιχο χρόνο του.

Κλάση Task:

Ιδιότητες

- ➢ Κάθε αντικείμενο τύπου task έχει ένα μοναδικό id και έναν χρόνο.

Μεθόδους

- Τον Constructor που αρχικοποίησή ένα task με το id του και τον χρόνο του
- getID που επιστρέφει το id
- getTime που επιστρέφει το time

Κλάση MaxPQ :

(Η υλοποίηση της έγινε με generics και έχει βασιστεί στον κώδικα του φροντιστηρίου 6.)

Ιδιότητες:

- Αρχικά χρησιμοποιούμε έναν πίνακα heap οπού μέσα αποθηκεύουμε τα δεδομένα

- Η μεταβλητή size κρατάει το χώρο του σωρού
- DEFAULT_CAPACITY = 4

Μεθοδους:

Αρχικα έχουμε υλοποιησει το ***PQInterface*** και

Οι πιο σημαντικες μεθοδοι είναι:

- Insert(T x) η οποια εισαγει στο πινακα heap στη θεση size το εντικειμενο x αφου πρωτα εχει αυξησει το size, και επειτα καλει την swim. (Το size κάθε φορα δειχνει τη θεση που θα πρεπει να μπει το προς εισαγωγη στοιχειο) Επισης η θεση 0 του πινακα heap για λογους απλοτητας δεν εχει κανενα στοιχειο.
  Η resize καλειται όταν έχουμε ξεπερασει το 75% του μεγεθους του πινακα heap
- Resize() η οποια δημιουργει έναν καινουριο πινακα ο οποιος εχει το διπλασιο μεγεθος από τον προηγουμενο και εισαγει με μια for τα στοιχεια του πρωτηου πινακα στον δευτερο και επειτα κανει τον πινακα heap να δειχνει στον καινουριο πινακα.
- Swap(int I, int j) που απλα αντιμεταθετει 2 στοιχεια του πινακα
- Swim(int i).
  Αν το I ==1 τοτε δεν κανει τιποτα. Αν δεν είναι, ουσιαστικα πρεπει το στοιχείο να μπει στην σωστη του θεση, οποτε αν ο χρονος του heap[i] είναι μικροτερος από τον χρόνο του «μπαμπα» του, τοτε πρεπει να γινει το heap[i] πατερας, οποτε τα αντιστρεφουμε και βαζουμε στην μεταβλητη I το i/2. Η διαδικασια αυτή γινεται οσο το i>1 και ο χρονος του παιδιου είναι μικροτερος από τον χρονο του πατερα (Μεγιστοστρεφης σωρος). Ετσι κάθε φορα στην θεση heap[1] θα βρισκεται το στοιχειο με τη μεγιστη προτεραιοτητα(Δηλαδη αυτό με τον μικροτερο χρονο)
- Η max επιστρεφει το heap[1].
- Getmax() η οποια αποθηκευει σε μια μεταβλητη max την ριζα, ανταλασσει τη ριζα με το τελευταιο στοιχειο του πινακα, μειωνει το size με αποτελεσμα να «φευγει» η ριζα (που εχει μολις ανταλλαχθει με το size) , βυθίζει τη νεα ρίζα ώστε να μπει στη σωστη του θεση και επιστρεφει το max.
- Sink(int i)
  Αρχικα για κάθε επαναληψη εντοπιζει το παιδι που εχει τη μεγαλυτερη προτεραιοτητα.
  Επειτα ελεγχει αν ο πατερας εξακολουθει να εχει μεγαλυτερη προτεραιοτητα από το παιδι με την μεγαλυτερη προτεραιοτητα που βρηκαμε πριν, και αν αυτό είναι αληθες, ουσιαστικα δεν κανει τιποτα γιατι ολοι οι κομβοι βρισκονται στη σωστη τους θεση. Αν αυτό δεν είναι αληθες, τοτε κανει swap τον πατερα με το παιδι με τη μεγαλυερη προτεραιοτητα και επαναλαμβανεται η ιδια διαδικασια.

Αλγόριθμος Greedy

Εδώ, αποφασίσαμε να «σπάσουμε τον αλγόριθμο σε μικρές μεθόδους» που κάθε μια υλοποιεί ένα μικρό κομμάτι του συνολικού κώδικα.

Αρχικά, φτιάξαμε έναν static πίνακα Task[] tasks ο οποίος θα περιέχει τα προς εισαγωγή tasks, και μια static int numberp που περιέχει τον αριθμό των processors.

Μέθοδοι

i. ReadTasks(String m) που παίρνει ως όρισμα το όνομα του αρχείου που θα διαβάσει και ουσιαστικά διαβάζει το αρχείο (Υποθέτουμε ότι δεν υπάρχουν συντακτικά λάθη,

εκτός από το αν ο δηλωθέν αριθμός από Tasks (2η γραμμή) δεν συμπίπτει με τα tasks που υπάρχουν στο υπόλοιπο αρχείο). Αυτός ο αλγόριθμος εκτός από το διάβασμα του αρχείου, αρχικοποίησή και τον στατικό πίνακα Tasks[] που έχουμε ορίσει πιο πάνω με αντικείμενα tasks που δημιουργεί κατά την ανάγνωση της 3ης γραμμής του txt και κάτω. Επίσης όταν διαβάζει την πρώτη γραμμή, κάνει και set τον αριθμό των επεξεργαστών.

ii. public static double greedy () που δημιουργεί έναν σωρό heap. Καλεί τον constructor για κάθε processor και ταυτόχρονα τον βάζει στον σωρό. Έπειτα για κάθε task που βρίσκεται στον πίνακα με τα tasks παίρνει τον επεξεργαστή με τη μέγιστη προτεραιότητα και εισάγει στην λίστα του το συγκεκριμένο task. Έπειτα (επειδή ο επεξεργαστής έχει αφαιρεθεί από την σωρό μέσω της getmax()) ξανά εισάγει τον επεξεργαστή στη σωρό στη νέα σωστή του θέση. Έπειτα δημιουργούμε έναν random processor pro (απλά για την αρχικοποίηση του) και τον αφαιρούμε από την heap τον επεξεργαστή που βρίσκεται στην κεφαλή μέσω της getmax έτσι ώστε ο επεξεργαστής που θα μείνει στο τέλος στην μεταβλητή pro να είναι αυτός με το μεγαλύτερο active time και να πάρουμε το makespan του. Επιπλέον, αν ο αριθμός των tasks είναι μικρότερος από 50, εκτυπώνει και τον κάθε επεξεργαστή σε αύξουσα σειρά όπως ζητείται, μέσω της printProcessor. Η μέθοδος αυτή επιστρέφει το makespan σε αυτόν που θα την καλέσει. Το όνομα του αρχείου προς έλεγχο δίνεται από το args[0].

ΜΕΡΟΣ Β

Επειδη οι ΑΜ μας τελειωνουν και οι 2 σε μονο αριθμο, υλοποιήσαμε την QuickSort. Βασιστηκαμε στον κωδικα των διαφανειων (Ενοτητα 9, Διαφανεια 9/16), και απλως αντικαταστησαμε τη μεθοδο less με τη μεθοδο greater γιατι θέλουμε ταξινομηση κατά φθινουσα σειρα. (Προφανως έχουμε αντιληφθεί την υλοποίηση της.) Με λιγα λογια, η quickSort δουλευει με αναδρομες, και κάθε φορα αναδιατάσσουμε τον πίνακα σε 2 υποπίνακες έτσι ώστε ο πρώτος υποπίνακας έχει τα στοιχεία που είναι >= pivot, ο δεύτερος υποπίνακας έχει τα στοιχεία που είναι <= pivot και το pivot μπαίνει στην τελική του (σωστη) θέση. Επειτα καλουμε αναδρομικα ταξινομηση στους 2 υποπινακες.

Σχόλια:

Έχουμε βάλει μια μεθοδο main σε σχολια ετσι ώστε να βεβαιωθουμε ότι ο αλγοριθμος της QuickSort λειτουργει σωστα.

ΜΕΡΟΣ C

Εδώ υλοποιούμε μια μέθοδο public static void writeData() η οποία γραφεί 10 αρχεία για tasks =100, 250, 500 (Σύνολο 30 αρχεία). Με τυχαίους τους χρόνους κάθε task. Έχουμε κάνει τη σύμβαση ότι κάθε διεργασία έχει το πολύ 78 δ/λεπτά χρόνο ώστε να παίρνουμε τιμές από 1 μέχρι 78 σαν χρόνος.

Μέσα στην main δημιουργούμε έναν πίνακα makespanarray 30x2 ο οποίος περιέχει κάθε makespan που βρίσκουμε για κάθε ένα από τα αρχεία και το εισάγει στην 1η στήλη αν πρόκειται για τον πρώτο αλγόριθμο ή στην 2η αν πρόκειται για τον 2ο αλγόριθμο. Επίσης για να βρούμε τον μέσο ορό δημιουργούμε έναν πίνακα averagemakespanarray στον οποίο βάζουμε το sum/10 (γιατί για κάθε ποσότητα από tasks δημιουργούνται 10 αρχεία. Το sum περιέχει το συνολικό makespan και των 10 αρχείων). Έπειτα τυπώνουμε καταλληλά το makespan με τους αντίστοιχους αλγορίθμους. Επιπλέον έχουμε δημιουργήσει έναν φάκελο με όνομα «πειραματικά δεδομένα» μέσα στον φάκελο data στον

οποίο έχουμε βάλει κάποια αρχεία που δημιουργήθηκαν από τον κώδικα μας, και τα οποία παραθέτουμε στο παρακάτω διάγραμμα. Για να ελέγξετε την υλοποίηση, μέσα στο αρχείο του πηγαίου κώδικα δεν υπάρχουν τα συγκεκριμένα αρχεία και όταν κληθεί η comparisons θα δημιουργήσει με τυχαίο τρόπο 30 διαφορετικά αρχεία και θα βρει το average makespan τους, όπως ζητείται από την εκφώνηση.

Για τα δικά μας πειραματικά δεδομένα έχουμε:

The average makespan for the 1st algorithm for N == 100 is : 417.7

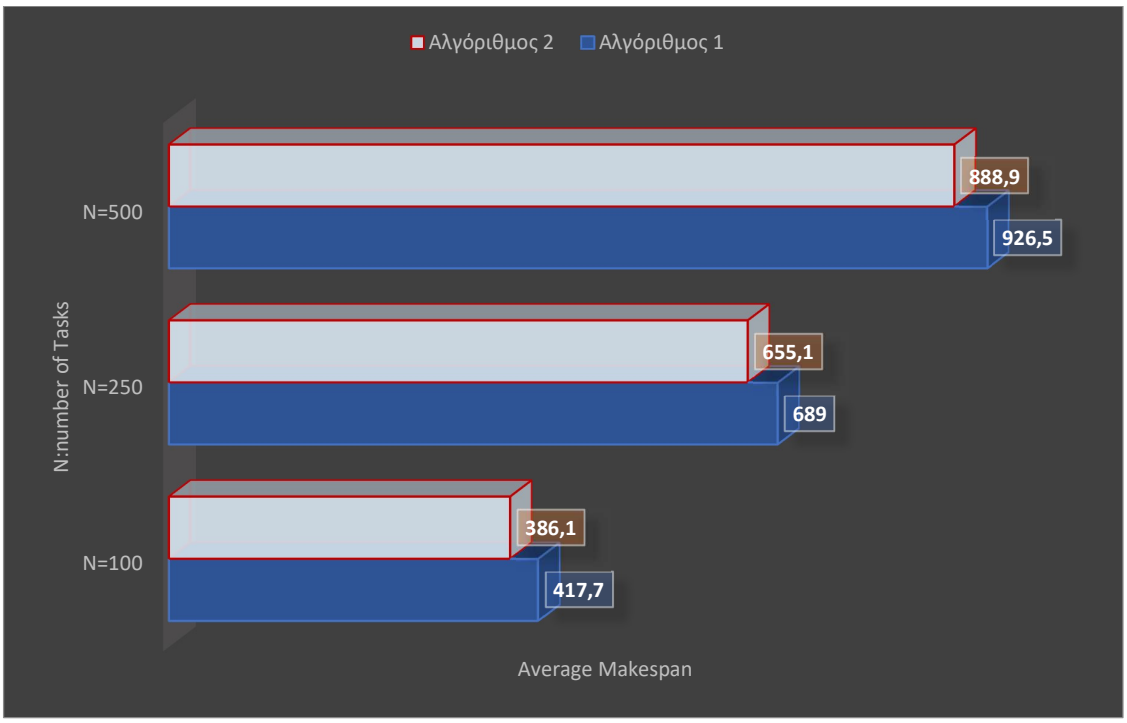
The average makespan for the 2nd algorithm for N == 100 is : 386.1

The average makespan for the 1st algorithm for N == 250 is : 689.0

The average makespan for the 2nd algorithm for N == 250 is : 655.1

The average makespan for the 1st algorithm for N == 500 is : 926.5

The average makespan for the 2nd algorithm for N == 500 is : 888.9



Όπως παρατηρούμε και από το διάγραμμα είναι σαφώς αποδοτικότερος ο 2ος αλγόριθμος εφόσον το average makespan είναι λιγότερο, που σημαίνει ότι οι επεξεργαστές θα χρειαστούν συνολικά λιγότερο χρόνο για την εκπλήρωση όλων των διεργασιών. Επίσης παρατηρούμε ότι όσο αυξάνεται ο αριθμός των διεργασιών, τόσο η διαφορά στο makespan των 2 αλγορίθμων αυξάνεται. (Ενδεικτικά, για N=100 το makespan του αλγορίθμου 2 μειώθηκε κατά 31.6 δ/λεπτά, για N=250 το makespan του αλγορίθμου 2 μειώθηκε κατά 33.9 δ/λεπτά, και για N=500 το makespan του αλγορίθμου 2 μειώθηκε κατά 37.6 δ/λεπτά), που μας οδηγεί στο συμπέρασμα ότι όσο πιο πολλές διεργασίες θα πρέπει να διεκπεραιωθούν τόσο μεγαλύτερη θα είναι η διαφορά στο makespan του 2ου δηλαδή για μεγάλο αριθμό από διεργασίες ο 2ος αλγόριθμος θα γίνεται όλο και αποδοτικότερος.

Καταλήξαμε στο συμπέρασμα ότι ο 2ος αλγόριθμος είναι πιο αποδοτικός επειδή πρώτα ταξινομεί κατά φθίνουσα σειρά τα tasks και υστέρα τα εισάγει στον επεξεργαστή με τη μεγαλύτερη προτεραιότητα. Αυτό πρακτικά σημαίνει ότι η διεργασία με τον μεγαλύτερο κάθε φορά χρόνο θα εκτελείται από τον επεξεργαστή με τον λιγότερο χρόνο εκτελεσμένων διεργασιών. Αυτό είναι προφανώς πιο αποδοτικό από τον αλγόριθμο 1 ο οποίος εισήγαγε στους επεξεργαστές τις διεργασίες με τη σειρά που τις διάβαζε από το txt αρχείο με αποτέλεσμα το συνολικό makespan να αυξάνεται αρκετά λόγω του ότι μπορεί κάποιος επεξεργαστής που θα είχε προτεραιότητα να έπαιρνε ένα αρκετά μεγάλο task ενώ είχε ήδη αλλά, οπότε ο συνολικός του χρόνος αυξάνεται.

ΜΕΡΟΣ D

Ο πηγαίος κώδικας βρίσκεται στο φάκελο src.

Για τον αλγόριθμο greedy, το txt αρχείο που δίνεται ως είσοδος διαβάζεται από command line arguments και ονομάζεται MerosB.txt και βρίσκεται στο φάκελο Data που παραδώσαμε. Όταν κληθεί η Comparisons θα δημιουργήσει 30 καινούρια txt αρχεία και θα τυπώσει το averagemakespan για κάθε αλγόριθμο όπως ζητείται.