Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Αλγόριθμοι και Πολυπλοκότητα

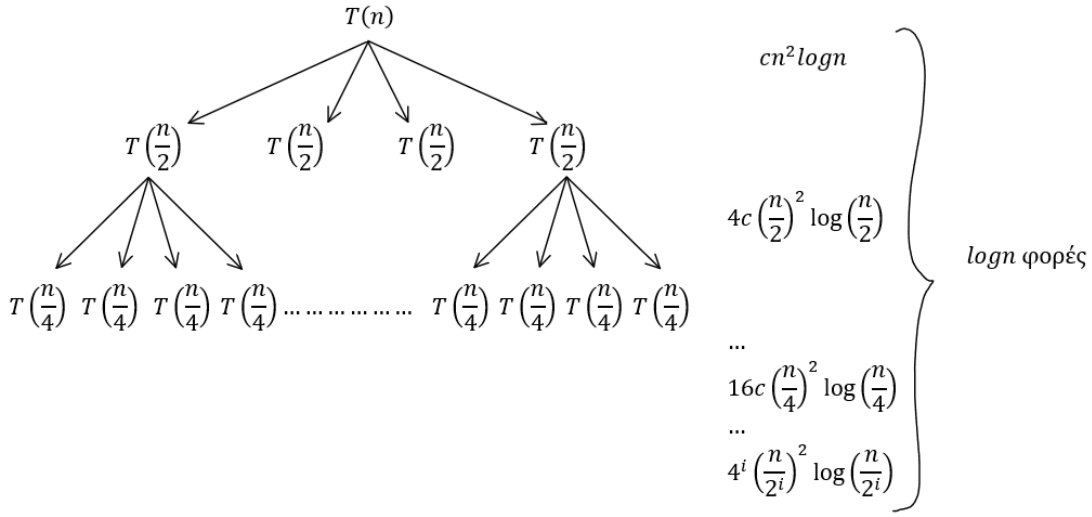
1η Σειρά Γραπτών Ασκήσεων – Χειμερινό Εξάμηνο 2021

Κονταλέξη Μαρίνα – el18022

Άσκηση 1:

1. $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 logn)$

   Έχουμε το παρακάτω δέντρο αναδρομής:



   Άρα το υπολογιστικό κόστος από τα επίπεδα είναι

$$c\sum_{i=0}^{logn} 4^i\left(\frac{n}{2^i}\right)^2 \log\left(\frac{n}{2^i}\right) = cn^2\sum_{i=0}^{logn}\log\left(\frac{n}{2^i}\right) = cn^2\left(\sum_{i=0}^{logn}\log(n) - \sum_{i=0}^{logn}\log(2^i)\right)$$
$$= cn^2\left(\log^2 n - \sum_{i=0}^{logn} i\right) = cn^2\left(\log^2 n - \frac{logn(logn+1)}{2}\right)$$
$$= cn^2\log^2 n - n^2\log n$$

   Ενώ από τα φύλλα είναι $n^{\log_2 4} = n^2$

   Προφανώς $T(n) = \Theta(n^2\log^2 n)$

2. $T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2 logn)$

   Αντίστοιχα με πριν έχουμε την παρακάτω πολυπλοκότητα από τα συνολικά επίπεδα του δέντρου.

$$c\sum_{i=0}^{logn} 5^i\left(\frac{n}{2^i}\right)^2 \log\left(\frac{n}{2^i}\right) = cn^2\sum_{i=0}^{logn}\left(\frac{5}{4}\right)^i \log\left(\frac{n}{2^i}\right)$$
$$= cn^2\left(logn\sum_{i=0}^{logn}\left(\frac{5}{4}\right)^i - \sum_{i=0}^{logn} i\left(\frac{5}{4}\right)^i\right)$$

   Θα υπολογίσουμε ξεχωριστά τα αθροίσματα:

$$\sum_{i=0}^{logn}\left(\frac{5}{4}\right)^i = \frac{\left(\frac{5}{4}\right)^{logn+1} - 1}{\frac{5}{4} - 1} = 4\left(2^{\log\left(\frac{5}{4}\right)(logn+1)} - 1\right) = 4\left((2n)^{\log\left(\frac{5}{4}\right)} - 1\right)$$
$$\approx 5n^{0,32} - 4$$

   και

$$\sum_{i=0}^{logn} i\left(\frac{5}{4}\right)^i = \frac{5}{4}\sum_{i=0}^{logn} i\left(\frac{5}{4}\right)^{i-1} = \frac{5}{4}\left(\sum_{i=0}^{logn}\left(\frac{5}{4}\right)^i\right)' = \frac{5}{4}\frac{d\left(\sum_{i=0}^{logn} a^i\right)}{da}_{a=\frac{5}{4}}$$

$$= \frac{5}{4}\frac{d\left(\frac{a^{logn+1}-1}{a-1}\right)}{da}_{a=\frac{5}{4}} = \frac{5}{4}\frac{logn\, a^{logn+1} - (logn+1)a^{logn} - 1}{(a-1)^2}_{a=\frac{5}{4}}$$

$$\approx 20\left(logn * \frac{5}{4}n^{0,32} - (logn+1)n^{0,32} - 1\right)$$

$$= (5logn - 20)n^{0,32} - 20$$

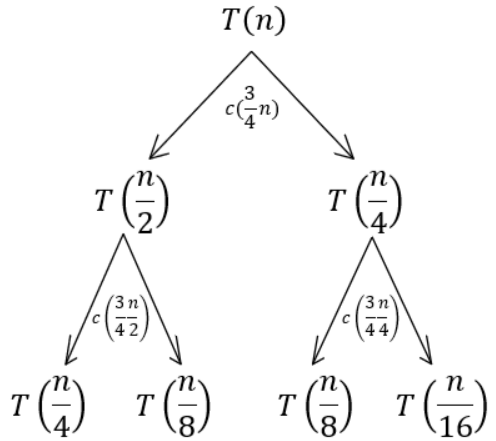
Τελικά η συνολική πολυπλοκότητα είναι η παραπάνω αυξημένη κατά το κόστος των φύλλων.

$$T(n) = cn^2(\cancel{5\, n^{0,32} logn} - 4logn \cancel{- 5n^{0,32} logn} + 20n^{0,32} - 20) + n^{\log_2 5} \Rightarrow$$

$$T(n) = cn^{2,32} - \frac{c}{5}n^2 logn - cn^2 + n^{2,32} = \Theta(n^{2,32})$$

3. $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$

Έχουμε το παρακάτω δέντρο αναδρομής:



$$c\frac{3}{4}\frac{n}{2} + c\frac{3}{4}\frac{n}{4} = c\frac{3}{4}\left(\frac{n}{2} + \frac{n}{4}\right) = c\left(\frac{3}{4}\right)^2 n$$

Το οποίο γενικεύεται σε $c\left(\frac{3}{4}\right)^i n$ για κάθε ένα από τα πρώτα $\log_4 n$ επίπεδα. Όσο αυξάνονται τα επίπεδα μειώνεται το υπολογιστικό κόστος, εφόσον το δέντρο δεν είναι πλήρες. Σε περίπτωση που ήταν θα είχε $n^{log2} = n$ φύλλα.

Άρα

$$T(n) \leq cn\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + n^{\log_2 2} = 4cn + n = bn$$

Όμως αφού Θ(n) = n, προφανώς $n \leq T(n)$.
Άρα τελικά, $T(n) = \Theta(n)$

4. $T(n) = 2T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$

Συγκρίνοντας με το δέντρο του προηγούμενου ερωτήματος, πλέον σε κάθε επίπεδο αναδρομής έχουμε $c\frac{3}{4}n = bn$ υπολογιστικό κόστος.

Το δέντρο της αναδρομής είναι πλήρες μέχρι το ύψος $\log_4 n = \frac{logn}{2}$ και έχει συνολικό ύψος $logn$.

Επίσης, τα φύλλα του δέντρου είναι n. (σε κάθε διαίρεση έχουμε $2\frac{n'}{4} + \frac{n'}{2} = n'$)

Επομένως έχουμε ότι $bn * \frac{logn}{2} + n \leq T(n) \leq bnlogn + n \Rightarrow T(n) = \Theta(nlogn)$

5. $T(n) = T\left(n^{\frac{1}{2}}\right) + \Theta(logn)$

   Έχουμε ότι

$$T(n) \leq \sum_{i=0}^{\infty} \log\left(n^{\left(\frac{1}{2}\right)^i}\right) + T(1) = logn \sum_{i=0}^{\infty} \frac{1}{2^i}$$

   Όπου $\sum_0^{\infty} \frac{1}{2^i} = 2$

   Άρα $T(n) \leq clogn$ και $T(n) \geq logn$ ($\alpha\varphi o\acute{\upsilon}\ \Theta(n) = logn$).

   Επομένως, $T(n) = \Theta(logn)$.

6. $T(n) = T\left(\frac{n}{4}\right) + \Theta\left(\sqrt{n}\right)$

   Έχουμε ότι:

$$\frac{f(n)}{n^{\log_b a}} = \frac{\sqrt{n}}{n^{\log_4 1}} = \frac{\sqrt{n}}{1} = n^{\frac{1}{2}}$$

   Δηλαδή, $\exists \varepsilon > 0: f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$

   Άρα, από Master Theorem $T(n) = \Theta\left(\sqrt{n}\right)$

Άσκηση 2:

(α) Εάν έχουμε n = 3, τότε εξετάζοντας όλες τις μεταθέσεις έχουμε ότι ο μέγιστος αριθμός περιστροφών που χρειαζόμαστε είναι 3.
Θα δείξουμε με επαγωγή ότι $\forall n > 3: r(n) = r(n-1) + 2$, όπου $r(n)$ = μέγιστος αριθμός περιστροφών.
Βάση επαγωγής:
Για n = 4, χρειαζόμαστε 2 περιστροφές για να μεταφέρουμε τον αριθμό 4 στην τέταρτη θέση (μία μέχρι τον εαυτό του και μία ολόκληρο τον πίνακα) και άλλες r(3) = 3 περιστροφές για να διατάξουμε τα υπόλοιπα 3 στοιχεία.
Έστω ότι η υπόθεση ισχύει για n. Θα δείξουμε ότι ισχύει για n+1.
Πράγματι, για n+1 αριθμούς χρειαζόμαστε 2 περιστροφές για τη μετακίνηση του μέγιστου στην τελευταία θέση και r(n) περιστροφές για να λύσουμε το υποπρόβλημα n στοιχείων.
Άρα έχουμε την αναδρομική σχέση $r(n) = r(n-1) + 2$ με αρχική συνθήκη $r(3) = 3$.
Λύνοντας την αριθμητική πρόοδο έχουμε ότι

$$r(n) = 2 * (n-3) + r(3) = 2n - 6 + 3 = 2n - 3.$$

(β) Ο αλγόριθμος θα κάνει n επαναλήψεις. Σε κάθε επανάληψη i τα βήματα του αλγορίθμου είναι τα εξής:

1) Βρίσκουμε το μεγαλύτερο σε απόλυτη τιμή στοιχείο m που δεν έχουμε ακόμα ταξινομήσει. Προφανώς |m| = n – i.
2) Κάνουμε μία προσημασμένη προθεματική περιστροφή στα στοιχεία μέχρι και το m.
3) Εάν το m μετά την περιστροφή είναι αρνητικό αγνοούμε το βήμα 4.
4) Περιστρέφουμε το m με τον εαυτό του.
5) Κάνουμε μία προσημασμένη προθεματική περιστροφή στα n – i πρώτα στοιχεία.

Στο τέλος κάθε επανάληψης, το στοιχείο n-i βρίσκεται με σωστό πρόσημο στη θέση n-i.

Για κάθε βήμα έχουμε το πολύ 3 περιστροφές. Επομένως, οι συνολικές περιστροφές είναι 3n.

(γ)

1. Θα διακρίνουμε περιπτώσεις:
   - Όλα τα στοιχεία του πίνακα είναι θετικά.
     Τότε, έστω ότι βρίσκουμε το μεγαλύτερο στοιχείο του πίνακα στη θέση j. Κάνουμε μία περιστροφή μέχρι και τη θέση αυτού και κατόπιν άλλη μία ολόκληρο τον πίνακα. Πλέον, το μέγιστο στοιχείο βρίσκεται με θετικό (και πάλι) πρόσημο στο τέλος του πίνακα επομένως κατά σύμβαση θεωρείται συμβατό ζεύγος.
   - Όλα τα στοιχεία του πίνακα είναι αρνητικά.
     Τότε υπάρχει τουλάχιστον ένα ζεύγος A[i] = x, A[j] = x+1 τέτοιο ώστε i+1<j. Δηλαδή ο πίνακας είναι της μορφής […, x, …, x+1, …]. [1]
     Κάνουμε περιστροφή μέχρι και τη θέση j επομένως ο πίνακας είναι της μορφής [-x, …., x+1, …] και άλλη μία περιστροφή μέχρι και τη θέση j-1. Άρα ο τελικός πίνακας είναι της μορφής […, x, x+1, ….] και διαθέτει ένα τουλάχιστον συμβατό ζεύγος.

[1] Σε περίπτωση που δεν υπάρχει τέτοιο ζεύγος ο πίνακας είτε είναι της μορφής [-1, -2, … -n] και δεν τον εξετάζουμε ως ενδιάμεσο είτε είναι ήδη ταξινομημένος και η εύρεση συμβατού ζεύγους είναι τετριμμένη.

- Υπάρχουν και αρνητικά και θετικά στοιχεία στον πίνακα.
  Τότε $\exists x: A_t[i] = x \; και \; A_t[j] = -(x+1)$. Δηλαδή υπάρχει ένα τουλάχιστον στοιχείο που εμφανίζεται με αντίθετο πρόσημο από το επόμενό του σε διάταξη.
  Εάν x > 0 (εναλλαγή από θετικό σε αρνητικό):
  Χωρίς βλάβη της γενικότητας θεωρούμε ότι i < j.
  Κάνουμε μία περιστροφή μέχρι τη θέση j, επομένως έχουμε τον πίνακα:
  [(x+1), …, -x, …]. Κατόπιν, κάνουμε μία περιστροφή μέχρι πριν το στοιχείο -x οπότε καταλήγουμε με τον πίνακα […, -(x+1), -x, …]. Ο πίνακας περιέχει ένα συμβατό ζεύγος. Όμοια για i > j.

  Εάν x < 0 (εναλλαγή από αρνητικό σε θετικό):
  Τότε εξετάζουμε τους επόμενους διαδοχικούς αριθμούς (σε απόλυτη τιμή) που εμφανίζονται με διαφορετικό πρόσημο, οι οποίοι θα είναι στην κατηγορία της εναλλαγής από θετικό σε αρνητικό και ακολουθούμε τα παραπάνω βήματα.
  Εάν δεν υπάρχει ζεύγος μεγαλύτερο από το (x, x+1) με πρόσημα που εναλλάσσονται, τότε $\forall y > x: A_t[y] > 0 \Longrightarrow$ το μέγιστο στοιχείο του A_t είναι θετικό $\Longrightarrow$ κάνουμε τα βήματα της πρώτης περίπτωσης.

Επομένως, αποδείξαμε ότι σε κάθε περίπτωση μπορούμε να δημιουργήσουμε ένα συμβατό ζεύγος.

2. Θα ταξινομήσουμε τον πίνακα δημιουργώντας αναδρομικά n συμβατά ζεύγη. Τα βήματα του αλγορίθμου φαίνονται παρακάτω.
   1) Δημιουργούμε ένα συμβατό ζεύγος μεταξύ των αριθμών x, y. Ο αριθμός y είναι ο μικρότερος αριθμός στον πίνακα τέτοιος ώστε x<y.
   2) Αντικαθιστούμε το ζεύγος που φτιάξαμε με τον αριθμό x. Εάν ο πίνακας περιλαμβάνει μόνο ένα στοιχειό παραλείπουμε το βήμα 3.
   3) Επαναλαμβάνουμε το βήμα 1 στον πίνακα που προέκυψε.
   4) Εάν το εναπομείναν στοιχείο του πίνακα είναι αρνητικό κάνουμε άλλη μία περιστροφή. Αλλιώς, ο πίνακας έχει ταξινομηθεί.

   Σχετικά με την ορθότητα του αλγορίθμου έχουμε:
   Κάθε νέο ζεύγος στοιχείων το αντιμετωπίζουμε σαν ένα νέο στοιχείο, ώστε τα εσωτερικά στοιχεία του ζεύγους να μη χάσουν τη μεταξύ τους διάταξη. Κάθε φορά που προκύπτει ένας νέος πίνακας μπορούμε βάσει του ερωτήματος γ1 να δημιουργήσουμε ένα νέο συμβατό ζεύγος, μέχρι τη στιγμή που ο πίνακας είναι πλήρως ταξινομημένος.[2]
   Ως προς τον αριθμό των απαιτούμενων περιστροφών, κάθε φορά που δημιουργούμε ένα ζεύγος το μέγεθος του πίνακα μειώνεται κατά 1. Επομένως, μέχρι το μέγεθος του πίνακα να γίνει 1 απαιτούνται n-1 δημιουργίες συμβατών ζευγών, αποτελούμενες από 2 το πολύ περιστροφές η καθεμία και 1 το πολύ ακόμη περιστροφή από το βήμα 4. Άρα συνολικά απαιτούνται το πολύ 2n περιστροφές.

[2] Εύκολα αποδεικνύουμε ότι εάν ο πίνακας είναι της μορφής [-1 -2 … -n] μπορούμε να τον ταξινομήσουμε κάνοντας 2n περιστροφές. Αρχίζοντας από μία περιστροφή όλων των στοιχείων και συνεχίζοντας με περιστροφή των πρώτων n-1 στοιχείων. Επαναλαμβάνουμε τη διαδικασία n φορές και ο πίνακας έχει ταξινομηθεί.

Άσκηση 3:

Στο εξής θα συμβολίζουμε M[i] τη θέση που κυριαρχεί της θέσης i.

Ο αλγόριθμος φαίνεται παρακάτω σε ψευδοκώδικα:

```
for i in range (1, n)
    if A[i] < A[i-1] then M[i] = i-1
    else {
        current = M[i-1]
        while A[i] >= A[current]:
            current = M[current]
        M[i] = current
    }
```

Αιτιολόγηση ορθότητας

1. Αν A[i] < A[i-1], δηλαδή εάν το στοιχείο στη θέση i είναι μικρότερο από το αμέσως προηγούμενο από αυτό, τότε (με τετριμμένο τρόπο) M[i] = i - 1, καθώς είναι σίγουρα η μέγιστη θέση στο διάστημα [0,i) και επιπλέον ικανοποιεί τη συνθήκη A[i-1] > A[i].
2. Αλλιώς, δηλαδή αν A[i] ≥ A[i-1], τότε ο αλγόριθμος **«προσπερνά» όλα τα στοιχεία στο διάστημα (M[i-1], i)** και συγκρίνει κατευθείαν το στοιχείο που εξετάζεται (A[i]) με το στοιχείο της θέσης που κυριαρχεί της θέσης i-1 (A[current] = A[M[i-1]]).

   Θα αποδείξουμε σύντομα γιατί $M[i] \notin (M[i-1], i-1)$.

   Έστω ότι M[i] = j. Τότε $\exists\, j \in (M[i-1], i-1) : A[j] > A[i]$

   $$\xRightarrow[A[i] \geq A[i-1]]{} A[j] > A[i] \geq A[i-1] \Longrightarrow A[j] > A[i-1]$$

   Όμως, από εκφώνηση η θέση που κυριαρχεί της θέσης i-1 είναι η μεγαλύτερη θέση μέχρι τη θέση i-1 με στοιχείο μεγαλύτερο του A[i]. Δηλαδή:

   $$M[i-1] < j \Longrightarrow \forall k \in (M[i-1], i-1): A[k] \leq A[i-1]$$

   Άτοπο. Άρα, αποδείξαμε πως το M[i] δεν ανήκει στο διάστημα αυτό και καλώς ο αλγόριθμος δεν ελέγχει κάθε επιμέρους θέση.

   Εάν και πάλι η σύγκριση αποτύχει ο αλγόριθμος, συνεχίζει συγκρίνοντας το στοιχείο με εκείνο της θέσης που κυριαρχεί της θέσης που μόλις έλεγξε. (βλ. εντολή: current = M[current]) Η απόδειξη του γιατί και πάλι δεν παραβιάζεται η ορθότητα του αλγορίθμου όταν προσπερνά το αντίστοιχο διάστημα είναι όμοια με την παραπάνω και στηρίζεται στο ότι και πάλι το key που εξετάζουμε είναι μεγαλύτερο ή ίσο του στοιχείου της θέσης που ελέγχουμε και άρα μπορούμε να αγνοήσουμε όλα τα μικρότερα από αυτό στοιχεία.

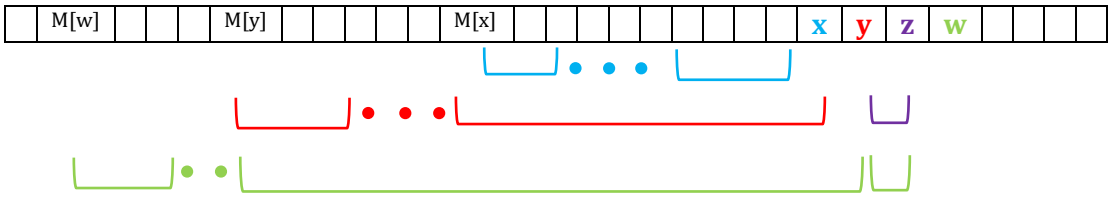
Είναι βέβαιο πως το while loop θα τερματίσει, καθώς όσο ο αλγόριθμος διατρέχει τον πίνακα προς τα πίσω για να βρει στοιχείο μεγαλύτερο από το key, σίγουρα κάποια στιγμή θα φτάσει στο 0 όπου θα ισχύει A[i] < A[0] = ∞

Υπολογιστική πολυπλοκότητα

Η πολυπλοκότητα του αλγορίθμου είναι O(n) και αυτό οφείλεται στο ότι κάθε φορά αγνοεί όλα τα εσωτερικά στοιχεία ενός διαστήματος (M[i-1], i-1) επειδή αποδείξαμε ότι αποκλείεται κάποιο από αυτά να είναι το M[i].

Θα εστιάσουμε στην πολυπλοκότητα του αλγορίθμου όταν χρειάζεται να αναζητήσει το M[i] σε θέσεις πριν την i-1.

Παρακάτω φαίνονται τα μονοπάτια που ακολουθεί ο αλγόριθμος για 4 τυχαία στοιχεία x < y < w και z < y χρησιμοποιώντας μη ορισμένο αριθμό βημάτων για το καθένα.



Οι τελείες χρησιμοποιούνται για να δείξουν ότι δεν μας ενδιαφέρει με πόσα βήματα φτάσαμε στο M[i], αλλά σε ποιο σημείο του πίνακα φτάσαμε.

Παρατηρούμε ότι όσα διαστήματα του πίνακα εξετάστηκαν από ένα στοιχείο i, **δεν θα προσπελαστούν δεύτερη φορά** από τον αλγόριθμό μας χάρη στο άλμα που κάνει.

Επομένως, ο αλγόριθμος θα προσπελάσει ολόκληρο τον πίνακα μία φορά για να υπολογίσει κάθε M[i] και κάθε στοιχείο το πολύ άλλη μία φορά όσο εκτελείται προς τα πίσω.

Άρα, η συνολική υπολογιστική πολυπλοκότητά του είναι O(n).

Άσκηση 4:

Για να λύσουμε το πρόβλημα χρειάζεται να παρατηρήσουμε πως ο μέγιστος αριθμός φορτιστών που χρειάζεται είναι n, καθώς θα ήταν εφικτό κάθε αυτοκίνητο να φορτιστεί σε μία ακριβώς μέρα από τη μέρα προσέλευσής του, ο οποίος είναι ο μικρότερος δυνατός χρόνος.

Ο αλγόριθμος φαίνεται παρακάτω.

```
start = 1
end = n
while (start <= end)
        mid = (start + end) div 2
        chargers = s[mid]
        if solved(chargers) then end = mid
        else start = mid + 1
return s[mid]


solved(chargers):
        current = 1                 // μέρα που το πιο πρόσφατο αμάξι θα είναι έτοιμο
        c = chargers                // διαθέσιμοι φορτιστές
        for i in range(n)
                if c == 0 or i >= current:    // εάν δεν υπάρχουν διαθέσιμοι φορτιστές ή
                        current = i + 1       // το επόμενο αμάξι έρθει αφού φορτιστούν
                        c = chargers          // τα προηγούμενα
                else if (i < current):        // εάν έρθει αμάξι όσο φορτίζουν άλλα, όμως
                        c -= 1                // υπάρχει κι άλλος διαθέσιμος φορτιστής
                if (i – current > d) then return false   // αν κάποιο αμάξι περιμένει
                                                         // περισσότερη ώρα τότε δεν φτάνουν οι
                                                         // φορτιστές
        return true
```

Η συνάρτηση ελέγχει εάν ο δοσμένος αριθμός φορτιστών επαρκεί για να λύσουμε το πρόβλημα. Κάθε φορά αναθέτει τους διαθέσιμους φορτιστές στα πιο πρόσφατα αφιχθέντα αμάξια. Αυτό είναι πάντα η βέλτιστη λύση καθώς δεν κερδίζουμε κάτι όταν ένας φορτιστής μένει ανεκμετάλλευτος, εφόσον είτε θα χρησιμοποιηθεί την ίδια μέρα από επόμενο αυτοκίνητο (ισοδύναμο) είτε θα ελευθερωθεί την ακριβώς επόμενη.

Άρα, η συνολική πολυπλοκότητα του αλγορίθμου είναι O(n) για κάθε μία από τις logn επαναλήψεις της binary search. Συνολικά O(nlogn).

Άσκηση 5:

(α) Για να λυθεί το πρόβλημα χρειάζεται να παρατηρήσουμε ότι η κατανομή Fs είναι μία αύξουσα συνάρτηση εφόσον ισχύει ότι $F_s(x) = F_s(x-1) + N_x$, όπου N_x είναι το πλήθος των στοιχείων του S που έχουν τιμή x.
Οπότε μας ενδιαφέρει πότε (δηλ. για ποια τιμή x) το πλήθος που καταγράφει η κατανομή Fs φτάνει ή ξεπερνά την τιμή k.
Θα χωρίσουμε το πρόβλημα σε δύο περιπτώσεις:

1. $\exists x: F_s(x) = k$
   Θεωρούμε x_0 την **μικρότερη** τιμή για την οποία ισχύει η ισότητα.
   Άρα $F_s(x_0 - 1) = k - N_{x_0}$, όπου $N_{x_0} \neq 0$. Προσθέτοντας τα N_{x_0} στοιχεία, στην κατανομή, η F_s έχει πλέον διατρέξει ακριβώς k στοιχεία και τα N_{x_0} τελευταία είναι ίσα με x_0, άρα και το k-οστό είναι ίσο με x_0.
   Αρκεί να βρούμε την τιμή x_0.
2. $\nexists\, x: F_s(x) = k$
   Θεωρούμε x_0 την **μικρότερη** τιμή για την οποία $F_s(x_0) > k$.
   Προσθέτοντας τα N_{x_0} στοιχεία, στην κατανομή, η F_s έχει πλέον διατρέξει περισσότερα από k στοιχεία και τα N_{x_0} τελευταία είναι ίσα με x_0. Προφανώς, το k-οστό στοιχείο βρίσκεται μεταξύ των N_{x_0} τελευταίων και είναι ίσο με x_0, καθώς το πλήθος των στοιχείων που είχε μετρήσει η κατανομή πριν από αυτά ήταν μικρότερο από k.
   Και πάλι, αρκεί να βρούμε την τιμή x_0.

Το πρόβλημα μπορεί να λυθεί αποδοτικά με εξαντλητικό binary search στις τιμές της Fs. Χρησιμοποιούμε τη λέξη εξαντλητικό επειδή δεν αρκεί να βρούμε τον αριθμό k -αν υπάρχει- στο σύνολο τιμών της Fs, αλλά χρειαζόμαστε την πρώτη εμφάνιση αυτού. Για αυτό τροποποιούμε το binary search ώστε να ολοκληρώνεται πάντα όταν οι δείκτες start, end δείχνουν στο ίδιο σημείο ακόμα και αν το κλειδί k υπάρχει στο σύνολο τιμών της F_s. Ο κώδικας του binary search φαίνεται παρακάτω.

```
start = 0

end = M

while (true)

    mid = (start + end) div 2

    if (end – start == 0) answer = mid

    else if (Fs(mid) >= k): end = mid

    else start = mid+1
```

Στην περίπτωση που η Fs παίρνει την τιμή k, μόλις ο αλγόριθμος το βρει θα συνεχίσει να ψάχνει λογαριθμικά «κάτω» από αυτό και κάθε φορά που θα το συναντάει σε μικρότερη θέση θα κρατάει αυτή σαν end. Στο τέλος:

3. Είτε θα έχει να συγκρίνει δύο γειτονικές θέσεις της μορφής $\left(F_s(x_1), F_s(x_0)\right)$ με $F_s(x_1) < k$. Τότε,

$$start = mid = x_1 \Longrightarrow F_s(mid) < k \Longrightarrow start_{new} = mid + 1 = x_0$$

$$\xRightarrow[\text{επόμενη επανάληψη}]{} mid = end = start_{new} \Rightarrow ans = start_{new} = x_0$$

4. Είτε θα έχει να συγκρίνει δύο γειτονικές θέσεις της μορφής $\left(F_s(x_1), F_s(x_2)\right)$ με $F_s(x_1) = F_s(x_2) = k$. Τότε,

$$start = mid = x_1 \Longrightarrow F_s(mid) == k \Longrightarrow end_{new} = mid = x_1$$

$$\xRightarrow[επόμενη\ επανάληψη]{} mid = start = end_{new} \Rightarrow ans = end_{new} = x_1$$

Προφανώς το x_1 είναι η θέση x_0 που ψάχνουμε καθώς εάν υπήρχε θέση y = x_1-1: $F_s(y) = k$. Θα το αποδείξουμε σύντομα με απαγωγή σε άτοπο.

Έστω ότι υπάρχει το y του παραπάνω ισχυρισμού.

Εφόσον, η τιμή start = $x_1 \neq 0$ (έχει προηγούμενο στοιχείο) ⇒ $x_1 = mid + 1$, όπου mid είναι η τιμή της μεταβλητής mid της προηγούμενης επανάληψης ⇒ $y = mid$.

Όμως για να φτάσει η ροή ελέγχου του αλγορίθμου στο σημείο να αλλάξει η τιμή της μεταβλητής start πρέπει $F_s(mid) < k \Leftrightarrow F_s(y) < k$ *άτοπο*.
Άρα και πάλι ο αλγόριθμος υπολογίζει σωστά το x_0.

Τέλος, στην περίπτωση που η Fs δεν παίρνει την τιμή k, ο αλγόριθμος και πάλι υπολογίζει την πρώτη θέση που η συνάρτηση παίρνει τιμή μεγαλύτερη του k, και η απόδειξη είναι όμοια με αυτή της πρώτης περίπτωσης παραπάνω.

Η υπολογιστική πολυπλοκότητα του αλγορίθμου είναι εύκολο να υπολογιστεί, εφόσον πρόκειται για ένα binary search πάνω σε δεδομένα μεγέθους M. Επομένως, έχουμε πολυπλοκότητα καλύτερης, μέσης και χειρότερης περίπτωσης (αφού πάντοτε εξαντλεί όλα τα διαστήματα μέχρι να φτάσει σε γειτονικά στοιχεία) O(logM).

Οι απαιτούμενες κλήσεις της F_s στην χειρότερη περίπτωση είναι logM.

(β) Ο παρακάτω αλγόριθμος υπολογίζει αποδοτικά το F_s(k).

```
A' = sorted(A)
for i in range (n):
    j = binary_search(A'[i] + k) in A'[i, … n]      // min(j): A'[j] >= A'[i] + k
    answer += j – i – 1
```

Συνοπτικά:

1. Ταξινομούμε τον πίνακα A ώστε να μπορούμε να κάνουμε binary search
2. Για κάθε θέση i βρίσκουμε τη θέση του πίνακα j κάτω από την οποία κάθε διαφορά A[p] – A[i] <= k, $p \in [i, j)$, ψάχνοντας με binary search την μικρότερη θέση j για την οποία A'[j] >= A'[i] + k (ίδιο binary search με το προηγούμενο ερώτημα, μόνο που τώρα μπορεί να τερματίσει και στην πρώτη εμφάνιση του στοιχείου A'[i] + k).
3. Προσθέτουμε όλες τις παραπάνω διαφορές πλήθους (j – i – 1) (μία για κάθε ενδιάμεσο στοιχείο).

Τελικά θα έχουμε μετρήσει τις ζητούμενες διαφορές με αφετηρία κάθε στοιχείο του A, άρα θα έχουμε λύσει το ζητούμενο.

Η πολυπλοκότητα υπολογισμού της τιμής $F_s(k)$ είναι

$O(max(log(n), log(n-1), \dots log(1)) = O(logn)$ για το binary search σε κάθε επανάληψη, και το πλήθος των επαναλήψεων είναι n.
Επομένως, συνολικά έχουμε πολυπλοκότητα της τάξης του O(nlogn)
Εφόσον έχουμε υπολογίσει αποδοτικά την τιμή $F_s(k)$, για την εύρεση του k-οστού στοιχείου θα χρησιμοποιήσουμε τον αλγόριθμο του ερωτήματος (α).
Η συνολική πολυπλοκότητα του αλγορίθμου μας είναι ίση με τις συνολικές κλήσεις της συνάρτησης F_s επί την πολυπλοκότητα κάθε κλήσης της.
Άρα $O(nlogn * logn) = O(n \log^2 n)$