ΑΛΓΟΡΙΘΜΟΙ & ΠΟΛΥΠΛΟΚΟΤΗΤΑ

1η Σειρά γραπτών ασκήσεων

Ακαδημαϊκό έτος 2021-2022

7ο εξάμηνο

Νικόλας Μπέλλος | ΑΜ : el18183

---

*Στην εργασία εμπεριέχονται εξισώσεις σε μορφή φωτογραφίας γραμμένες σε Latex.

Άσκηση 1

Master Theorem

Για αναδρομικούς τύπους της παρακάτω μορφής ισχύει :

$$T(n) = aT(\frac{n}{b}) + \Theta(n^d),\ d \geqslant 0$$

- $If\ a > b^d : T(n) = \Theta(n^{log_b a})$
- $If\ a = b^d : T(n) = \Theta(n^d logn)$
- $If\ a < b^d : T(n) = \Theta(n^d)$

1. $T(n) = 4T(n/2) + \Theta(n^2 logn)$

Τι περιμένουμε ?

Περιμένουμε κάτι κοντά στο $\Theta(n^2 log^2 n)$ λόγω του ότι $a=b^d$ από τη παραπάνω σχέση.

<u>Κάτω φράγμα :</u>

Παίρνουμε την : $T'(n) = 4T'(n/2) + \Theta(n^2) < T(n) = 4T(n/2) + \Theta(n^2 logn)$

η οποία από τον τύπο του master theorem προκύπτει ίση με $\Theta(n^2 logn)$.

Άρα η πολυπλοκότητα είναι σίγουρα μεγαλύτερη από $\Theta(n^2 logn)$.

Αναλύοντας όμως τα iterations μπορούμε να βρούμε κάτι καλύτερο :

$$1st\ iteration : n^2 logn$$
$$2nd\ iteration : 4 \cdot (n/2)^2 \cdot log(n/2) = 4 \cdot n^2(logn - 1)/4 = n^2(logn - 1)$$
$$3rd\ iteration : 4^2 \cdot (n/2^2)^2 \cdot log(n/2^2) = 4^2 \cdot n^2(logn - 2)/4^2 = n^2(logn - 2)$$
$$\vdots$$
$$kth\ iteration : 4^{k-1} \cdot (n/2^{k-1})^2 \cdot log(n/2^{k-1}) = 4^{k-1} \cdot n^2(logn - (k-1))/4^{k-1} = n^2(logn - (k-1))$$

Παρατηρούμε ότι κάθε βήμα εκτελούνται κάτι λιγότερο από $n^2 logn$ πράξεις. Επίσης ξέρουμε ότι το recursion tree αυτού του αλγορίθμου έχει ύψος $log_2 n$ λόγω του όρου $T(n/2)$ και άρα τα iterations θα είναι ακριβώς logn. Επομένως, συνολικά η πολυπλοκότητα θα είναι **$T(n) = \Theta(n^2 logn) * \Theta(logn) = \Theta(n^2 log^2 n)$**

2. $T(n) = 5T(n/2) + \Theta(n^2 \log n)$

Τι περιμένουμε ?

Περιμένουμε κάτι κοντά στο $\Theta(n^{\log 5})$ λόγω του ότι $a > b^d$ από τη παραπάνω σχέση.

Υπολογίζοντας όπως και στο προηγούμενο τύπο το αποτέλεσμα για κάθε iteration / επίπεδο του recursion tree, προκύπτει ότι σε κάθε iteration (έστω k) εκτελούνται :

$5^k n^2(logn - k)/2^{2*k} = n^2 logn * (\frac{5}{4})^k$ πράξεις.

Και πάλι το ύψος του δέντρου είναι logn και από το άθροισμα της γεωμετρικής προόδου για $a_1 = n^2 \log n$, $\lambda = 5/4$, $\nu = \log n$ προκύπτει ότι η πολυπλοκότητα είναι :

$$T(n) = \sum_{\nu} = n^2 logn \cdot (\frac{(\frac{5}{4})^{logn} - 1}{\frac{5}{4} - 1}) \Rightarrow n^2 logn \cdot (\frac{5}{4})^{logn} \Rightarrow n^{log_2 5}$$

Τελική πολυπλοκότητα : $\mathbf{\Theta(n^{\log 5})}$

3. $T(n) = T(n/4) + T(n/2) + \Theta(n)$

Τι περιμένουμε ?

Περιμένουμε κάτι κοντά στο $\Theta(n)$ γιατί το $T(n) = 2T(n/2) + \Theta(n)$ (sorting) έχει πολυπλοκότητα $\Theta(n \log n)$ και λειτουργεί ως άνω όριο, ενώ ταυτόχρονα η πολυπλοκότητα δεν μπορεί να πέσει κάτω από $\Theta(n)$ λόγω του πρώτου iteration.

Από το recursion tree, μπορούμε να διαπιστώσουμε ότι το άθροισμα των πράξεων που χρειάζονται σε κάθε επίπεδο του δέντρου μειώνεται πάρα πολύ γρήγορα (στο 3ο επίπεδο έχουν πέσει κάτω από $n/2$), οπότε δεν μπορεί να πλησιάζει το $\Theta(n \log n)$. Επίσης, η παραπάνω σχέση είναι ισοδύναμη με την $T(n) = T(3n/4) + \Theta(n)$, όπου από το Master Theorem ισχύει ότι $a < b^d$ και άρα η πολυπλοκότητα ανάγεται σε $\mathbf{T(n)} = \Theta(n^d) = \mathbf{\Theta(n).}$

4. $T(n) = 2T(n/4) + T(n/2) + \Theta(n)$

Τι περιμένουμε ?

Περιμένουμε κάτι κοντά στο $\Theta(n \log n)$ γιατί θυμίζει πολύ την αναδρομική σχέση των sorting αλγορίθμων.

Το ύψος του recursion tree αρχικά είναι logn λόγω του όρου $T(n/2)$. Σε κάθε επίπεδο του δέντρου παρατηρούμε ότι το άθροισμα των παιδιών ενός πατέρα κόμβου δεν μειώνεται και σε κάθε επίπεδο γίνονται n πράξεις. Την παραπάνω σχέση μπορούμε να την ανασχηματίσουμε και στην $T(n) = 2T(n/2) + \Theta(n)$ και από το Master Theorem για $a = b$ να βρούμε την πολυπλοκότητα. Αυτή και με τους δύο τρόπους προκύπτει ότι είναι $\mathbf{\Theta(n \log n).}$

5. $T(n) = T(n^{1/2}) + \Theta(\log n)$

Τι περιμένουμε ?

Δεν περιμένουμε κάτι μεγαλύτερο από $\Theta(\log n)$ γιατί θυμίζει τη περίπτωση του $T(n) = T(n/2) + \Theta(n)$.

Υλοποιώντας τα πρώτα iterations μπορούμε εύκολα να διαπιστώσουμε ότι η πολυπλοκότητα ανάγεται σε ένα άθροισμα γεωμετρικής προόδου.

1st iteration : $\log n$

2nd iteration : $\log n^{1/2} = \log n/2$

3rd iteration : $\log n^{1/4} = \log n/4$

4th iteration $\log n^{1/8} = \log n/8$:

…

Άρα για $a_1=\log n$, $\lambda=1/2$, $\nu=\log n$ προκύπτει :

$$T(n) = \sum_{\nu} = logn \cdot \frac{1/2^{logn} - 1}{1/2 - 1} \Rightarrow T(n) = 2 \cdot logn \Rightarrow T(n) = logn$$

Τελική πολυπλοκότητα : **$\Theta(\log n)$**

6. $T(n/4) + \Theta(\sqrt{n})$

Εδώ μπορούμε να εφαρμόσουμε εύκολα το Master Theorem όπως αναφέρεται παραπάνω. Για $a=1$, $b=4$ και $d=½$ παρατηρούμε ότι $a < b^{1/2}$ και επομένως προκύπτει ότι η πολυπλοκότητα θα είναι **$T(n) = \Theta(n^d) = \Theta(n^{1/2})$.**

Άσκηση 2

(α) Αλγόριθμος ταξινόμησης (2n) :

1. Βρες το μεγαλύτερο στοιχείο

2. Κάνε περιστροφή μέχρι εκείνο το στοιχείο (δηλαδή φέρτο στην αρχή)

3. Κάνε περιστροφή όλο το πίνακα

4. Πήγαινε στο βήμα 1

Για να βρούμε πόσες το πολύ περιστροφές γίνονται παίρνουμε τη χειρότερη περίπτωση για κάθε βήμα. Για πίνακα A[1:n] θα χρειαστούν 2 περιστροφές για κάθε στοιχείο στη χειρότερη περίπτωση. Αυτό όμως ισχύει μέχρι τα n-2 πρώτα στοιχεία (άρα μέχρι εκεί γίνονται 2(n-2) = 2n-4 περιστροφές). Για τα τελευταία 2 στοιχεία είτε αυτά θα είναι ταξινομημένα (0.5 πιθανότητα) και άρα χρειάζονται 0 επιπλέον περιστροφές, είτε θα είναι αντιστραμένα, οπότε θα χρειάζεται μόνο 1 περιστροφή. Επομένως, προκύπτει ότι στη χειρότερη περίπτωση θα έχουμε 2n-4+1=2n-3 περιστροφές.

(β) Αλγόριθμος ταξινόμησης (3n) :

1. Βρες το μεγαλύτερο στοιχείο κατά απόλυτη τιμή.

2. Κάνε περιστροφή μέχρι εκείνο το στοιχείο (δηλαδή φέρτο στην αρχή)

3. Αν το πρώτο στοιχείο είναι θετικό κάνε μία περιστροφή στο πρώτο στοιχείο.

4. Κάνε περιστροφή όλο το πίνακα

5. Πήγαινε στο βήμα 1

Στη χειρότερη περίπτωση για τα πρώτα n-2 στοιχεία που ταξινομούνται θα χρειαστεί να κάνουν και μία τρίτη περιστροφή στο πρώτο στοιχείο. Άρα χρειάζονται $3(n-2)=3n-6$ περιστροφές. Τώρα, για τα τελευταία 2 στοιχεία, στη χειρότερη περίπτωση (1ο αρνητικό, 2ο αρνητικό) θα χρειαστούν 4 περιστροφές. Άρα συνολικά χρειάζονται στη χειρότερη περίπτωση $3n-6+4=3n-2$ περιστροφές (αν και η πρακτική πολυπλοκότητα είναι αρκετά καλύτερη).

(γ) 1. Στη γενική περίπτωση, μπορούμε να δημιουργήσουμε συμβατά ζεύγη για τα a,b, όπου a ένα στοιχείο αριστερότερα του b στις εξής περιπτώσεις :

a. Καταχρηστική περίπτωση - <u>Αν το στοιχείο n</u> (κατά απόλυτη τιμή) <u>δεν είναι στο τέλος</u> του πίνακα το μετακινούμε στο τέλος με 2 περιστροφές (υποθέτουμε ότι γίνεται στην αρχή που όλα τα στοιχεία είναι θετικά).

b. <u>Αν $a>0, b>0 : a<b$</u>

Με 1 περιστροφή στο a το μετακινούμε στην αρχή και με 1 ακόμα το μετακινούμε στο κελί πριν το b.

c. <u>Αν $a<0, b<0 : a<b$</u>

Ίδιες περιστροφές με το b, λόγω ότι και στις δύο περιπτώσεις διατηρείται το πρόσημο.

d. <u>Αν $a>0, b<0 : |a|<|b|$</u>

Με 1 περιστροφή στο b το μετακινούμε στην αρχή και με ένα ακόμα το μετακινούμε πριν το a (γίνονται αρνητικά και τα δύο)

e. <u>Αν $a<0, b>0 : |a|>|b|$</u>

Ίδια διαδικασία με το d (μόνο που τώρα τα a,b γίνονται και τα δύο θετικά).

Θα αποδείξουμε τώρα, ότι μπορούμε πάντα να εφαρμόσουμε κάποια από αυτές τις περιπτώσεις στον πίνακα A_t.

<u>Περιπτώσεις :</u>

1. Το στοιχείο n δεν είναι στο τέλος του πίνακα A (στην αρχή του αλγορίθμου).
2. Θεωρώντας ότι το στοιχείο n είναι ταξινομημένο και θετικό, ψάχνουμε να βρούμε τον αμέσως μικρότερο αριθμό (κατά απόλυτη τιμή) ο οποίος να μην έχει δημιουργήσει ήδη ζευγάρι με τον μεγαλύτερό του.
    a. Αν αυτός είναι θετικός έχουμε τη περίπτωση **b**

b. Αν αυτός είναι αρνητικός ψάχνουμε τον αμέσως μικρότερο αριθμό (κατά απόλυτη τιμή) που να έχει θετικό πρόσημο.
   i. Αν υπάρχει τέτοιος αριθμός τότε ανήκει στη περίπτωση **d ή e** (γιατί ξέρουμε σίγουρα ότι θα υπάρχει ένας μεγαλύτερος (κατά απόλυτη τιμή) ο οποίος θα είναι αρνητικός).
   ii. Αν δεν υπάρχει τέτοιος αριθμός, τότε όλοι οι υπόλοιποι είναι αρνητικοί και δεδομένου ότι ο πίνακας δεν είναι ο $A_t = [-1, -2, .., -n]$ τότε βρισκόμαστε είτε στη περίπτωση **c** είτε στην $A_t = [-n, -(n-1), .., -1]$ όπου για n αριθμούς ξέρουμε ότι υπάρχουν n-1 ζεύγη και άρα έχουμε κάνει το πολύ 2(n-1) περιστροφές για ν. Κάνοντας άλλη μία τον ταξινομούμε με λιγότερο από 2n περιστροφές.

Επαγωγικά, προσθέτωντας στοιχεία στο πίνακα Α διαπιστώνουμε ότι πάντα θα ισχύει μία από τις παραπάνω περιπτώσεις. Στη γενική περίπτωση, αν έχουμε k αριθμούς αυτοί θα δημιουργούν το πολύ k-1 ζεύγη τα οποία όπως αποδείξαμε πιο πάνω θα χρειάζονται το πολύ 2k περιστροφές για να ταξινομηθούν.

(γ) 2. Για να παράξουμε τον αλγόριθμο θα βασιστούμε στα βήματα που αναλύσαμε στο 2.γ1.

Αλγόριθμος ταξινόμησης (2n) :

1. Αν το στοιχείο n δεν είναι στο τέλος του πίνακα το μετακινούμε με το πολύ 2 κινήσεις

2. Βρίσκουμε τον αμέσως μικρότερο αριθμό (κατά απόλυτη τιμή) από τον τελευταίο αριθμό που έχει ταξινομηθεί (πχ. Αν ο πίνακας είναι ο A = [-3, -1, ..., 4,5] ο τελευταίος που έχει ταξινομηθεί είναι ο 4).

3. Αν αυτός είναι <u>θετικός</u>, με 2 περιστροφές τον ταξινομούμε στο τέλος (περίπτωση b).

4. Αν αυτός είναι <u>αρνητικός</u> διατρέχουμε τη λίστα μέχρι να βρούμε τον αμέσως μικρότερο που να έχει θετικό πρόσημο (πχ. Αν ο πίνακας είναι ο A = [-1, 3, -4, -2, -5, 6] ο μικρότερος θετικός μετά το -5 θα είναι ο 3).

5. Αν υπάρχει τέτοιος αριθμός, τότε σίγουρα θα έχει ζευγάρι γιατί θα υπάρχει μεγαλύτερος (κατά απόλυτη τιμή) που να είναι αρνητικός. Από τις περιπτώσεις d και e έχουμε δείξει πως μπορούμε να φτιάξουμε νέο ζεύγος σε 2 περιστροφές.

6. Αν δεν υπάρχει τέτοιος αριθμός, τότε όλοι οι υπόλοιποι είναι αρνητικοί και ψάχνουμε αν υπάρχουν 2 αριθμοί που να βρίσκονται στη περίπτωση c, όπου τότε φτιάχνουμε καινούργιο ζευγάρι με 2 περιστροφές.

7. Αν βρήκαμε ένα ζευγάρι μέσω των παραπάνω περιπτώσεων, τότε επιστρέφουμε στο **βήμα 2**

8. Αλλιώς τσεκάρουμε αν ο πίνακάς μας είναι ταξινομημένος και σε αυτή τη περίπτωση ο αλγόριθμος τερματίζει μην έχοντας ξεπεράσει τις 2n περιστροφές για τα n-1 ζεύγη.

9. Αν ο πίνακας δεν είναι ταξινομημένος τότε θα βρίσκεται σε μία από τις παρακάτω μορφές :
$A_t = [-n, -(n-1), .., -1]$
Ή
$A_t = [-1, -2, .., -n]$
Και στις δύο περιπτώσεις, οι υποπίνακες αυτοί ταξινομούνται με το πολύ 2n περιστροφές.
Στη πρώτη, έχουμε θεωρήσει ότι τα n-1 ζευγάρια έχουν δημιουργηθεί με 2(n-1) περιστγορές και τότε μπορούμε να κάνουμε άλλη μία και να ταξινομήσουμε με θετικό πρόσημο
Στη δεύτερη, έχουμε 0 ζευγάρια και για n επαναλήψεις αν κάνουμε 1 περιστροφή στο n και μία στο n-1 τότε ταξινομείται θετικά με 2n περιστροφές.
Και στις δύο περιπτώσεις, αφού ταξινομήσουμε τους υποπίνακες, επιστρέφουμε στο **βήμα 2**

Σε κανένα βήμα δεν χρειάστηκαν περισσότερες από 2k περιστροφές για k αριθμούς.
Επομένως, στο τέλος ο πίνακας θα πρέπει να έχει ταξινομηθεί σωστά με το πολύ 2n περιστροφές.

Άσκηση 3

Διαισθητικά, αν φανταστούμε την A[i] ως μία διακριτή συνάρτηση, τότε όσο εκείνη είναι φθίνουσα, τα στοιχεία αυτά της φθίνουσας ακολουθίας έχουν μία ευκαιρία να κυριαρχούν ενός μεταγενέστερου στοιχείου. Μόλις, όμως, η ακολουθία γίνει αύξουσα, όσα στοιχεία ήταν μικρότερα, δεν υπάρχει περίπτωση να κυριαρχήσουν κάποιου άλλου στο μέλλον.
Αν είχαμε λοιπόν ένα stack θα μπορούσαμε να αναπαραστήσουμε την "ευκαιρία" αυτή του κάθε στοιχείου να κυριαρχήσει κάποιου άλλου αργότερα κάνοντάς το push, και κάνοντάς το pop όταν αυτό χάνει αυτή την "ευκαιρία" να κυριαρχήσει κάποιου άλλου στη συνέχεια.

Αλγόριθμος :
Αρχικα έχουμε ένα stack με tuples (A[i], i), το οποίο περιέχει το στοιχείο (inf, 0).
Για κάθε στοιχείο A[i] :

1. Σύγκρινέ το με το πάνω στοιχείο του stack.
    a. Αν είναι μικρότερο, πάρε τη θέση του στοιχείου αυτού από το tuple και κάνε push() το καινούργιο στοιχείο και κάνε continue
    b. Αλλιώς, κάνε pop() από το stack και πήγαινε ξανά στο **βήμα 1.**

Πολυπλοκότητα :
Κάθε στοιχείο έχει 2 ευκαιρίες μόνο να "χάσει". Στη πρώτη θα είναι που θα τοποθετηθεί στο stack και στη δεύτερη που θα γίνει pop. Επομένως συνολικά θα γίνουν 2n συγκρούσεις και άρα η πολυπλοκότητα του αλγορίθμου θα είναι **Θ(n).**

Άσκηση 4

Δεδομένου ότι ο πίνακας Α είναι ταξινομημένος :

Θα δοκιμάσουμε να κάνουμε binary search στον αριθμό των φορτιστών. Ξεκινώντας από n/2 φορτιστές, σε κάθε προσπάθεια αν οι φορτιστές δεν είναι αρκετοί θα τους αυξάνουμε, και αν είναι αρκετοί θα τους μειώνουμε βρίσκοντας τελικά τον ελάχιστο αριθμό που να αρκούν για τη φόρτιση όλων των οχημάτων. Αυτό το "ψάξιμο", κοστίζει logn

Χρειαζόμαστε όμως και έναν validation αλγόριθμο για να επαληθεύσουμε αν οι φορτιστές (s*) που δοκιμάζουμε είναι αρκετοί ή όχι.

Validation αλγόριθμος :

Δημιουργούμε ένα tuple c = (s, a), όπου s η εκάστοτε χρονική στιγμή και a το πλήθος των διαθέσιμων φορτιστών εκείνη τη χρονική στιγμή.

Δίνουμε στο tuple c τη τιμή **c = (A[1], s*)**, όπου s* η τιμή που δοκιμάζουμε από το binary search.

Επίσης έχουμε και μία τιμή **max_slot = A[1]+d** η οποία αντιπροσωπεύει το αυτοκίνητο 1 μέχρι ποιό time slot μπορεί να περιμένει.

Για κάθε στοιχείο A[i] :

Αν A[i] <= c[0] έλεγξε αν c[1]=0 : # Έλεγξε αν τη στιγμή που ήρθε το αμάξι δεν υπάρχουν φορτιστές

{

Αν ναι τότε :

c[0]++, c[1]=s* # Αν δεν υπάρχουν, πήγαινε στο επόμενο time slot

Αν c[0] >= max_slot : # Αν το επόμενο time slot ξεπερνάει το max_slot κάνε return false

return false

c[1] -- # Μείωσε τους διαθέσιμους φορτιστές κατά 1

}

Αλλιώς δημιούργησε το tuple c = (A[i], s*-1)

return true

Πολυπλοκότητα :

Στο validation αλγόριθμο γίνεται γραμμική προσπέλαση του πίνακα Α οπότε για κάθε iteration του s* η χρονική πολυπλοκότητα είναι Θ(n). Επομένως, η συνολική πολυπλοκότητα του αλγορίθμου θα είναι **Θ(nlogn).**

Άσκηση 5

(a) Για να υπολογίσουμε το k-οστό μικρότερο στοιχείο του S θα κάνουμε binary search στην Fs με άκρα τα a=0, b=M. Αυτό περιμένουμε να δουλέψει γιατί η συνάρτηση Fs για όλα τα νούμερα από 1 ως M είναι ένας ταξινομημένος πίνακας ο οποίος παίρνει τιμές από 0 ως n.

Αλγόριθμος :

Αρχικοποιούμε a=0, b=M, και median m=(b-a)/2

1. Αν ισχύει a ≥ b :
    Επέστρεψε το m

2. Αλλιώς :
    m = (b-a)/2

3. Αν το k < F(m) :
    Ορίζουμε b = m-1 ; goto step 1

4. Αν το k > F(m) :
    Ορίζουμε a = m+1 ; goto step 1

5. Αν το k = F(m) :
    Επέστρεψε το m

goto step 1

Πολυπλοκότητα :

Ο αλγόριθμος δουλεύει και για πολλαπλές τιμές του ίδιου αριθμού σε ένα πίνακα. Λόγω του binary search το οποίο εκτελείται σε logn βήματα (T(n)=T(n/2)+Θ(1)), θα γίνουν **logn** κλήσεις **στην Fs.**

(b) Για αυτό το πρόβλημα μπορούμε να διατυπώσουμε δύο διαφορετικούς αλγορίθμους, όπου ο καθένας είναι αποδοτικός για διαφορετική περίπτωση χρήσης.

<u>1ος τρόπος</u> $O(n^2+M)$:

1. Φτιάχνουμε ένα πίνακα συχνότητας F[1..M-1] ο οποίος θα περιέχει το πολύ n(n-1)/2 στοιχεία και θα περιγράφει την συχνότητα του κάθε στοιχείου στο πίνακα S.
2. Για κάθε S[i] αυξάνουμε τη θέση F[S[i]]++. ($O(n^2)$).
3. Δημιουργούμε ένα πίνακα SUM[1..M-1] ο οποίος θα περιέχει σε κάθε θέση SUM[i] το άθροισμα όλων των στοιχείων μέχρι τη θέση i.
4. Για κάθε F[i] το προσθέτουμε σε ένα global sum (gsum += F[i]) και θέτουμε το SUM[i] ίσο με αυτό το sum (SUM[i]=gsum). (O(M)).

Κατά αυτό το τρόπο, έχουμε φτιάξει μία Fs(l) η οποία χρειάζεται O(1) σε κάθε κλήση της.

5. Για να βρούμε το k-οστό μικρότερο στοιχείο του S βασιζόμαστε στον αλγόριθμο του 5a ο οποίος κάνει logM κλήσεις της Fs και άρα συνολικά θα χρειάζεται μόνο O(1)*O(logM)=O(logM) χρόνο για να τρέξει.

Πολυπλοκότητα :

Η συνολική πολυπλοκότητα θα είναι :

$O(n^2 + M)$ για το preprocessing

$+ O(\log M)$ για την εύρεση ενός k

$= \mathbf{O(n^2 + M)}$

Και αυτό που έχουμε καταφέρει είναι ότι ο αλγόριθμος μπορεί να βρεί πολύ γρήγορα ένα k-οστό στοιχείο στο πίνακα S, άρα είναι καταλληλότερος για πολλά ψαξίματα.

Μπορούμε να κάνουμε κάτι καλύτερο για τη συνολική πολυπλοκότητα ?

Μπορούμε να εκμεταλλευτούμε τον πίνακα A, ώστε να μην χρειαστεί να διασχίσουμε τον S.

<u>2ος τρόπος</u> $O(n\log n\log M)$

Γενικότερη ιδέα :

Αν έχουμε τον A να είναι ταξινομημένος από το μικρότερο στοιχείο στο μεγαλύτερο, τότε μπορούμε να χρησιμοποιήσουμε την ιδιότητα του binary search για να βρούμε για κάθε στοιχείο του, από πόσα άλλα στοιχεία απέχει απόλυτη απόσταση μικρότερη ή ίση με k και στο τέλος να αθροίσουμε τα αποτελέσματα για όλα τα στοιχεία.

Αλγόριθμος :

1. Ταξινομούμε τον πίνακα A με κάποιο γρήγορο αλγόριθμο ταξινόμησης (έστω quicksort) σε χρόνο $O(n\log n)$ - preprocessing.

2. (Για να βρούμε το k-οστό μικρότερο στοιχείο)

Για κάθε στοιχείο του A (έστω i) :

Κάνουμε binary search στο k, κάθε φορά με αρχή του υποπίνακα το A[i] και τέλος του το A[n]. Για κάθε σύγκριση του k με ένα A[j] δεν το συγκρίνουμε με το A[j] αλλά με το A[j]-A[i]. Αυτό δουλεύει γιατί, όντας ταξινομημένος ο πίνακας, θα είναι ταξινομημένες και οι διαφορές A[j]-A[i], $i \neq j$ & $i < j$.

3. Αθροίζοντας τα αποτελέσματα από τα binary searches για κάθε A[i], έχουμε φτιάξει μία Fs η οποία για κάθε κλήση της χρειάζεται χρόνο $O(n\log n)$.

4. Εφαρμόζοντας και πάλι τον αλγόριθμο του 5α για την εύρεση του k-οστού μικρότερου στοιχείου στο πίνακα S, εκτελούμε $\log M$ κλήσεις της Fs και άρα η πολυπλοκότητα για κάθε ψάξιμο θα είναι $O(n\log n)*O(\log M)=O(n\log n\log M)$.

Πολυπλοκότητα :

Η συνολική πολυπλοκότητα θα είναι :

$O(n\log n)$ για το preprocessing

$+ O(n\log n\log M)$ για την εύρεση ενός k

$= \mathbf{O(n\log n\log M)}$

Η διαφορά με τον 1ο αλγόριθμο είναι ότι έχουμε μοιράσει κατά κάποιο τρόπο την χρονική πολυπλοκότητα στο preprocessing και στις κλήσεις του Fs και για μοναδικό ψάξιμο ενός k-οστού στοιχείου, ο 2ος αλγόριθμος θα είναι πιο γρήγορος.