Δομές Δεδομένων: Εργασία 2

απο Βασιλης Παπαδημας (3220150) & Μαριος Ματσα (3220120)

Μέρος Α: Influenza_k.java

Μάριος

Μέρος Β: ΑΤΔ ουράς προτεραιότητας

Μάριος

Μέρος Γ: DynamicInfluenza_k_withPQ.java

Στο DyamicInfluenza_k_withPQ.java, αρχικά μετράμε το μέγεθος του αρχείου εισόδου και μετά ζητάμε από τον χρήστη να εισάγει το k. Επειτα ελέγχουμε αν k>N, όπου N ο αριθμός γραμμών του αρχείου εισόδου. Μετά, δημιουργούμε ένα PQ cities με μέγεθος 2k και Comparator της τάξης NegativeComparator (η οποία ουσιατικά ταξινομεί τα στοιχεία με την αντίστροφη σειρά από την κανονική, δηλαδή το min στοιχείο θα είναι το max στοιχείο κ.ο.κ). Τότε, κάνουμε ένα loop που διαβάζει όλες τις γραμμές του αρχείου εισαγωγής και δημιουργεί για κάθε γραμμή ένα αντικείμενο City με τα αντίστοιχα δεδομένα. Εάν η PQ έχει λιγότερα από k στοιχεία τότε το εισάγει σε αυτή. Αλλιώς, το εισάγει μόνο εάν έχει μεγαλύτερη προτεραιότητα από το στοιχείο της PQ με την μικρότερη προτεραιότητα, το οποίο σύμφωνα με τον NegativeComparator θα είναι το PQ.min(). Σε αυτή την περίπτωση επίσης αφαιρεί το min ώστε η PQ να έχει k αντικείμενα.

Αφού τελειώσει το loop, σε ένα άλλο loop εξάγουμε τα k στοιχεία της PQ (κάθε φορά με το getmin()) και εισάγουμε το όνομα τους σε έναν String[] leaderboard με την αντίστροφη σειρά. Ετσι, όταν στο επόμενο και τελευταίο loop εκτυπώνουμε τα περιεχόμενα του leaderboard, τα ονόματα των πόλεων είναι σε αύξουσα σειρά πυκνότητας κρουσμάτων.

Επομένως, για την ανάλυση της πολυπλοκότητας το ουσιαστικό κομμάτι του κώδικα που πρέπει να αναλύσουμε είναι το loop που επεξεργάζεται το input και εισάγει ή μη τα αντικείμενα City στην PQ. Σε αυτό το loop έχουμε N επαναλήψεις. Κάθε επανάληψη έχει σταθερό κόστος το splitting του String με regex, την δημιουργία του αντικειμένου City και την ρύθμιση των πεδιών του. Ολα αυτά ειναι O(1). Για τις πρώτες k επαναλήψεις κάνουμε πάντα insert το αντικείμενο στη PQ, αυτό είναι O(logn). Στις επόμενες N-k, βρίσκουμε το min αντικείμενο (O(1) λόγω της υλοποίησης μας) και εάν έχει μικρότερη προτεραιότητα από το τρέχον city τότε το αφαιρούμε O(logn) και εισάγουμε το τρέχον O(logn). Αρα το συνολικό loop έχει worst-case complexity O(Nlogn). Στην πραγματικότητα, είναι O(N) μετά από την επανάληψη που θα εισαχτεί το στοιχείο που έχει globally την k-οστή υψηλότερη προτεραιότητα. Επομένως, άμα αυτό συμβεί νωρίς (περιπτώσεις όπου το k είναι αρκετά μικρότερο του συνολικού αριθμού πόλεων) τότε η πολυπλοκότητα θα πλησιάζει O(N) (best-case άμα τα πρώτα k στοιχεία του input είναι οι k πόλεις με την μικρότερη πυκνότητα).

Μέρος Δ: Dynamic_Median.java

Στο Dynamic_Median.java, δημιουργούμε δυο PQ (hi και lo) με μέγεθος 500 (ώστε να μην γίνει ποτέ resize). Η lo χρησιμοποιεί τον NegativeComparator και η hi τον PositiveComparator. Μετά κάνουμε ένα loop, διαβάζοντας σε κάθε επανάληψη μια γραμμή του αρχείου εισόδου και δημιουργώντας ένα αντικείμενο City με τα αντίστοιχα στοιχεία. Τότε, αυτό που θέλουμε να κάνουμε είναι να εισάγουμε το αντικείμενο στην hi ή την lo, με στόχο όλες οι k πόλεις που έχουμε διαβάσει στην k-οστή επανάληψη να είναι διαμερισμένες στις hi και lo έτσι ώστε όλες οι πόλεις στην hi να έχουν υψηλότερη προτεραιότητα από όλες τις πόλεις στην lo, και επίσης αν ο k είναι άρτιος να ισχύει size(lo) = size(hi) = k/2 (συνθήκη 1) και αν ο k είναι περιττός να ισχύει size(lo) = size(hi) + 1 = $\lceil k/2 \rceil$ (συνθήκη 2). Ωστε να επιτευχτεί αυτό:

- αν και οι δύο PQ είναι κενές (πρώτη επανάληψη), το εισάγουμε στην lo. Ισχύει η συνθήκη 2.

- αν η hi είναι κενή και η lo περιέχει ένα στοιχείο (δεύτερη επανάληψη), εισάγουμε το τρέχον στην hi άμα έχει ψηλότερη προτεραιότητα από αυτό στην lo, αλλιώς τους αλλάζουμε σειρά. Ισχύει η συνθήκη 1.

- στις υπόλοιπες επαναλήψεις, εισάγουμε το στοιχείο στην lo άμα έχει χαμηλότερη προτεραιότητα από το στοιχείο στην lo με την υψηλότερη προτεραιότητα, αλλιώς το εισάγουμε στην hi. Τότε, άν το k είναι άρτιος ((hi.size() + lo.size()) % 2 == 0), στην προηγούμενη επανάληψη ήταν περιττός, άρα αρχικά ίσχυε η συνθήκη 2, και η συνθήκη 1 δεν θα ισχύει μόνο άμα προσθέσαμε στο hi (δηλαδή lo.size() == hi.size() + 2, οπότε αφαιρούμε από την lo το στοιχείο με την υψηλότερη προτεραιότητα (λόγω NegativeComparator) κα το προσθέτουμε στην hi για να έχουν ίδιο μέγεθος. Αλλιώς, άμα το k είναι περιττός τότε στην προηγούμενη επανάληψη ήταν άρτιος άρα ίσχυε η συνθήκη 1, και η συνθήκη 2 δεν θα ισχύει τώρα μόνο εάν προσθέσαμε στο hi (lo.size() < hi.size()) άρα αφαιρούμε το στοιχείο του hi με την χαμηλότερη προτεραιότητα και το προσθέτουμε στην lo ώστε να ισχύει η συνθήκη 2.

Πρακτικά λοιπόν, κάθε επανάληψη έχει κόστος O(logn) και συνολικά το πρόγραμμα μας έχει πολυπλοκότητα O(Nlogn), αφού ό ίδιος ο υπολογισμός του mean είναι (O(1)): όταν θέλουμε να υπολογίσουμε το mean, στο τέλος οποιασδήποτε επανάληψης, ξέρουμε ότι θα είναι το lo.min().