

ΜΕΡΟΣ Ε΄

******(Αν θέλετε να δείτε πρώτα πως δίνεται το μονοπάτι ανατρέξτε στο μέρος d του report στην τελευταία σελίδα)

α) Το μέρος Α΄ υλοποιήθηκε με την χρήση του **heapsort**.

Αρχικά διαβάζουμε τις γραμμές του αρχείου, έστω η γραμμές και αρχικοποιούμε έναν πίνακα `cities` τύπου `City` $n + 1$ θέσεων. Θέτουμε το στοιχείο `cities[0] = null` ώστε να μπορέσουμε να χειριστούμε τον πίνακα σαν ένα πλήρες δυαδικό δέντρο.

Έπειτα δημιουργούμε την κλάση `HearSort` όπου και γίνεται η ταξινόμηση του πίνακα `cities`. Στην κλάση `HearSort` έχουμε τις εξής μεθόδους:

- **`void swap (City[] cities, int i, int j)`**
Η μέθοδος όταν κληθεί αντιστρέφει τις πόλεις των θέσεων i και j του πίνακα `cities[]`.
- **`void sink (City[] cities, int parent, int len)`**
Η μέθοδος αυτή έχει ως ορίσματα τον πίνακα των πόλεων, τον κόμβο από τον οποίο ξεκινάμε την διαδικασία της ταξινόμησης σε κάθε κλήση της `sink()` και την τελευταία θέση που θέλουμε να εξετάσουμε του πίνακα των πόλεων. Η μέθοδος ξεκινάει από τον κόμβο `parent` και συγκρίνει τα 2 παιδιά του και αν κάποιο παιδί έχει μεγαλύτερη πυκνότητα κρουσμάτων ανά 50000 κατοίκους από τον γονέα, τότε γίνεται μία εναλλαγή μεταξύ των πόλεων μέσω της μεθόδου `swar()`. Αυτό γίνεται για κάθε γονέα που έχει παιδί από την θέση `parent` του πίνακα μέχρι το τέλος.
- **`void heapsort (City cities[])`**
Η μέθοδος δέχεται ως όρισμα τον πίνακα `cities` που πρέπει να ταξινομηθεί και τον ταξινομεί ως εξής:
Αρχικά μετατρέπουμε τον πίνακα `cities[]` σε σωρό, δηλαδή στη ρίζα έχουμε την πόλη με την μεγαλύτερη πυκνότητα κρουσμάτων ανά 50000 κατοίκους και σε κάθε κόμβο η πυκνότητα είναι μεγαλύτερη ή ίση και από αυτήν των παιδιών του. **$O(N)$**
Έπειτα κάνουμε την ταξινόμηση, μέσα σε μία `while` με N επαναλήψεις (πλήθος πόλεων στον πίνακα). Στην 1^η επανάληψη έχουμε την πόλη με τον μικρότερο $A.K/50000$ στην 1^η θέση του πίνακα, κάνουμε μία αντιστροφή με το τελευταίο στοιχείο του πίνακα και κάνουμε `sink()` μέχρι την προ-τελευταία θέση του πίνακα για να βρεθεί στην κορυφή η επόμενη πόλη με τον μικρότερο $A.K/50000$. Στην 2^η επανάληψη βάζουμε την πόλη στην κορυφή στην προ – τελευταία θέση του πίνακα και κάνουμε `sink` μέχρι την θέση $N-2$. Μέχρι τώρα έχουμε βάλει τις 2 πόλεις με τον μικρότερο $A.K/50000$ στις 2 τελευταίες θέσεις με φθίνουσα σειρά. Επαναλαμβάνουμε μέχρι να μη μπορεί να γίνει άλλη εναλλαγή (δηλαδή $N=1$) και έτσι έχουμε καταφέρει να ταξινομήσουμε τον πίνακα σε φθίνουσα σειρά. **$O(N\log N)$**

πολυπλοκότητα heapsort $O(N\log N)$

b)

- Υλοποίηση της μεθόδου **remove(int id)** της ουράς προτεραιότητας PQ του μέρους Β':
Οι δυνατές τιμές για το ID κάθε πόλης ανήκουν στο διάστημα [1,999] με κάθε πόλη να έχει μοναδικό ID. Στην κλάση PQ αρχικοποιούμε τον πίνακα IDposition[] τύπου int, 1000 θέσεων. Στην ουσία, θα χρησιμοποιήσουμε τις θέσεις από το 1 έως το 999 που είναι και το διάστημα τιμών του ID.
Κάθε φορά που εισάγουμε κάποια πόλη στην ουρά προτεραιότητας με την μέθοδο insert(), έχουμε και την εντολή IDposition[city.getID()] = size, όπου city το αντικείμενο τύπου City που εισάγεται στην ουρά και size το τρέχων μέγεθος του πίνακα (ουράς) όπου και θα μπει η city. Έτσι έχουμε πρόσβαση σε χρόνο $O(1)$ για να βρούμε σε ποια θέση του πίνακα βρίσκεται μία πόλη με κάποιο ID.
Σαφώς, με την κλήση της swap(int i, int j) θα πρέπει να ενημερωθεί και ο πίνακας των IDposition και έχουμε ορίσει τις κατάλληλες εντολές. Οπότε σε κάθε νέα εισαγωγή πόλης, εάν υπάρξει κάποια αλλαγή σε θέσεις του πίνακα, αλλάζει και ο πίνακας IDposition. Επίσης, σε κάθε διαγραφή γίνεται το ίδιο. Πριν καλέσουμε την sink(), θα πρέπει πρώτα να ενημερώσουμε τον πίνακα IDposition, δηλαδή το IDposition του ID της πόλης που μπήκε στη θέση της πόλης που διαγράψαμε, θα πρέπει να γίνει ίσο με την θέση του πίνακα heap που μόλις εισάχθηκε. (Ίσως όχι τόσο καλή επεξήγηση, αλλά φαίνεται στις remove και getmax τι εννοούμε). Φροντίζουμε στην getmax() και remove() να θέσουμε την IDposition[city.getID()] της πόλης που αφαιρέθηκε ίσο με 0. Τέλος, όλες οι εντολές αυτές είναι προφανές ότι δεν επηρεάζουν την πολυπλοκότητα των άλλων μεθόδων καθώς είναι σε $O(1)$. Τώρα στην remove, έχουμε σε χρόνο $O(1)$ τη θέση που βρίσκεται η πόλη με ένα συγκεκριμένο id. Συντάσσουμε την remove(int id) όπως και την getmax(), με τη διαφορά ότι στη remove(int id) δεν αφαιρούμε το στοιχείο στην θέση 1 του πίνακα, αλλά το στοιχείο στη θέση IDposition[id]. Έτσι καταφέραμε να υλοποιήσουμε τη μέθοδο σε **$O(\log N)$** .
- Αναφορικά με την υλοποίηση της **DynamicCovid_k_withPQ** του μέρους Γ :
Καταρχάς, πριν αναφερθούμε αναλυτικά στην ιδέα υλοποίησης του μέρους Γ, θα πρέπει να σταθούμε λίγο στον τρόπο που έχουμε υλοποιήσει τον comparator στην ουρά μας (PQ). Έχουμε την **κλάση Comparator**, και αρχικοποιούμε την public μεταβλητή priority = MIN (με χρήση enum). Οι δυνατές τιμές για την τιμή priority είναι MIN/MAX. Όταν κατασκευάζουμε το αντικείμενο comparator τύπου Comparator **μέσα στην PQ** προεπιλεγμένη τιμή για την priority είναι η MIN. Στην κλάση comparator έχουμε τη μέθοδο compareTo η οποία δέχεται ως ορίσματα 2 πόλεις. Εάν η μεταβλητή priority είναι MIN τότε η σύγκριση των πόλεων γίνεται με βάση τον μικρότερο αριθμό κρουσμάτων ανά 50000 κατοίκους, ενώ αν είναι MAX η σύγκριση γίνεται με βάση τον μεγαλύτερο. Όταν, λοιπόν, δημιουργήσουμε μία ουρά προτεραιότητας PQ η σύγκριση θα γίνεται με βάση το ελάχιστο στοιχείο

(δηλαδή τον μικρότερο αριθμό κρουσμάτων ανά 50000 κατοίκους στην εργασία μας) και κάθε φορά στην ρίζα της ουράς θα βρίσκεται το στοιχείο με τον μικρότερο αριθμό κρουσμάτων/50000. Έχουμε προσθέσει την μέθοδο `minToMax()` στην PQ η οποία αλλάζει την τιμή του `comparator.priority` σε MAX έτσι ώστε οι συγκρίσεις και το κλειδί με την μεγαλύτερη προτεραιότητα να είναι αυτό με τον μεγαλύτερο αριθμό κρουσμάτων/50000. Πέρα από την αλλαγή αυτή πρέπει με κάποιες `sink()` να μετατρέψουμε την ουρά σε σωρό που να έχει ως ρίζα την πόλη με τον μεγαλύτερο αριθμό κρουσμάτων/50000. **$O(N)$** .

Ας συζητήσουμε τώρα για το πως υλοποιήσαμε την

DynamicCovid_k_withPQ. Ζητάμε από τον χρήστη να μας δώσει έναν ακέραιο k , όπου k οι πόλεις με τον μεγαλύτερο αρ. κρ. /50000 που θα τυπωθούν στην κονσόλα. Δημιουργούμε μία ουρά προτεραιότητας `pq` και διαβάζουμε το αρχείο εισόδου γραμμή προς γραμμή. Όσο το μέγεθος της `pq` (`pq.size()`) είναι μικρότερο του k απλώς κάνουμε `insert()` την πόλη στην ουρά. Εξασφαλίζουμε έτσι ότι τα ενεργά στοιχεία της ουράς είναι μικρότερα ή ίσα του k . Εάν το μέγεθος της ουράς είναι ίσο με το k , ελέγχουμε το `pq.max()` - που είναι η πόλη με τον μικρότερο αρ.κρ./50000 ανάμεσα στις πόλεις της ουράς- με την πόλη που διαβάζουμε την επανάληψη που βρισκόμαστε. Εάν ο αρ.κρ./50000 της πόλης που διαβάζουμε τώρα είναι μικρότερος από αυτόν της `pq.max()` τότε απλά προχωράμε στην επόμενη επανάληψη. Στην περίπτωση που η πόλη που διαβάζουμε από το αρχείο έχει μεγαλύτερο αρ.κρ./50000 από της `pq.max()` τότε διαγράφουμε το `pq.max()` και εισάγουμε στην ουρά μας τη νέα πόλη που μόλις διαβάσαμε. Έτσι λοιπόν έχουμε εξασφαλίσει πλήρως, ότι σε καμία περίπτωση το μέγεθος της ουράς (ενεργά στοιχεία) δε θα ξεπερνά το k . Περιληπτικά, έχουμε συνεχώς μία ουρά που έχει τις πόλεις (το πολύ k) με τον μεγαλύτερο αρ.κρ./50000 και στην ρίζα βρίσκεται η πόλη με τον μικρότερο αριθμό εξ αυτών. Μία πόλη θα πρέπει να μπει στην ουρά και να βγει η ρίζα αν και μόνο αν ο αρ.κρ. της νέας πόλης είναι μεγαλύτερος από τον αρ. κρ της ρίζας. Τέλος όσο αφορά την έξοδο, αν επιχειρήσουμε να τυπώσουμε με μία επανάληψη τις k μεγαλύτερες πόλεις της `pq` θα εμφανιστούν σε φθίνουσα σειρά. Κάνουμε `pq.minToMax()` που περιγράψαμε παραπάνω, και αν τα τυπώσουμε σε μία επανάληψη θα τυπωθούν σε αύξουσα σειρά.

- **Πολυπλοκότητα μέρους Γ και σύγκριση με πολυπλοκότητα μέρους Α.**

Έχουμε μια `while` με N επαναλήψεις $O(\log N)$, όπου N ο συνολικός αριθμός των πόλεων του αρχείου. Και k οι πόλεις που θα τυπωθούν στο τέλος. Αναφορικά με την εισαγωγή και γενικά την επεξεργασία της ουράς, η πολυπλοκότητα θα είναι $O(\log k)$, καθώς θα βρίσκονται το πολύ k ενεργά αντικείμενα στην ουρά. Έχουμε λοιπόν **$O(N \log k)$** . Αναφορικά με την έξοδο και την εμφάνιση των k πόλεων, έχουμε:

$O(N)$ για την `pq.minToMax()` και **$O(k \log k)$** για την εμφάνιση/διαγραφή των πόλεων.

Συνολική πολυπλοκότητα = $O(N \log k) + O(N) + O(k \log k) = O(N \log k)$.

Αναφορικά με την πολυπλοκότητα του μέρους Α στην Covid_k έχουμε $O(N \log N)$ λόγω ταξινόμησης. Επομένως έχουμε καλύτερη πολυπλοκότητα σε σύγκριση με το μέρος Α ειδικά για περιπτώσεις όπου το k είναι πολύ μικρό.

c) Επεξήγηση υλοποίησης και πολυπλοκότητας [Dynamic_Median](#).

Αρχικοποιούμε 2 ουρές προτεραιότητας maxHeap και minHeap. Καθώς η τιμή του comparator.priority είναι ίση με MIN και στις 2 ουρές η πόλη με την μέγιστη προτεραιότητα είναι εκείνη με τον χαμηλότερο αρ.κρ/50000. Επομένως με την εντολή maxheap.minToMax() η μέγιστη προτεραιότητα της πόλη στην maxHeap να είναι εκείνη με τον μεγαλύτερο αρ. κρ./50000. Και πάλι, διαβάζουμε γραμμή προς γραμμή το αρχείο εισόδου. Η γενική ιδέα είναι να έχουμε στην maxHeap τις k/2 πόλεις με τον μικρότερο αρ.κρ/50000 και στην minHeap τις k/2 πόλεις με τον μεγαλύτερο (Θα είναι στο περίπου ίσο k/2 γιατί αν έχουμε συνολικό αριθμό πόλεων περιπτώ σε μία επανάληψη, μία ουρά θα έχει ένα στοιχείο περισσότερο).


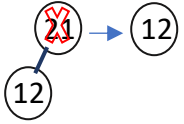


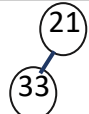
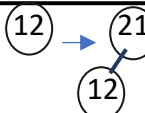
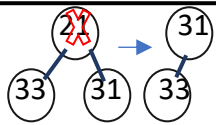
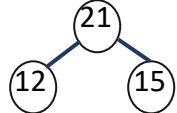
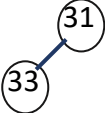
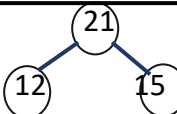
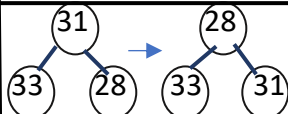
Αρχικά, εάν η maxheap είναι κενή ή το maxHeap().max() έχει μεγαλύτερο αρ.κρ./50000 από την πόλη που πρέπει να εισαχθεί, τότε η νέα πόλη θα εισαχθεί στην maxHeap. Διαφορετικά στην minHeap. Μετά από κάθε εισαγωγή πρέπει να επιβεβαιώσουμε ότι το πλήθος των πόλεων σε κάθε ουρά είναι ίσο ή κάποια ουρά έχει το πολύ μία παραπάνω πόλη από ότι η άλλη. Αν δεν συμβαίνει αυτό, τότε παίρνουμε από την ουρά με τις -κατά 2- περισσότερες πόλεις, την ρίζα της και την τοποθετούμε στην άλλη ουρά και είμαστε εντάξει όσο αφορά τους περιορισμούς που έχουμε θέσει. Στο τέλος κάθε επανάληψης έχουμε τον median για την επανάληψη εκείνη. Εάν οι ουρές έχουν το ίδιο πλήθος πόλεων ο median είναι η πόλη που βρίσκεται είτε στην ρίζα της maxHeap είτε στη ρίζα της minheap. Συγκρίνουμε τον αριθμό κρ./50000 (με την μέθοδο getDensity() της City – μιας και δεν την αναφέραμε καθόλου-) και ο median είναι η πόλη με τον μεγαλύτερο density. Εάν μία ουρά έχει 1 στοιχείο παραπάνω από την άλλη, τότε ο median είναι η πόλη που βρίσκεται στη ρίζα της ουράς με τις περισσότερες πόλεις.

Πολυπλοκότητα προγράμματος: $O(N \log k)$

Πολυπλοκότητα median: $O(1)$, καθώς με το πολύ 3 ελέγχους βρίσκουμε σε ποια ουρά είναι ο median και αφού θα βρίσκεται στην ρίζα με την μέθοδο max() της pq παίρνουμε σε $O(1)$ την πόλη αυτή.

Θα δώσουμε ένα παράδειγμα, με χρήση ακεραίων, αντί για πόλεις για ευκολία. Έστω ότι διαβάζουμε με τη σειρά τους αριθμούς: 21,12,33,31,15,28. Προφανώς ο median είναι ο 28. Ας το δείξουμε με την παραπάνω υλοποίηση.

Δείτε παρακάτω (δε μπορούσαμε να κάνουμε κάτι καλύτερο για το σχήμα)

Loop	Integer	maxHeap	minHeap	Median
1	21			21
2	12			21
3	33			21
4	31			31
5	15			21
6	28			28

d) Όλα τα αρχεία μεταγλωττίζονται από την γραμμή εντολών. Όταν τρέξετε κάθε main θα σας ζητηθεί να πληκτρολογήσετε το αρχείο για ανάγνωση.

Έστω το αρχείο Europe_Covid_Info.txt στην επιφάνεια εργασίας.

Ο τρόπος για να διαβαστεί το αρχείο είναι να γράψετε το ακόλουθο μονοπάτι:

C:/Users/thanos/Desktop/Europe_Covid_Info.txt

Εάν το όνομα κάποιου φακέλου στο μονοπάτι που θα πρέπει να διαβαστεί περιλαμβάνει ελληνικούς χαρακτήρες, δε θα μπορεί να διαβαστεί.