1η Γραπτή Εργασία Αλγορίθμων

Ιωάννης Δάρας
03115018
(daras.giannhs@gmail.com)

"In algorithms, as in life, persistence usually pays off."
— Steven S. Skiena, The Algorithm Design Manual

Άσκηση 1

(α)

1. $n^2 = O(n^2)$
2. $2^{(log_2 n)^4} = O\left(2^{(log_2 n)^4}\right)$
3. $\frac{logn!}{(logn)^3} = O\left(\frac{n}{(logn)^2}\right)$

   Από τον τύπο του Stirling (βλ. ερώτημα 5.) έχουμε:

   $$logn! = \Theta(nlogn)$$

   Άρα:

   $$\frac{logn!}{(logn)^3} = \frac{\Theta(nlogn)}{(logn)^3} = \frac{n}{(logn)^2}$$

4. $n \cdot 2^{2^{2^{100}}} = O(n)$
5. $log\left(\binom{n}{logn}\right) = O\left(lognloglogn\right)$

   Από τον τύπο του Stirling έχουμε ότι:

   $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \iff$$

   $$logn! = \frac{1}{2}log2\pi n + nlogn - nloge \Rightarrow$$

   $$\boxed{\log n! = n\log n - n + \frac{1}{2}\log 2\pi n + O\left(\frac{1}{n}\right)}$$

   Αλλά:

   $$\log\binom{n}{\log n} = \log n! - \log((n - \log n)!) - \log((\log n)!)$$

   Με αντικατάσταση από τη σχέση στο κουτάκι προκύπτει ότι:

   $$\log\binom{n}{\log n} = n\log n - (n - \log n)\log(n - \log n) + O(\log n \log\log n)$$

   $$= n\log n - (n - \log n)\left(\log n + \log\left(1 - \frac{\log n}{n}\right)\right) + O(\log n \log\log n)$$

$$= \log^2 n + (n - \log n)\left(\frac{\log n}{n} + O\left(\frac{\log^2 n}{n^2}\right)\right) + O(\log n \log\log n)$$

$$= \log^2 n + O(\log n \log\log n)$$

Άρα:

$$\log\binom{n}{\log n} = (log^2 + O(lognloglogn)) = O(lognloglogn)$$

6. $\frac{(logn)^2}{loglogn} = O\left(\frac{(logn)^2}{loglogn}\right)$
7. $log^4 n = O(log^4 n)$
8. $\sqrt{n!} = O(n^n)$

$$\sqrt{n!} = c(n) \iff n! = c^2(n) \iff log_2 n! = 2log_2 c(n) \iff$$
$$\Theta(nlogn) = log_2 c(n) \iff c(n) = \Theta(n^n) = O(n^n)$$

9. $\binom{n}{6} = O(n^6)$
10. $\frac{n^3}{(logn)^8} = O\left(\frac{n^3}{(logn)^8}\right)$
11. $(log_2 n)^{log_2 n} = O\left((log_2 n)^{log_2 n}\right)$
12. $log\left(\binom{2n}{n}\right) = O(n)$

Θα χρησιμοποιήσουμε τον τύπο του Stirling:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Έτσι έχουμε:

$$\binom{2n}{n} = \frac{(2n)!}{2\cdot(n!)} = \Theta\left(\frac{2\sqrt{\pi n}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)^2}\right) = \Theta\left(\frac{4^n}{\sqrt{n}}\right)$$

Άρα:

$$log_2\left(\binom{2n}{n}\right) = \Theta(2n - log_2\sqrt{n}) = O(n)$$

13. $n\sum_{k=0}^{n}\binom{n}{k} = \Theta(n2^n) = O(n2^n)$
14. $\sqrt{n}^{log_2 log_2(n!)} = O(n^{logn})$

Στο 5. είδαμε ότι: $\log n! = n\log n - n + \frac{1}{2}\log 2\pi n + O\left(\frac{1}{n}\right) = O(nlogn)$ Άρα:

$$\sqrt{n}^{log_2 log_2(n!)} = O(\sqrt{n^{log(nlogn)}}) = O(n^{0.5log(nlogn)}) = O(n^{0.5logn}) = O(n^{logn})$$

15. $\sum_{k=1}^{n} k2^k = O\left(n2^n\right)$

Από ορισμό αθροίσματος γεωμετρικής προόδου έχουμε:

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \Rightarrow \left(\sum_{k=0}^{n} x^k\right)' = \frac{(n+1)x^n(x-1)-(x^{n+1}-1)}{(x-1)^2} \iff$$

$$\sum_{k=0}^{n} kx^k = \frac{(n+1)x^{n+1}(x-1)-x(x^{n+1}-1)}{(x-1)^2} \overset{x=2}{\Rightarrow}$$

$$\sum_{k=0}^{n} kx^k = (n+1)2^{n+1} - 2(2^{n+1}-1) = O(n2^{n+1}) \iff$$

$$\sum_{k=1}^{n} kx^k = O(n2^n)$$

16. $\sum_{k=1}^{n} k2^{-k} = O(1)$

$$\sum_{k=1}^{n} kx^k \leq \sum_{k=1}^{\infty} kx^k \overset{|x|<1}{=} \frac{x}{(1-x)^2} \overset{x=\frac{1}{2}}{\Rightarrow}$$

$$\sum_{k=1}^{n} k\frac{1}{2}^k \leq \frac{\frac{1}{2}}{(\frac{1}{2})^2} = O(1)$$

Από τα παραπάνω, προκύπτει ότι η διάταξη των συναρτήσεων σε αύξουσα σειρά πολυπλοκότητας Ο είναι η ακόλουθη:

$$\sum_{k=1}^{n} k2^{-k} \subset log\left(\binom{n}{logn}\right) \subset \frac{(logn)^2}{loglogn} \subset \frac{logn!}{(logn)^3} \subset log^4 n \quad (\text{υπογραμμικές})$$

$$\subset log\left(\binom{2n}{n}\right) \subset n \cdot 2^{2^{2^{100}}} \subset n^2 \subset \frac{n^3}{(logn)^8} \subset \binom{n}{6} = O(n^6) \quad (\text{πολυωνυμικές})$$

$$\subset (log_2 n)^{log_2 n} \quad (\text{σχεδόν πολυωνυμικές})$$

$$\subset 2^{(log_2 n)^4} \subset \sqrt{n}^{log_2 log_2 (n!)} \subset n\sum_{k=0}^{n}\binom{n}{k} \subset \sum_{k=1}^{n} k2^k \subset \sqrt{n!} \quad (\text{εκθετικές})$$

(β)

1.

$$T(n) = 2T(\frac{n}{3}) + nlogn$$

Θα δουλέψουμε με το Master Theorem.

$$a = 2, \quad b = 3, \quad n^{log_b a} = n^{log_3 2} = n^{0.63} \Rightarrow f(n) = \Omega(n^{log_b a})$$

$$T(n) = \Theta(f(n)) = \Theta(nlogn)$$

2.

$$T(n) = 3T(\frac{n}{3}) + nlogn$$

Θα δουλέψουμε με το Master Theorem.

$$a = 3 \quad b = 3, \quad n^{log_b a} = n \Rightarrow f(n) = \Theta(n^{log_b a} log^k n) \text{ (με k=1)}$$

$$T(n) = \Theta(n^{log_b a} log^{k+1} n) = \Theta(nlog^2 n)$$

3.

$$T(n) = 4T(\frac{n}{3}) + nlogn$$

Θα δουλέψουμε με το Master Theorem.

$$a = 4, \quad b = 3, \quad n^{log_b a} = n^{1.26} \Rightarrow f(n) = O(n^{log_b a})$$

$$T(n) = \Theta(n^{1.26})$$

4.

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + n$$

Έχουμε ότι

$$2T(\frac{n}{2}) + n \leq T(n) \leq 2T(\frac{n}{3}) + n$$

Από Master Theorem στις συναρτήσεις που ορίζουν κάτω και άνω φράγμα αντίστοιχα παίρνουμε ότι:

$$\Theta(n) \leq T(n) \leq \Theta(nlog_2 n)$$

Από την παραπάνω ανάλυση υποψιαζόμαστε τη μορφή της πολυπλοκότητας. Θέτουμε:

$$\lim_{n\to\infty} \frac{T(n)}{n} = c, \quad c \in \{\Re, \infty\}$$

$$\frac{T(n)}{n} = \frac{T(n)}{\frac{n}{2}} + \frac{T(n)}{\frac{n}{3}} + 1$$

$$c = \frac{c}{2} + \frac{c}{3} + 1 \Rightarrow c = 6 \iff$$

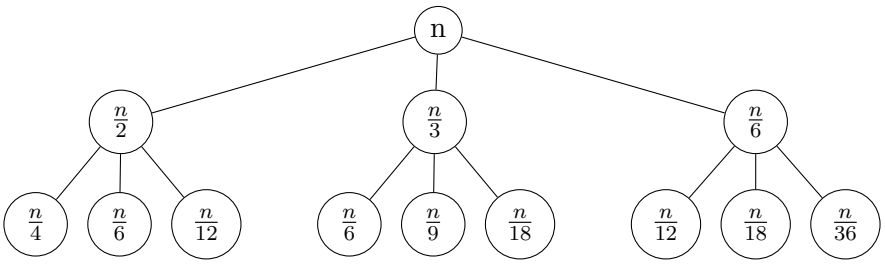
$$T(n) = \Theta(n)$$

5.

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + T(\frac{n}{6}) + n$$

Σχεδιάζουμε το δέντρο της αναδρομής.

Μια εικόνα του για τα δύο πρώτα επίπεδα φαίνεται παρακάτω.



Παρατηρούμε ότι σε κάθε επίπεδο το άθροισμα των πράξεων που γίνεται είναι n. Έχουμε ότι:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + T(\frac{n}{6}) + n \geq 3T(\frac{n}{2}) + n$$

Άρα ο συνολικός αριθμός των επιπέδων είναι μεγαλύτερος $log_2 n$. Άρα, αφού έχουμε περίπου $log_2 n$ επίπεδα και σε κάθε επίπεδο γίνονται n πράξεις, συνολικά για την πολυπλοκότητα του T(n) έχουμε ότι:

$$T(n) = \Theta(nlog_2 n)$$

6.

$$T(n) = T(n^{\frac{5}{6}}) + \Theta(logn)$$

Θέτω $n = 2^m$.

$$(m) = T(2^{\frac{5m}{6}}) + \Theta(log2^m) = T(2^{\frac{5m}{6}} + \Theta(m)$$

Θέτω $S(m) = T(2^m)$ Άρα, έχουμε:

$$S(m) = S(\frac{5m}{6}) + \Theta(m)$$

Έχουμε

$$m^{log_b^a} = m^{log_{\frac{6}{5}}1} = 1 \Rightarrow S(m) = \Theta(m)$$

Αλλά:

$$S(m) = T(2^m) \Rightarrow T(n) = \Theta(log_2 n)$$

7.

$$T(n) = T(\frac{n}{4}) + \sqrt{(}n)$$

Θα δουλέψουμε με το Master Theorem.

$$a = 1, \quad b = 4, \quad n^{log_b a} = n^0 = 1 \Rightarrow f(n) = \Omega(n^{log_b a})$$

$$T(n) = \Theta(f(n)) = \Theta(\sqrt{n})$$

Άσκηση 2

(α)

1.

Ορισμός προβλήματος:

Θέλουμε να χωρίσουμε έναν πίνακα A n αριθμών σε k buckets $(b_1, b_2, ..., b_k)$ ώστε να είναι εξωτερικά ταξινομημένος δηλαδή:

$$\forall i, j \in indexes\{A\} : \quad i \leq j \wedge \forall a_1, a_2 : a_1 \in b_i, a_2 \in b_j, \quad a_1 < a_2$$

Διατύπωση αλγορίθμου:

Για να διατυπώσουμε τον αλγόριθμο μας, θα χρησιμοποιήσουμε τον αλγόριθμο quickselect. Ο αλγόριθμος quickselect είναι ένας αλγόριθμος επιλογής που μας δίνει το k-οστό μικρότερο στοιχείο σε μια μη ταξινομημένη λίστα. Συγκεκριμένα, ο αλγόριθμος quickselect με είσοδο μια μη ταξινομημένη λίστα μήκους n μας επιστρέφει σε χρόνο μέσης περίπτωσης $O(n)$ μια νέα λίστα με στοιχεία τα στοιχεία της προήγουμενης όπου όλα τα στοιχεία πριν το index k (με αρίθμηση που ξεκινά από το 1) είναι μικρότερα από το στοιχείο στο index k.

Παρατηρούμε ότι κάθε φορά που τρέχει η quickselect μας χωρίζει τον πίνακα σε 2 υποπίνακες όπου ο πρώτος είναι εξωτερικά ταξινομημένος σε σχέση με τον 2ο. Παρατηρούμε ακόμη ότι η σχέση της εξωτερικής ταξινόμησης πινάκων είναι μεταβατική. Δηλαδή αν ο πίνακας A_i είναι εξωτερικά ταξινομημένος ως προς τον πίνακα A_j και ο πίνακας A_j είναι εξωτερικά ταξινομημένος ως προς τον πίνακα A_k τότε ο πίνακας A_i είναι εξωτερικά ταξινομημένος ως προς τον πίνακα A_k. Ενώνοντας αυτές τις δύο ιδέες φτάνουμε στον αλγόριθμο μας για την διαμέριση του πίνακα μας σε k υποπίνακες που ικανοποιούν το κριτήριο της εξωτερικής ταξινόμησης: Φτιάχνουμε μια αναδρομική συνάρτηση που παίρνει ως όρισμα έναν πίνακα A n στοιχειων, τρέχει την quickselect για να βρεί το $\frac{n}{2}$ μεγαλύτερο στοιχείο και μετά καλεί τον εαυτό της δύο φορές, μία με το πρώτο μισό του πίνακα και μία με το δεύτερο. Όταν βρισκόμαστε σε βάθος i στις αναδρομικές κλήσεις της συνάρτησης αυτής ο πίνακας θα έχει χωριστεί 2^i μέρη που θα ικανοποιούν την ιδιότητα της εξωτερικής ταξινόμησης. Εμείς θέλουμε να χωρίσουμε τον πίνακα σε K μέρη άρα συνολικά θα φτάσουμε σε βάθος $O(log_2 K)$ στην αναδρομή. Σε κάθε επίπεδο της αναδρομής κάνουμε quickselect για όλα τα στοιχεία του πίνακα, δηλαδή κάνουμε (n) πράξεις. Συνεπώς, η συνολική πολυπλοκότητα του αλγορίθμου μας θα είναι: $O(nlog_2 K)$.

Όσα περιγράψαμε, εκφράζονται στον ακόλουθο ψευδοκώδικα:

Algorithm 1: External Sort pseudocode

Data: $A[1, 2, ..., n]$

Result: $A = [[1, .., \overset{A_1}{\frac{n}{k}}], [\frac{n}{k} + 1, ..., \overset{A_2}{2\frac{n}{k}}], ..., [(k-1)\frac{n}{k} + 1, ..., \overset{A_k}{n}]]$

$a \leq b \iff (a \in A_i) \wedge (b \in A_j) \wedge (i \leq j)$

Function `external_sort(`A, $index$`):`

- **if** $index == log_2 k$ **then** return A
- size = size(A)
- external_sort($A[1 : size/2], index + 1$)
- external_sort($A[size/2 + 1, size], index + 1$)
- **return**

Απόδειξη κάτω φράγματος:

Ο καλύτερος συγκριτικός αλγόριθμος για ταξινόμηση n αριθμών χρειάζεται O(nlogn) πράξεις. Από αυτές τις πράξεις μπορούμε να αφαιρέσουμε τις πράξεις ταξινόμησης των αριθμών μέσα στους k υποπίνακες καθώς αυτοί δεν χρειάζεται να είναι ταξινομημένοι. Κάθε υποπίνακας έχει $\frac{n}{k}$ στοιχεία άρα μπορούμε να γλυτώσουμε $\frac{n}{k} log_2 \frac{n}{k}$ πράξεις, k φορές όσοι δηλαδή είναι οι υποπίνακες.

Άρα, η καλύτερη πολυπλοκότητα που μπορούμε να πετύχουμε είναι:

$$\Omega(nlog_2 n - k\frac{n}{k} log_2 \frac{n}{k}) = \Omega(nlog_2 n - nlog_2 \frac{n}{k} = \Omega(nlog_2 k))$$

2.

Διατύπωση αλγορίθμου:

Για να ταξινομήσουμε έναν πίνακα που είναι εξωτερικά ταξινομημένος αρκεί να ταξινομήσουμε όλα τα υπομέρη του. Κάθε υπομέρος έχει $\frac{n}{k}$ στοιχεία και συνεπώς η ταξινόμηση ενός υπομέρους έχει πολυπλοκότητα: $O(\frac{n}{k} log \frac{n}{k})$. Υπάρχουν k υπομέρη άρα η συνολική πολυπλοκότητα είναι: $O(nlog \frac{n}{k})$.

Απόδειξη κάτω φράγματος:

Το κάτω φράγμα προκύπτει από το γεγονός ότι αφού έχει ταξινομηθεί ο πίνακας εξωτερικά, η ταξινόμηση των διαφορετικών υποπινάκων είναι ανεξάρτητα γεγονότα. Ο κάθε υποπίνακας όμως έχει $\frac{n}{k}$ στοιχεία και συνεπώς για την ταξινόμηση κάθε υποπίνακα έχουμε πολυπλοκότητα $\Omega \frac{n}{k} log(\frac{n}{k})$. Αφού έχουμε ανεξάρτητα γεγονότα και είναι k υποπίνακες συνολικά η πολυπλοκότητα προκύπτει: $\Omega\left(nlog(\frac{n}{k})\right)$

(β)

Η βασική ιδέα είναι να φτιάξουμε ένα δυαδικό AVL δέντρο αναζήτησης όπου κάθε κόμβος θα περιέχει μια λίστα από όμοια στοιχεία. Η διάταξη του δέντρου θα είναι ως προς το 1ο στοιχείο της λίστας (που είναι όμοιο με τα υπόλοιπα στοιχεία της λίστας για κάθε κόμβο). Όλες οι πράξεις στο δυαδικό δέντρο γίνονται ως προς το πρώτο στοιχείο των λιστών των κόμβων. Έτσι, η πολυπλοκότητα όλων των πράξεων (αναζήτηση, διαγραφή, εισαγωγή) είναι η ίδια με εκείνη των κλασσικών AVL δέντρων.

Κατασκευή δέντρου

Το δέντρο κατασκευάζεται ως εξής: Για κάθε στοιχείο της εισόδου μεγέθους n κάνουμε δυαδική αναζήτηση στο υπάρχον δέντρο. Αν βρούμε το στοιχείο στο δέντρο μας, το προσθέτουμε στη λίστα του κόμβου που βρήκαμε (η πράξη του append σε λίστα έχει πολυπλοκότητα O(1)). Αν δεν το βρούμε, κάνουμε εισαγωγή ενός νέου στοιχείου στο δέντρο που είναι μια λίστα με 1 στοιχείο, τον αριθμό που δεν βρήκαμε. Επειδή

στα δυαδικά δέντρα οι πράξεις της δυαδικής αναζήτησης και της εισαγωγής στοιχείου είναι ισοδύναμες θεωρούμε ότι για κάθε στοιχείο της εισόδου κάνουμε από άποψη πολυπλοκότητας 1 δυαδική αναζήτηση στο δέντρο. Αφού σε κάθε κόμβο του δέντρου υπάρχει 1 διακριτό στοιχείο και υπάρχουν το πολύ ($logn$) στοιχεία (από εκφώνηση), το ύψος του δέντρου θα είναι το πολύ ($loglogn$). Η πράξη της δυαδικής αναζήτησης έχει πολυπλοκότητα χειρότερης περίπτωσης ίδια με το ύψος του δυαδικού δέντρου. Άρα, για n αναζητήσεις που θα κάνουμε θα έχουμε O(nloglogn) πολυπλοκότητα χειρότερης περίπτωσης.

Ταξινόμηση

Αφού έχουμε κατασκευάσει το AVL δυαδικό δέντρο η ταξινόμηση αντιστοιχεί πλέον σε μια πράξη in order διάσχισης του δέντρου. Κάθε φορά που συναντάμε ένα στοιχείο ενώνουμε το τέλος της λίστας του με την αρχή της λίστας του επόμενου στοιχείου στην in-order διάσχιση. Όταν η in-order διάσχιση τελειώσει η ρίζα του δέντρου θα είναι μια λίστα που περιέχει ταξινομημένα όλα τα στοιχεία της εισόδου. Η πολυπλοκότητα της ένωσης δύο λιστών είναι η ίδια με την μετακίνηση ενός δείκτη, δηλαδή O(1). Άρα η πολυπλοκότητα αυτού του σταδίου του αλγορίθμου μας είναι η πολυπλοκότητα της in-order διάσχισης, δηλαδή O(logn) (αφού το δέντρο μας έχει το πολύ logn στοιχεία).

Συνολική πολυπλοκότητα

Η συνολική πολυπλοκότητα του αλγορίθμου μας είναι:

$$O(nloglogn + logn) = O(max\{nloglogn, logn\}) = O(nloglogn)$$

Ο λόγος που αίρεται το κάτω φράγμα του Ω(nlogn) είναι ότι δεν ταξινομούμε ως προς όλα τα στοιχεία αλλά μόνο ως προς τα διακριτά στοιχεία που στην περίπτωση μας είναι το πολύ O(logn).

Άσκηση 3

(α)

Ορισμός προβλήματος

Με δεδομένους δύο πίνακες $A_1[1...n_1], A_2[1...n_2]$ θέλουμε:

$$argmin_{i,j}(|A_1 - A_2|)$$

Διατύπωση αλγορίθμου

Κρατάμε δύο δείκτες, τον δείκτη p_1 για τον πίνακα A_1 και τον δείκτη p_2 για τον πίνακα A_2. Οι δύο δείκτες αρχικοποιούνται στην πρώτη θέση των αντίστοιχων πινάκων. Υπολογίζουμε κάθε φορά τη διαφορά μεταξύ των στοιχείων που δείχνουν οι δείκτες και ανανεώνουμε το min. Αν το στοιχείο που δείχνει ο δείκτης p_1 είναι μεγαλύτερο από το στοιχείο που δείχνει ο δείκτης p_2 προχωράμε τον p_2, αλλιώς προχωράμε τον p_1. Συνεχίζουμε, μέχρι κάποιος δείκτης να πρέπει να μετακινηθεί έξω από τα όρια του πίνακα του.

Τα παραπάνω μπορούν να εκφραστούν στον ακόλουθο ψευδοκώδικα:

Algorithm 2: Minimum length interval that covers both arrays

Data: $A_1[1...n_1], A_2[1...n_2]$
Result: $i, j : \quad argmin_{i,j}\,(|A_1[i] - A_2[j]|)$

Function `minim_cover`(A_1, A_2)**:**
- $p_1 \leftarrow 0$
- $p_2 \leftarrow 0$
- $minim \leftarrow \infty$
- $pointer_1 \leftarrow \&A_1[0]$
- $pointer_2 \leftarrow \&A_2[0]$
- **while** p_1 *and* p_2 *inside array bounds* **do**
  - **if** $abs(A[p_1] - A[p_2] < minim)$ **then**
    - $minim \leftarrow A[p_1]$
    - $pointer1 \leftarrow p_1$
    - $pointer2 \leftarrow p_2$
  - **if** $A[p_1] > A[p_2]$ **then**
    - p_2++
  - **else**
    - p_1++
- **return** $pointer_1, pointer_2, minim$

Απόδειξη ορθότητας

Για να αποδείξουμε ότι ο αλγόριθμος μας είναι ορθός αρκεί να αποδείξουμε δύο πράγματα:

1. Ότι κάθε φορά πρέπει να μετακινηθεί ο δείκτης που δείχνει στο μικρότερο στοιχείο.
2. Ότι με την μετακίνηση ενός δείκτη δεν χρειάζεται να υπολογίσουμε διαφορές μεταξύ στοιχείων που έδειχνε ο άλλος δείκτης στο παρελθόν.

   Έστω ότι μετακινήσαμε το δείκτη του $A_1[i]$ όταν ο δείκτης p_2 έδειχνε στο $A_2[j]$. Ο δείκτης p_1 τώρα δείχνει στη θέση $A_1[i+1]$. Αρκεί να δείξουμε ότι:

$$A_1[i+1] - A_2[j-1] > A_1[i+1] - A_2[j] \iff A_2[j-1] < A_2[j], \quad \text{που ισχύει.}$$

Αφού δείξαμε ότι ο τρόπος που μετακινούμε τους δείκτες είναι ορθός και ότι ο χώρος αναζήτησης των διαφορών είναι σωστά ορισμένος (δηλαδή μας δίνει το ίδιο αποτέλεσμα με το να ψάχναμε τις διαφορές για κάθε στοιχείο του άλλου πίνακα), ο αλγόριθμος μας είναι ορθός.

Υπολογιστική πολυπλοκότητα

Στον αλγόριθμο μας κάθε φορά μετακινούμε ένα δείκτη, είτε από τον έναν πίνακα είτε από τον άλλον, μέχρι να βγούμε έξω από τα όρια κάποιου πίνακα. Άρα, η υπολογιστική πολυπλοκότητα είναι:
$O(max(n_1, n_2)) = O(n_1 + n_2)$

(β)

Ο αλγόριθμος που διατυπώνουμε στο ερώτημα αυτό είναι μια γενίκευση του αλγορίθμου που διατυπώσαμε στο ερώτημα (α). Συγκεκριμένα, αρχικοποιούμε δείκτες $p_1, p_2, ..., p_m$ στην αρχή των αντίστοιχων πινάκων, υπολογίζουμε το min και το max μεταξύ των στοιχείων που δείχνουν οι πίνακες, βρίσκουμε τη διαφορά και προχωράμε το δείκτη που δείχνει στο μικρότερο στοιχείο. Στο τέλος, επιστρέφουμε τη μικρότερη διαφορά που υπολογίσαμε. Σε μορφή ψευδοκώδικα, έχουμε τα ακόλουθα:

Algorithm 3: Minimum length interval that covers all arrays

Data: $A_1[1...n_1], A_2[1...n_2], ..., A_m[1, 2, ..., n_m]$

Result: $i_1, i_2, ..., i_m$:

$argmin_{i_1,i_2,...,i_m}\ (max(A_1[i_1], A_2[i_2], ..., A_m[i_m]) - min(A_1[i_1], A_2[i_2], ..., A_m[i_m]))$

Function `minim_cover`(A_1, A_2)**:**

```
/* initialize all pointers                                          */
p_1, p_2, ..., p_m ← &A_1[0], &A_2[0], ..., &A_m[0]
pmin_1, pmin_2, ..., pmin_m ← null
minim ← ∞
while p_1, p_2, ..., p_m all inside array bounds do
    min_pointer, max_pointer ← get_min_max_pointers(A_1[p_1], ..., A_m[p_m])
    if A_{max_pointer[max_pointer]} − A_{min_pointer[min_pointer]} < minim then
        minim = A_{max_pointer[max_pointer]} − A_{min_pointer[min_pointer]}
        pmin_1, pmin_2, ..., pmin_m ← p_1, p_2, ..., p_m
    min_pointer++
return pointer_1, pointer_2, minim
```

Απόδειξη ορθότητας

Η ορθότητα του συγκεκριμένου αλγορίθμου προκύπτει ως μια γενίκευση της ορθότητας για τους 2 πίνακες. Αν προχωρήσουμε το δείκτη οποιουδήποτε άλλου στοιχείου εκτός από το μέγιστο ή διαφορά είτε θα αυξηθεί είτε θα μείνει ίδια γιατί το ελάχιστο μένει ίδιο ενώ το μέγιστο ή μένει ίδιο ή αυξάνεται. Έτσι, κάθε φορά η επιλογή μετακίνησης του μικρότερου στοιχείου εξασφαλίζει ότι κινούμαστε προς την βέλτιστη λύση με δεδομένα τα άλλα στοιχεία που δεν μετακινήθηκαν σε αυτή την κίνηση.

Υπολογιστική πολυπλοκότητα

Ο αλγόριθμος μας έχει υπολογιστική πολυπλοκότητα $\mathbf{O(N \cdot m)}$. Η αιτιολόγηση είναι ότι διατρέχουμε γραμμικά τα στοιχεία των πινάκων που έχει πολυπλοκότητα O(N) και σε κάθε θέση υπολογίζουμε το min και το max μεταξύ m στοιχείων το οποίο έχει πολυπλοκότητα O(m).

(γ)

Η λειτουργία που καθορίζει το χρόνο εκτέλεσης του αλγορίθμου στο (β) είναι ότι σε κάθε θέση πρέπει να βρούμε το min στοιχείο το οποίο έχει πολυπλοκότητα O(m). Στην πραγματικότητα αυτή η λειτουργία μπορεί να γίνει πιο αποδοτικά. Αρχικά, δημιουργούμε ένα σωρό με στοιχεία το πρώτο στοιχείο από τον κάθε πίνακα. Η λειτουργία αυτή θέλει χρόνο O(m). Στη συνέχεια, για να βρούμε το min στην κάθε θέση κάνουμε extract από τον σωρό με πολυπλοκότητα O(logm). Με την μετακίνηση του δείκτη κάνουμε insert στο σωρό του νέου στοιχείου, με πολυπλοκότητα πάλι O(logm). Συνεπώς, σε κάθε μετακίνηση δείκτη έχουμε πολυπλοκότητα $O(2logm) = O(logm)$.

Επειδή θα γίνουν το πολύ N μετακινήσεις δεικτών (δηλαδή μπορεί να χρειαστεί να προσπελάσουμε το πολύ όλα τα στοιχεία), η συνολική πολυπλοκότητα του αλγορίθμου μας βελτιώνεται σε: **O(Nlogm)**

Άσκηση 4

(α)

Χρειαζόμαστε 20 εθελοντές. Αριθμούμε τις φιάλες από το 1 έως και το 1.000.000. Σε κάθε φιάλη, κολλάμε μια ετικέτα, που είναι η δυαδική αναπαράσταση μήκους 20 bits της αρίθμησης της. Το μήκος 20 bits επιλέχθηκε έτσι καθώς τόσα bit χρειαζόμαστε για να αναπαραστήσουμε στο δυαδικό σύστημα τον αριθμό 1.000.000. Αριθμούμε τους εθελοντές από το 1 έως το 20. Ο E_i εθελοντής θα πιεί από τη φιάλη Φ_j αν το i MSB (Most Signifficant Bit) της ετικέτας της Φ_j φιάλης είναι 1, αλλιώς δεν θα πιει. Μόλις περάσει

το 24ωρο, βάζουμε πάλι τους εθελοντές στη σειρά (στην ίδια σειρά που ήταν και πριν) και για κάθε εθελοντή βάζουμε στην αντίστοιχη θέση 1 αν έχει υπερδυνάμεις ή 0 αν δεν έχει. Έτσι, σχηματίζεται ένας δυαδικός αριθμός μήκους 20 bits. Η φιάλη που έχει αυτό τον αριθμό ως ετικέτα είναι και η φιάλη που περιέχει το μαγικό υγρό που ψάχνουμε.

(β)

Ορισμός προβλήματος

Θέλουμε να κάνουμε ένα ταξίδι που θα διαρκέσει k ημέρες και θέλουμε να επισκεφθούμε $n > k$ σταθμούς που έχουν σχετικές αποστάσεις: $d_1, d_2, ..., d_n$. Ζητούμενο είναι να βρούμε τους σταθμούς που θα διανυκτερεύσουμε ώστε να ελαχιστοποιήσουμε τη μέγιστη απόσταση που θα διανύσουμε σε μια μέρα.

Διατύπωση αλγορίθμου

Η ελαχιστη απόσταση που μπορούμε να διανύσουμε σε μια ημέρα είναι 0 (να μείνουμε στον ίδιο σταθμό) και η μέγιστη απόσταση που μπορούμε να διανύσουμε είναι: $d_1 + d_2 + ... + d_n$ (να επισκεφθούμε όλους τους σταθμούς σε μια ημέρα).

Η λύση του προβλήματος μας θα είναι μια απόσταση: η μέγιστη απόσταση που θα διανύσουμε σε μια ημέρα. Θέλουμε αυτή η λύση να είναι ελάχιστη. Μπορούμε λοιπόν, να την ψάξουμε με δυαδική αναζήτηση στον ταξινομημένο χώρο αποστάσεων. Συγκεκριμένα, θα ξεκινήσουμε να ψάχνουμε σε όλο το χώρο, δηλαδή στο διάστημα $[0, d_1 + d_2 + ... + d_n]$, και κάθε φορά θα εξετάζουμε μια πιθανή απόσταση ως λύση του προβλήματος μας. Θα ελέγχουμε αν είναι δυνατό να γίνει με αυτή την απόσταση κάθε μέρα να καλύψουμε όλους τους σταθμούς και αν γίνεται θα ψάχνουμε αναδρομικά στο πρώτο μισό του πίνακα προκειμένου να την μειώσουμε παραπάνω. Αλλιώς, αν δεν γίνεται σημαίνει ότι η μέγιστη απόσταση πρέπει να αυξηθεί και συνεπώς η αναζήτηση θα γίνεται αναδρομικά στο δεύτερο μισό του πίνακα.

Ο έλεγχος για το αν γίνεται με τη συγκεκριμένη απόσταση είναι γραμμικός ως προς το πλήθος των σταθμών. Συγκεκριμένα, ξεκινάμε από την αφετηρία και αθροίζουμε αποστάσεις σταθμών μέχρι ακριβώς πριν να ξεπεράσουμε την τρέχουσα απόσταση που έχει προκύψει από την δυαδική αναζήτηση, οπότε και αυξάνουμε έναν δείκτη που μετράει πόσες μέρες έχει διαρκέσει το ταξίδι ως τώρα. Αν ο δείκτης αυτός ξεπεράσει τις k ημέρες που θέλουμε να διαρκέσει το ταξίδι τότε δεν γίνεται. Σε διαφορετική περίπτωση, δηλαδή αν επισκεφθούμε όλους τους σταθμούς σε λιγότερο ή ίσο από k ημέρες τότε γίνεται.

Σε μορφή ψευδοκώδικα, ο αλγόριθμος που περιγράψαμε είναι ως εξής:

Algorithm 4: Minimizing the maximum distance we cover each day

Data: $k, d_1, d_2, ..., d_n$
Result: d

$left \leftarrow 0$
$right \leftarrow d_1 + d_2 + ... + d_n$
$minim \leftarrow \infty$
while $left < right$ **do**
    $middle \leftarrow (left + right)/2$
    /* D is distances matrix */
    $possible \leftarrow check\_if\_possible(D, k, middle)$
    if *possible* **then**
        $minim \leftarrow middle$
        $right \leftarrow middle - 1$
    else
        $left \leftarrow middle + 1$
/* minim now has the result */
bool `check_if_possible`(D, k, d)**:**
    $count \leftarrow 0$
    $summ \leftarrow 0$
    for $i = 0; i < size(D); i++$ **do**
        if $summ + D[i] > d$ **then**
            $count \leftarrow count + 1$ $summ \leftarrow D[i]$
        else
            $summ \leftarrow summ + D[i]$
    if $count <= k$ **then**
        return true;
    else
        return false;

Ορθότητα αλγορίθμου

Ο αλγόριθμος μας είναι ορθός γιατί πληρεί τα ακόλουθα κριτηρία:

1. Πληρότητα ως προς το χώρο αναζήτησης της λύσης

   Η χώρος που ψάχνουμε την απάντηση του προβλήματος είναι πλήρης καθώς δεν γίνεται να διανύσουμε λιγότερο από 0 χιλιόμετρα κάποια μέρα ούτε γίνεται να διανύσουμε σε μια μέρα περισσότερα χιλιόμετρα από όσα είναι τα χιλιόμετρα του συνολικού ταξιδιού.

2. Η διάσχιση του χώρου αναζήτησης εξασφαλίζει την εύρεση της καλύτερης λύσης.

   Τον χώρο αναζήτησης δεν τον διασχίζουμε με γραμμικό τρόπο αλλά με λογαριθμικό μέσω της δυαδικής αναζήτησης. Από τη στιγμή που βλέπουμε ότι μια απόσταση είναι ορθή, ψάχνουμε να δούμε αν έχουμε κάνει υπερεκτίμηση της ελάχιστης μέγιστης απόστασης που μπορούμε να ψάχνουμε κάθε μέρα, οπότε ψάχνουμε στο πρώτο μισό του πίνακα. Το δεύτερο μισό δεν χρειάζεται να το ψάξουμε γιατί έχουμε βρει ήδη μια ορθή απόσταση και οποιαδήποτε απόσταση στο δεύτερο μισό λόγω της ταξινομημένης διάταξης θα είναι μεγαλύτερη από την απόσταση που έχουμε ήδη βρει. Από την άλλη, αν κάποια απόσταση δεν είναι ορθή καμία μικρότερη απόσταση της δεν θα είναι ορθή. Αυτό σημαίνει ότι αν δεν μπορούμε να διασχίζουμε το πολύ d χιλιόμετρα κάθε μέρα ώστε να δούμε όλους τους σταθμούς ολοκληρώνοντας το ταξίδι σε k μέρες τότε σίγουρα δεν μπορούμε να διασχίζουμε το πολύ k χιλιόμετρα κάθε μέρα, με $k < d$. Έτσι, η αναζήτηση μόνο στο δεύτερο μισό του πίνακα σε αυτή την περίπτωση είναι ορθή.

3. Ο έλεγχος για το αν μπορούμε να διανύσουμε όλους τους σταθμούς σε k μέρες με το πολύ d χιλιόμετρα την μέρα είναι ορθός.

   Η απόδειξη αυτής της πρότασης είναι διαισθητικά προφανής, καθώς αυτό που κάνουμε είναι να σταματάμε στα d χιλιόμετρα κάθε μέρα. Αν μας τελειώσουν οι μέρες και έχουν μείνει ακόμα χιλιόμετρα (δηλαδή σταθμοί), προφανώς δεν γίνεται. Σε διαφορετική περίπτωση, γίνεται.

Υπολογιστική πολυπλοκότητα

Ο αλγόριθμος μας κάνει δυαδική αναζήτηση σε έναν πίνακα $d_1 + d_2 + ... + d_n$ τιμών και για κάθε στοιχείο της δυαδικής αναζήτησης κάνει έναν γραμμικό έλεγχο ως προς το πλήθος τον σταθμών. Άρα, η πολυπλοκότητα του αλγορίθμου μας είναι: $\mathbf{O\left(nlog(d_1 + d_2 + ... + d_n)\right)}$

Άσκηση 5

(α)

Ορισμός προβλήματος

Είμαστε εφοδιασμένοι με μια συνάρτηση $\mathcal{F}_s(l)$ και με ένα άγνωστο πολυσύνολο $\mathcal{S}$ από άγνωστα στοιχεία που ανήκουν στο χώρο των ακεραίων και φράσσονται από έναν επίσης ακέραιο αριθμό M.

Για την $\mathcal{F}_s$ έχουμε:

$$F_s(l) = |\{x \in \mathcal{S} : x \leq l\}|$$

Θέλουμε να βρούμε τον βέλτιστο αλγόριθμο για να βρούμε το $k-$οστό ελάχιστο στοιχείο του $\mathcal{S}$.

Διατύπωση αλγορίθμου

Έστω ότι το στοιχείο που ψάχνουμε είναι το x. Για το x ισχύει ότι:

$$F_s(x-1) < k \quad \wedge \quad \mathcal{F}_s(x+1) > k$$

Από τη στιγμή που ψάχνουμε ένα στοιχείο που μπορούμε να αποφανθούμε σε O(1) αν είναι το σωστό σε ένα χώρο λύσεων και υπάρχει μια μετρική σχετικά με το αν απομακρυνόμαστε ή πλησιάζουμε στη λύση μας γεννάται απευθείας η ιδέα της δυαδικής αναζήτησης.

Ο χώρος λύσεων στον οποίο ψάχνουμε είναι ένας πίνακας με στοιχεία από το 1 εώς και το M. Κάθε φορά δοκιμάζουμε ένα τρέχον number $curr\_number$ για το αν ικανοποιεί την ιδιότητα που θέλουμε. Αν την ικανοποιεί, το βρήκαμε. Αν δεν την ικανοποιεί, ανάλογα με το αν το $\mathcal{F}_s(curr\_number)$ είναι μεγαλύτερο ή μικρότερο από το k χωρίζουμε τον πίνακα στα δύο και ψάχνουμε στο αριστερό μισό ή στο δεξί μισό αντίστοιχα.

Η πολυπλοκότητα της λύσης μας είναι η πολυπλοκότητα της δυαδικής αναζήτησης σε πίνακα μήκους M, δηλαδή $O(log_2 M)$.

Ορθότητα αλγορίθμου

Η ορθότητα του αλγορίθμου που διατυπώνουμε στην ενότητα αυτή έχει τις ίδιες ιδέες με την ορθότητα του αλγορίθμου που αναπτύξαμε στην άσκηση 4β. Συγκεκριμένα:

1. Ο χώρος αναζήτησης είναι πλήρης.

   Αυτό σημαίνει ότι ο χώρος στον οποίο αναζητάμε περιέχει όλες τις πιθανές λύσεις του προβλήματος αφού το k-οστό μικρότερο στοιχείο του πολυσυνόλου θα ανήκει πάντα από 1 εως M.

2. Η διάσχιση του χώρου αναζήτησης εξασφαλίζει την εύρεση λύσης.

   Αφού ψάχνουμε σε έναν ταξινομημένο χώρο ένα στοιχείο που ικανοποιεί μια συγκεκριμένη ιδιότητα η δυαδική αναζήτηση μας εγγυάται ότι το στοιχείο αυτό αν υπάρχει θα βρεθεί.

Bonus - Υλοποιήση σε C++

Μια υλοποίηση σε C++ του συγκεκριμένου προβλήματος φαίνεται παρακάτω:

```
#include <iostream>
using namespace std;
#include <random>
#include <math.h>
#include <algorithm>

const int N = pow(10, 6);
const int M = pow(10, 8);
const int kmin = 1000;
const int minim = 1;
const int maxim = (int) M/2;

int arr[N];

random_device rd;
mt19937 rng(rd());
uniform_int_distribution<int> uni(minim,maxim);

int count_less(int number){
int count = 0;
for (int i=0; i<N; i++){
if (arr[i] < number) count++;
}
return count;

}
int main(void){
for (int i=0; i<N; i++){
arr[i] = uni(rng);
}
int steps = 0;
sort(arr, arr + N);
cout << "Element we are looking for" << endl;
cout << arr[kmin-1] << endl;
int left = 0;
int right = M;
bool flag = false;
while (!flag){
steps += 1;
int curr_num = (left + right) / 2;
int counted = count_less(curr_num);
if (counted >= kmin){
right = curr_num;
}
else if (counted < kmin){
left = curr_num;
}
if (abs(left - right) == 1) flag = true;
}
cout << "Prediction" << endl;
cout << left << endl;
cout << "Steps" << " " << steps << endl;
}
```

(β)

Διατύπωση αλγορίθμου

Θα χρησιμοποιήσουμε τον αλγόριθμου του ερωτήματος (α). Όλα τα στοιχεία του συνόλου S φράσσονται από το μέγιστο στοιχείο του πίνακα M. Όπως είδαμε στο ερώτημα (α) για να υπολογίσουμε το k-οστό μικρότερο στοιχείο ενός πολυσυνόλου S κάνουμε το πολύ $logM$ φορές χρήση της συνάρτησης F_s. Άρα, η πολυπλοκότητα του αλγορίθμου μας είναι: $O(logM \cdot F_s)$.

Από τα παραπάνω, το πρόβλημα μας ανάγεται στην ελαχιστοποίηση της πολυπλοκότητας της συνάρτησης F_s. Θέλουμε να πετύχουμε μια καλύτερη πολυπλοκότητα από την $O(n^2)$ που προκύπτει από τον υπολογισμό όλων των διαφορών. Η ιδέα μας είναι ότι η $F_s(k)$ μπορεί να γίνει γραμμική ως προς το πλήθος των στοιχείων του πίνακα, αν ο πίνακας είναι αρχικά ταξινομημένος. Ο αλγόριθμος που προτείνουμε εκφράζεται στον ακόλουθο ψευδοκώδικα:

Algorithm 5: Efficient implementation of $F_s(k)$

Data: $A[1, 2, ..., n], k$
Result: The number of positive differences of A smaller than k.

Function `f(`A, k`)`**:**

```
N ← size(A)
count ← 0
count_prev ← 0
p1 ← N
p2 ← N − 1
while p1, p2 in proper array bounds do
    if p1 == p2 then
        p2 ← p2 + 1
    if A[p1] − A[p2] ≤ k then
        count_prev = count_prev + 1
        p2 ← p2 + 1
    else
        p1 ← p1 + 1
        count ← count + count_prev
        count_prev ← max(0, count_prev − 1)
return counter
```

Αυτή η λύση έχει πολυπλοκότητα υπολογισμού της F_s $\mathbf{O(n)}$, δεδομένου ότι ο πίνακας A είναι αρχικά ταξινομημένος.

Απόδειξη ορθότητας

Σημαντική παρατήρηση: Σε όλη αυτή την ενότητα θεωρούμε ότι οι δείκτες προχωράνε από το τέλος προς την αρχή του πίνακα. Έτσι, όταν λέμε ότι ο δείκτης p_1 προχωράει κατά 1 εννοούμε ότι κοιτάει στο στοιχείο με το αμέσως μικρότερο index. Επίσης, όταν αναφερόμαστε στα στοιχεία πριν το στοιχείο στο οποίο δείχνει ο p_1 αναφερόμαστε στα στοιχεία με μεγαλύτερα indexes από τα στοιχεία στα οποία δείχνει ο p_1.

Ο αλγόριθμος που περιγράψαμε παραπάνω μας λέει ότι ξεκινάμε από το τέλος του πίνακα και προχωράμε με δύο δείκτες. Ο ένας δείκτης δείχνει στο στοιχείο το οποίο θέλουμε να μετρήσουμε πόσες θετικές διαφορές έχουν αυτό ως μέγιστο στοιχείο και είναι στα επιτρεπτά όρια που επιβάλλει το k και ο δεύτερος δέκτης μας δείχνει το στοιχείο με το οποίο εξετάζουμε τη διαφορά του αυτή τη χρονική στιγμή. Ο αλγόριθμος έχει κάποιες βασικές ιδέες πάνω στις οποίες θα τεκμηριώσουμε την ορθότητα του:

1. Όλες οι διαφορές που μας ενδιαφέρουν θα έχουν πάντα κάποιο από τα στοιχεία του πίνακα ως μέγιστο στοιχείο. Άρα, για να βρούμε όλες τις μη αρνητικές διαφορές αρκεί να μετρήσουμε πόσες διαφορές μας δίνει κάθε στοιχείο όταν αυτό είναι το μέγιστο στοιχείο της διαφοράς.

2. Ο δείκτης p_2 θα δείχνει πάντα σε μικρότερο στοιχείο από το στοιχείο που θα δείχνει ο δείκτης p_1.

   Στην αρχικοποίηση που κάνουμε αυτό ισχύει. Στη συνέχεια, αφού κάθε φορά που είναι ίσοι οι δύο δείκτες προχωράμε τον δείκτη p_2 κατά 1 προς την αρχή ενός ταξινομημένου πίνακα, εξασφαλίζεται η αλήθεια της πρότασης.

3. Όταν προχωράμε τον δείκτη p_1 οι διαφορές που χρειάζεται να εξετάσουμε ξεκινάνε από τον δείκτη p_2 και μετά.

   Η αλήθεια αυτής της πρότασης τεκμηριώνεται ως εξής:

   - Για όλα τα στοιχεία που είναι πριν από τον p_1 (δηλαδή τα στοιχεία με μεγαλύτερα indexes), δεν χρειάζεται να μετρήσουμε διαφορές αφού θα είναι όλες αρνητικές λόγω της ταξινομημένης διάταξης του πίνακα.
   - Για τα στοιχεία που είναι ανάμεσα στο p_1 και στο p_2 έχουμε ότι αληθεύει πάντα η πρόταση ότι η διαφορά τους θα είναι μέσα στα επιτρεπτά όρια, αφού η διαφορά ήταν μέσα στα επιτρεπτά όρια και για το προηγούμενο στοιχείο στο οποίο έδειχνε ο p_1, όταν δηλαδή η διαφορά ήταν μεγαλύτερη. Πρακτικά, αυτό σημαίνει ότι η διαφορά είναι φθίνουσα για σταθερό p_2 όσο μειώνονται τα $indexes$ στα οποία δείχνει ο p_1. Έτσι δικαιολογείται στον αλγόριθμο μας η χρήση ενός $count\_prev$. Το χρησιμοποιούμε για να πούμε ότι οι διαφορές που έχουν ως μέγιστο στοιχείο το στοιχείο του p_1 με τα στοιχεία που είναι μεταξύ του p_1, p_2 θα είναι όσες πριν - 1 (αφαιρούμε το 1 για να βγάλουμε την αρνητική πλέον διαφορά μεταξύ των θέσεων που έδειχνε ο p_1 πριν και δείχνει ο p_1 τώρα).

4. Αν η διαφορά του στοιχείου που δείχνει ο δείκτης p_1 με το στοιχείο που δείχνει ο δείκτης p_2 είναι έξω από τα επιτρεπτά όρια, όλες οι διαφορές που προκύπτουν μεταξύ του στοιχείου που δείχνει ο δείκτης p_1 και των στοιχείων που παίρνουμε προχωρώντας το δείκτη p_2 θα είναι έξω από τα επιτρεπτά όρια.

   Πρακτικά αυτό σημαίνει ότι η διαφορά για σταθερό p_1 είναι αύξουσα όσο μειώνονται τα $indexes$ στα οποία δείχνει ο p_2. Αυτό είναι λογικό καθώς ο πίνακας είναι ταξινομημένος και έχουμε ότι:

$$\text{Για } a < b < c: \quad c - b < c - a$$

Από τα παραπάνω, δικαιολογείται πλήρως η ορθότητα του αλγορίθμου μας.