ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ

ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αλγόριθμοι και Πολυπλοκότητα

1η Σειρά Γραπτών Ασκήσεων

ΔΗΜΗΤΡΙΟΣ ΓΕΩΡΓΟΥΣΗΣ, 03119005

Ο ψευδοκώδικας είναι γραμμένος στα περισσότερα σημεία με τρόπο που θυμίζει python. Το range(A,B) θεωρούμε ότι συμπεριλαμβάνει το Α αλλά όχι το Β.

Άσκηση 1: Πλησιέστερο Ζεύγος Σημείων

(α)

```
Αλγόριθμος:

# Έστω ότι ο χώρος έχει 3 διαστάσεις
# x, y, z
S # S είναι η λίστα σημείων που έχουμε
  # τα οποία είναι τριάδες της μορφής
  # (x,y,z), δηλαδή, S[i]=(x,y,z) με
  # 1 <= i <= n

closest_pair(P):
    if (|P| == 1): return +inf
    Z = median(P,3) # θεωρούμε ότι median
    # είναι η κλασσική median του μαθήματος
    # που βρίσκει το μεσαίο στοιχείο του συνόλου
    # αν αυτό ήταν ταξινομημένο και βάζει πριν από
    # αυτό στοιχεία που είναι μικρότερα ή ίσα των
    # στοιχείων που βρίσκονται μετά από αυτό.
    # Εδώ η δεύτερη παράμετρος της median αντιπροσωπεύει
    # ως προς ποιο στοιχείο της τριπλέτας να γίνει αυτή η
    # "ταξινόμηση".Δηλαδή, διάσταση z -->3, y -->2, x -->1
    # Το Z που επιστρέφει είναι ο δείκτης στο median.
    Z_coord = P[Z].z

    d1 = closest_pair(P[:(Z+1)]) # αναδρομή στο κάτω μισό
    d2 = closest_pair(P[(Z+1):]) # αναδρομή στο πάνω μισό
    # υποθέτουμε ότι γίνεται πέρασμα κατ' αναφορά
    d = min(d1,d2) # κρατάμε το ελάχιστο των δύο αποστάσεων

    for el in P:
        if (abs(el.z - Z_coord) > d):
            P.pop(el) # αφαιρούμε το el από το σύνολο σημείων
    # θέλουμε πέρασμα κατ' αναφορά ώστε οι αφαιρέσεις των
    # αναδρομικών κλήσεων να επηρεάζουν και το σύνολο S της δικής
    # μας κλήσης
    d3 = helper(P,d,Z_coord)
    d = min(d,d3)
    return d

helper(P,dist,Z_coord): # εδώ συμπεριφερόμαστε στα σημεία σαν
# να τα έχουμε προβάλει στο z = Z επίπεδο, αφού η προβολή
# αυτή δεν πρόκειται να αυξήσει τις μεταξύ τους αποστάσεις.
    if (|P| == 1): return +inf
    X = median(P,1)
    X_coord = P[X].x
    d1 = helper(P[:(X+1)]) # αναδρομή στο αριστερό μισό
    d2 = helper(P[(X+1):]) # αναδρομή στο δεξί μισό
    # πάλι υποθέτουμε πέρασμα κατ' αναφορά
    merge(P[:(X+1)],P[(X+1):],2) # η merge αυτή είναι η merge
    # που είδαμε στο μάθημα, η οποία ταξινομεί δύο ήδη
    # ταξινομημένες λίστες και στην τρίτη παράμετρο δίνουμε
    # ως προς ποιο στοιχείο των σημείων να γίνει η ταξινόμηση.
    # Εδώ y -->2. Και εδώ θέλουμε τα περάσματα κατ' αναφορά γιατί
    # θέλουμε η merge να επηρεάζει την ταξινόμηση στην αρχική μας
```

```
53          # λίστα P ώστε να λειτουργεί σωστά στις αναδρομές.
54          d = min(d1,d2,dist) # αν το dist το οποίο βρήκαμε από την
55          # closest_pair είναι μικρότερο από τα d1,d2 μας νοιάζουν μόνο
56          # αποστάσεις μικρότερες από αυτό
57          for el in P:
58              if((abs(el.x - X_coord) > d) or (abs(el.z - Z_coord) > d)):
59                  P.pop(el) # αφαιρούμε το el από το σύνολο σημείων
60          for el in P: # P είναι ταξινομημένη λίστα ως προς y στο σημείο αυτό
61              n_el = el.next()
62              for i in range(0,48): # σύγκρινε με τους επόμενους 48
63                  if (n_el doesn't exist):
64                      break
65                  new_d = distance(el,n_el)
66                  d = min(d,new_d)
67                  n_el = n_el.next()
68          return d
69
70      return closest_pair(S) # καλούμε την closest_pair με τη λίστα σημείων
71                             # και επιστρέφουμε την απόσταση που βρίσκει.
```

Η distance θεωρούμε ότι είναι μια συνάρτηση υπολογισμού της ευκλείδειας απόστασης 2 σημείων.

Ορθότητα:

Κοιτάζουμε πρώτα της closest_pair(P):

Αν το P που της δώσουμε έχει μόνο ένα σημείο μέσα τότε επιστρέφουμε άπειρο, γιατί θέλουμε μια πολύ μεγάλη τιμή ώστε να συνεχίσει να περιέχεται αυτό το σημείο στον χώρο που θα εξεταστεί από την βοηθητική συνάρτηση. Βρίσκουμε τον median για να μπορέσουμε να χωρίσουμε το πρόβλημα στην μέση και να εφαρμόσουμε divide and conquer. Βρίσκουμε αναδρομικά τις λύσεις στα δύο μισά του χώρου. Κρατάμε την μικρότερη από τις δύο αποστάσεις στην μεταβλητή d. Τώρα πρέπει να κάνουμε conquer το πρόβλημα της συνένωσης των δύο μισών. Αρχικά, d είναι η καλύτερη εκτίμηση που έχουμε μέχρι στιγμής για την ελάχιστη απόσταση 2 σημείων του χώρου μας και τα μόνα ζεύγη σημείων που δεν έχουμε ελέγξει ακόμα είναι τέτοια ώστε να βρίσκονται εκατέρωθεν του επιπέδου διαχωρισμού και να απέχουν απόσταση μικρότερη του d.

Αν ένα σημείο απέχει απόσταση μεγαλύτερη από d από το επίπεδο z = Z_coord, όπου έχουμε ορίσει το Z_coord να είναι η συντεταγμένη z του επιπέδου διαχωρισμού που βρήκαμε με το median τότε θα απέχει και απόσταση μεγαλύτερη από d από τα σημεία του χώρου που βρίσκονται στην άλλη μεριά αυτού του επιπέδου, άρα δεν πρόκειται να προκύψει μικρότερη τιμή για το d από αυτό το σημείο. Το εξαιρούμε λοιπόν και είμαστε σίγουροι ότι δεν επηρεάζουμε την ορθότητα της λύσης με τον τρόπο αυτόν (αυτό κάνει το for loop που έχουμε). Για τα υπόλοιπα σημεία καλούμε την helper(P,dist,Z_coord) η οποία παίρνει σαν ορίσματα: P → λίστα σημείων που μένουν, dist → η μέχρι τώρα εκτίμηση της ελάχιστης απόστασης, Z_coord → η συντεταγμένη z του επιπέδου διαχωρισμού που έχουμε χρησιμοποιήσει.

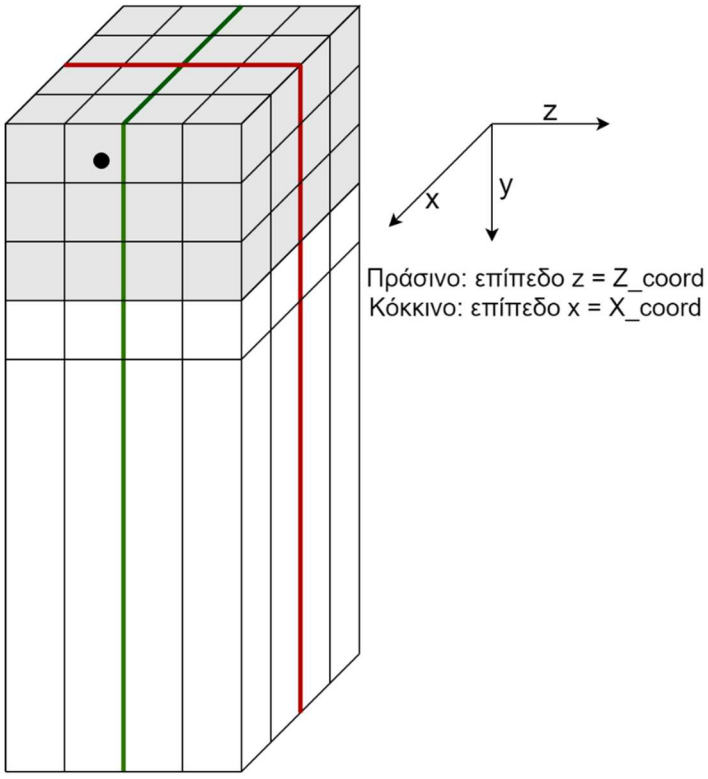
Ας δούμε τώρα την helper(P, dist, Z_coord):

Είναι η συνάρτηση closest_pair() του μαθήματος για 2 διαστάσεις, αλλά ελαφρώς τροποποιημένη ώστε να λειτουργεί στις 3 διαστάσεις. Τα βήματα μέχρι πριν το merge αιτιολούνται όπως και στην

closest_pair. Τώρα η σημασία του merge είναι ότι λόγω των αναδρομικών κλήσεων θα καταλήγουμε σε κάθε βήμα τα στοιχεία του P να είναι ταξινομημένα ως προς y. Ενσωματώνουμε ένα mergesort μέσα στην helper, δηλαδή. Θα τα χρειαστούμε ταξινομημένα αργότερα, οπότε με τον τρόπο αυτό φροντίζουμε η ταξινόμησή τους να μην ζημιώσει πολύ την πολυπλοκότητα του αλγορίθμου. Μετά το d είναι πάλι η καλύτερη εκτίμηση που έχουμε για την ελάχιστη απόσταση 2 σημείων και θέλουμε να εφαρμόσουμε το conquer step. Δηλαδή να βρούμε σημεία τα οποία απέχουν απόσταση μικρότερη του d και βρίσκονται εκατέρωθεν του επιπέδου x = X_coord. Τα σημεία αυτά με την ίδια αιτιολόγηση για τα σημεία στον closest_pair και το επίπεδο z = Z_coord δεν γίνεται να απέχουν πάνω από d από το επίπεδο διαχωρισμού x = X_coord.

Γίνεται να απέχουν πάνω από d από το επίπεδο Z και να έχουν μεταξύ τους απόσταση μικρότερη από d; Έστω ότι υπάρχουν 2 τέτοια σημεία τότε είναι αναγκαστικά από την ίδια μεριά του επιπέδου διαχωρισμού z = Z_coord άρα την μεταξύ τους απόσταση θα έπρεπε να είχαν βρει ως ελάχιστη οι αναδρομικές κλήσεις της closest_pair που έγιναν στο βήμα Divide πριν την κλήση της helper. Αυτό, λοιπόν, είναι άτοπο.

Γιατί εξετάζουμε 48 και όχι 11 επόμενους γείτονες όπως κάναμε στις 2 διαστάσεις;



Στο σημείο αυτό έχουμε έναν χώρο άπειρης έκτασης στον άξονα y που η διατομή του (στο επίπεδο xz) είναι ένα τετράγωνο πλευράς 2*d.

Όπως δείχνουμε δίπλα, χωρίζουμε τον χώρο σε κύβους διαστάσεων (d/2)X(d/2)X(d/2). Λόγω του ότι τα στοιχεία αριστερά του x = X_coord έχουν απόσταση το λιγότερο d μεταξύ τους και το ίδιο ισχύει και για στοιχεία δεξιά του επιπέδου διαχωρισμού αυτού έχουμε ότι ένας τέτοιος κύβος θα έχει το πολύ 1 στοιχείο μέσα του. Συνεπώς, αν κάνουμε τόσους ελέγχους όσα είναι τα γκριζαρισμένα κυβάκια δίπλα θα πάρουμε μια τιμή που σίγουρα αρκεί για να βρούμε το ελάχιστο. Είναι 3X4X4 = 48 κυβάκια.

Αφού τα στοιχεία μας είναι ταξινομημένα ως προς y τα παραπάνω μας πληροφορούν ότι αν συγκρίνουμε κάθε σημείο με τα 48 επόμενά του τότε σίγουρα θα το έχουμε συγκρίνει με κάθε σημείο που πιθανώς να έδινε απόσταση μικρότερη του d.

Υπολογιστική Πολυπλοκότητα:

Αρχικά, ας δούμε την πολυπλοκότητα της helper, έστω H(n) όπου n το μέγεθος του συνόλου P.

Κάνουμε μια εκτέλεση του median → O(n)

Divide βήμα σε 2 μισά → 2H(n/2)

Κάνουμε ένα merge → O(n)

Διατρέχουμε το P μια φορά για τις διαγραφές → O(n)

Στην χειρότερη περίπτωση δεν διαγράψαμε τίποτα. Μετά διατρέχουμε το P μια φορά κάνοντας σταθερό αριθμό (48) ελέγχων για κάθε στοιχείο του → O(n)

Έχουμε, δηλαδή, $H(n) = 2H\left(\frac{n}{2}\right) + O(n) \Longrightarrow H(n) = O(nlogn)$

Ας δούμε την πολυπλοκότητα του closest_pair(S): (Του αλγορίθμου μας). Έστω C(n)

Έστω ότι |S| = n.

Κάνουμε μια εκτέλεση του median → O(n)

Divide βήμα σε 2 μισά → 2C(n/2)

Διατρέχουμε το S για τις διαγραφές → O(n)

Έστω ότι δεν έγινε καμία διαγραφή και έχουμε ακόμα n σημεία στο S.

Καλούμε την helper → O(nlogn)

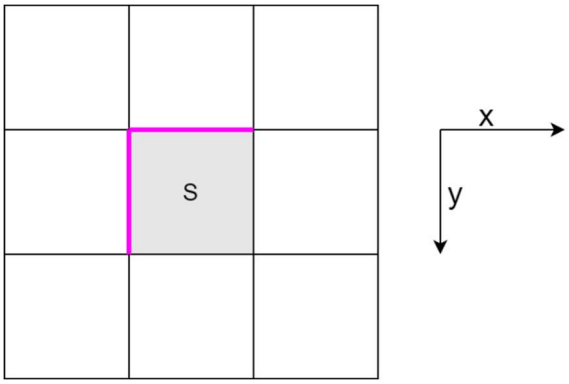
Άρα ο αλγόριθμός μας είναι: $C(n) = 2C\left(\frac{n}{2}\right) + O(nlogn) \Rightarrow C(n) = O(nlog^2 n)$

Χωρική πολυπλοκότητα: Μια διπλά διασυνδεδεμένη λίστα για να μπορούμε να κάνουμε τις διαγραφές σε σταθερό χρόνο και γενικά να έχουμε την παραπάνω χρονική πολυπλοκότητα η οποία περιέχει τριπλέτες των σημείων του χώρου μας (ή κάποια άλλη δομή που να επιτρέπει τις παραπάνω πράξεις έτσι όπως τις περιγράφουμε). Η χωρική πολυπλοκότητα είναι O(n).

(β)

Θα αρχίσουμε αναφέροντας μερικές ιδιότητες που θα χρησιμοποιήσουμε για να κατασκευάσουμε τον αλγόριθμο. Η διαγώνιος ενός κύβου d διαστάσεων (για d = 2 είναι ένα τετράγωνο) πλευράς L είναι $M = \sqrt{d} \cdot L$ (το χρησιμοποιούμε σαν γνωστό γεγονός από τα μαθηματικά).

Επειδή θα ασχοληθούμε με τέτοιους κύβους που συνορεύουν πρέπει να ορίσουμε ποιες έδρες ανήκουν σε κάθε κύβο ώστε αν ένα σημείο ανήκει στην έδρα ενός κύβου να μην τύχει να νομίζουμε ότι είναι και σημείο του γειτονικού του κύβου και έτσι το μετρήσουμε 2 φορές. Θα θεωρούμε ότι σε κάθε κύβο ανήκουν τα εσωτερικά του σημεία και τα σημεία των εδρών (πλευρών αν είναι τετράγωνο) του με την μικρότερη αντίστοιχη συντεταγμένη. Δείχνουμε ένα παράδειγμα σε 2 διαστάσεις ώστε να βγάζει νόημα αυτό:



Το τετράγωνο S θεωρούμε ότι περιλαμβάνει τα εσωτερικά του στοιχεία και από το σύνορό του περιλαμβάνει τις μωβ/ροζ γραμμές.

Η απόσταση M προκύπτει ως απόσταση 2 σημείων ενός d – διάστατου κύβου όταν αυτά είναι τα ακραία σημεία της κύριας διαγωνίου του. Ωστόσο, από τον τρόπο που έχουμε ορίσει ποια σημεία ανήκουν σε κάθε κύβο έχουμε ότι δεν θα ανήκουν και τα 2 αυτά σημεία στον ίδιο κύβο. Συνεπώς η απόσταση 2 σημείων ενός κύβου θα είναι πάντα μικρότερη από την διαγώνιό του M.

Στο ερώτημα (β) θέλουμε αλγόριθμο που να βρίσκει το δ* το οποίο από την εκφώνηση είναι ορισμένο για χώρο 3 διαστάσεων. Θεωρούμε d = 3.

Τι πλευρά πρέπει να έχει ένας κύβος με διαγώνιο M = l;

$$l = M = \sqrt{3} \cdot L \Longrightarrow L = \frac{l}{\sqrt{3}}$$

Συμπεραίνουμε ότι σε έναν κύβο με πλευρά $L = \frac{l}{\sqrt{3}}$ δεν μπορώ να βάλω 2 ή περισσότερα σημεία μέσα του αν θέλω όλα μου τα σημεία να απέχουν μεταξύ τους απόσταση τουλάχιστον l λόγω της ιδιότητας για την διαγώνιο του κύβου που είδαμε παραπάνω.

Έστω ένας κύβος πλευράς cl, θέλουμε να βρούμε ένα άνω όριο στο πλήθος σημείων που μπορεί να περιέχει για τα οποία κανένα ζευγάρι δεν έχει απόσταση μικρότερη από l.

Θέλω να δω πόσοι κύβοι πλευράς $L = \frac{l}{\sqrt{3}}$ μπορούν να χωρέσουν μέσα στον κύβο πλευράς cl. Έστω ότι η πλευρά cl αποτελείται από x τμήματα μήκους L τότε: $x \cdot \frac{l}{\sqrt{3}} = c \cdot l \Longrightarrow x = c\sqrt{3}$ και κρατάμε το ceiling αυτού του αριθμού, δηλαδή, $x = \lceil c\sqrt{3} \rceil$. Τότε θα χωρέσουν το πολύ x^3 κυβάκια μέσα στον κύβο

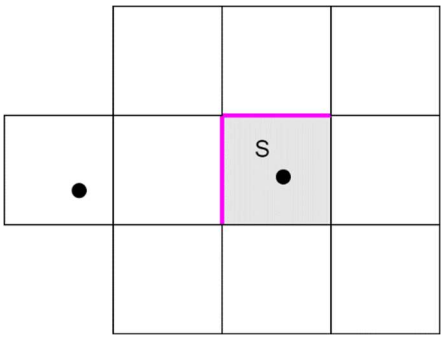
πλευράς cl και κάθε τέτοιο κυβάκι μπορεί να περιέχει το πολύ ένα σημείο (αν όλα τα σημεία απέχουν μεταξύ τους απόσταση το λιγότερο ίση με l) άρα το x^3 είναι μια σταθερά (εξαρτάται μόνο από το c) η οποία μας λέει πόσα σημεία (με αυτόν τον περιορισμό απόστασης) μπορούν να μπουν μέσα σε έναν κύβο πλευράς cl.

Ας διατυπώσουμε τον αλγόριθμο για το πρόβλημα αυτό. Έχουμε ως προσέγγιση $l \leq \delta^* \leq cl$ για την μικρότερη απόσταση 2 σημείων στον χώρο. Μπορούμε σε γραμμικό χρόνο να βρούμε τα όρια του χώρου (minimum και maximum στα x,y,z: τις 3 διαστάσεις) με αλγόριθμο που έχουμε δει στο μάθημα.

Χωρίζουμε τον χώρο μας σε ένα πλέγμα (mesh) από κυβάκια πλευράς cl τα οποία ονομάζουμε κελιά.

Διατρέχουμε όλα τα σημεία που έχουμε και με βάση τις συντεταγμένες τους τα τοποθετούμε στα αντίστοιχα κελιά. Λόγω του περιορισμού ότι η ελάχιστη απόσταση δύο σημείων είναι το λιγότερο l και τα κελιά είναι κύβοι πλευράς cl με βάση τα παραπάνω έχουμε ότι κάθε κελί περιέχει το πολύ $p = \lceil c\sqrt{3} \rceil^3$ σημεία. Το οποίο είναι μια σταθερά (εξαρτάται από το c μόνο). Η ιδέα αυτή είναι παρόμοια με τα buckets του bucketSort.

Ονομάζουμε γειτονιά ενός κελιού όλα τα κελιά που συνορεύουν αυτού συν τον εαυτό του.



Έστω ένα σημείο που βρίσκεται στο κελί S του χώρου, τότε, επειδή τα κελιά έχουν μήκος πλευράς c * l και η ελάχιστη απόσταση γνωρίζουμε ότι είναι το πολύ c * l, δεν χρειάζεται να κοιτάξουμε τις αποστάσεις μεταξύ σημείων που το ένα δεν ανήκει στην γειτονιά του άλλου γιατί θα είναι σίγουρα μεγαλύτερη του c * l όπως βλέπουμε για τα 2 σημεία της διπλανής εικόνας. Συνεπώς, η σύγκριση των αποστάσεων κάθε σημείου με τα σημεία του κελιού του και των γειτονικών κελιών και κανενός άλλου κελιού αρκεί για να βρούμε την δ*.

Σε 3 διαστάσεις κάθε κελί έχει 9 γείτονες από πάνω του, 9 γείτονες από κάτω του και 8 γείτονες γύρω του. Συνολικά, 26 γείτονες. Αν για ένα σημείο ελέγξω τις αποστάσεις του από κάθε σημείο όλων των γειτόνων του τότε θα κάνω το πολύ 26 * p πράξεις, το οποίο είναι μια σταθερά. Συνεπώς, για κάθε σημείο το συγκρίνω με τα σημεία στο ίδιο κελί με αυτό και με όλα τα σημεία σε γειτονικά κελιά άρα για κάθε σημείο θα κάνω 27 * p πράξεις. Η διαδικασία αυτή επαναλαμβάνεται n φορές (1 για κάθε σημείο). Συνεπώς, ο αλγόριθμος αυτός για την εύρεση της δ* έχει χρονική πολυπλοκότητα

$$O\left(n \cdot 3^3 \cdot \lceil c\sqrt{3} \rceil^3\right) = O(n)$$

```
Αλγόριθμος:
# Έστω ότι ο χώρος έχει 3 διαστάσεις
# x, y, z
S # S είναι η λίστα σημείων που έχουμε
# τα οποία είναι τριάδες της μορφής
# (x,y,z), δηλαδή, S[i]=(x,y,z) με
# 1 <= i <= n

# έχουμε δει στο μάθημα τον αλγόριθμο που
# βρίσκει το μέγιστο και το ελάχιστο στοιχείο
# ενός πίνακα σε Θ(n) χρόνο, ας τον ονομάσουμε
# minmax εδώ και ας θεωρήσουμε ότι επιστρέφει
# ζεύγος (min, max) ενώ στα ορίσματα παίρνει και
# το όνομα της διάστασης στην οποία θέλουμε να
# γίνει η αναζήτηση
# x --> 1, y --> 2, z --> 3 για τις διαστάσεις
(x_min, x_max) = minmax(S,1) # O(n)
(y_min, y_max) = minmax(S,2) # O(n)
(z_min, z_max) = minmax(S,3) # O(n)
# αφού γνωρίζουμε τον χώρο στον οποίο εργαζόμαστε
# δημιουργούμε ένα πλέγμα (mesh) με κελιά που είναι
# κυβάκια πλευράς c * l. Θεωρουμε ότι υπάρχει η συνάρτηση
# place(p) που βάζει το σημείο p στο κατάλληλο κελί με βάση
# τις συντεταγμένες του. Αυτό γίνεται σε O(1) χρόνο.
# Και η συνάρτηση get_cell(p) που μας λέει σε ποιο κελί
# ανήκει (θα έπρεπε να ανήκει) το σημείο p. O(1) χρόνος.
# Η συνάρτηση Mesh απλά μας δημιουργεί ένα τέτοιο πλέγμα
# με τις διαστάσεις που της δίνουμε
mesh = Mesh(x_min,x_max,y_min,y_max,z_min,z_max) # Έστω ότι
# η αρχικοποίηση γίνεται σε O(1) (είναι εφικτό)
for point in S:       # O(n)
    mesh.place(point) # βάλε όλα τα σημείο στα κελιά τους
d_star = c * l # Αρχικοποίησε το δ* στη μεγαλύτερη δυνατή τιμή

for point in S:
    cell = mesh.get_cell(point) # O(1)
    for my_neighbor in mesh.Neighbor(cell): # η συνάρτηση
    # Neighbor(cell) μας επιτρέπει απλά να κάνουμε iterate
    # στα γειτονικά κελιά του κελιού μας. Μπορούμε να θεωρήσουμε
    # ότι χρειάζεται O(1) χρόνο.
    # Έχουμε 27 γείτονες (η συνάρτηση αυτή εμπεριέχει και το δικό
    # μας κελί). Το 27 είναι σταθερά.
        for other_point in my_neighbor: # σταθερό μέγιστο πλήθος
        # σημείων μέσα σε κάθε κελί
            if (point != my_point):
                new_d = distance(point,my_point) # μια συνάρτηση
                # υπολογισμού της ευκλείδειας απόστασης.
                d_star = min(d_star, new_d) # ενημέρωσε το d_star
# ο βρόχος αυτός είναι O(n) (το μόνο που κάνει είναι να
# πραγματοποιεί τις συγκρίσεις αποστάσεων σημείων που έχουμε δείξει ότι
# είναι O(n).
return d_star # Συνολικά O(n).
```

Ορθότητα: ο αλγόριθμος υλοποιεί μόνο ό,τι περιγράψαμε στο κείμενο παραπάνω για τα οποία έχουμε αιτιολογήσει την ορθότητά τους εκεί.

Πολυπλοκότητα:

Χωρική πολυπλοκότητα:

Πίνακας S με τα σημεία → O(n)

Κάθε κελί αρκεί να έχει μια λίστα με τα σημεία που περιέχει άρα χρειαζόμαστε ήδη O(n) χώρο και μάλλον κάποιο map που να έχει τις συντεταγμένες κάθε κελιού που υπάρχει. Άρα O(n). Οι συναρτήσεις place και get_cell θα μετατρέπουν τις συντεταγμένες κάθε σημείου σε συντεταγμένες κελιού ώστε να βρουν το κατάλληλο κελί. Μπορεί να γίνει όντως σε O(1) χρόνο αφού είναι μόνο πράξεις.

Άρα συνολικά $O(n)$ χωρική πολυπλοκότητα.

Χρονική πολυπλοκότητα:

Εύρεση ορίων χώρου → O(n)

Δημιουργία πλέγματος → O(1)

Τοποθέτηση σημείων μέσα στα κελιά του πλέγματος → O(n)

Για κάθε σημείο εξέτασε όλους τους γείτονές του → $O\left(n \cdot 3^3 \cdot \left\lceil c\sqrt{3} \right\rceil^3\right) = O(n)$

Άρα συνολικά $O(n)$ χρονική πολυπλοκότητα.

Γενίκευση αλγορίθμου για d >= 2 διαστάσεις:

Θα βρίσκουμε τα όρια του χώρου και θα κατασκευάζουμε το mesh πάλι με τον ίδιο τρόπο σε O(n) χρόνο και τα υπόλοιπα είναι ίδια στον αλγόριθμο.

Διαφορές που έχουμε είναι: $M = \sqrt{d} \cdot L$ αλλά θέλουμε και πάλι M = l άρα εδώ $L = \frac{l}{\sqrt{d}}$

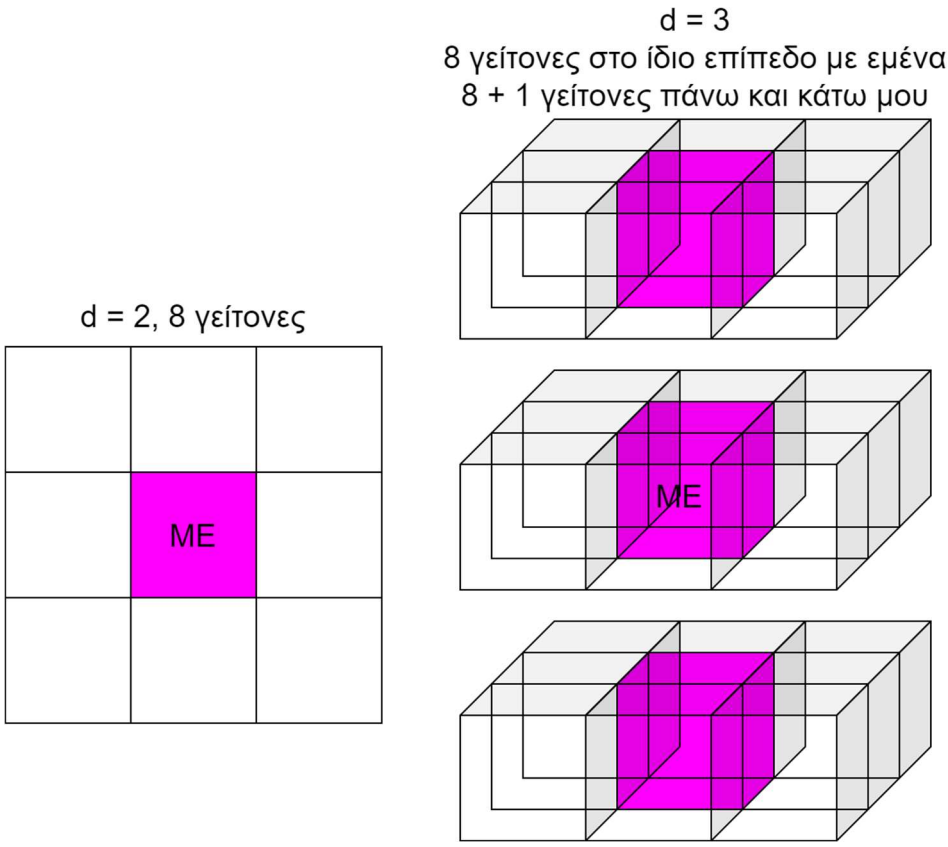
Με την ίδια ακριβώς ανάλυση με πριν βλέπουμε ότι κάθε κελί περιέχει το πολύ $p = \left\lceil c\sqrt{d} \right\rceil^d$ σημεία.

Πόσα γειτονικά κελιά έχει κάθε κελί; 3^d – 1. Απόδειξη με επαγωγή: Βάση επαγωγής: Ο τύπος ισχύει για 2 διαστάσεις. Επαγωγική υπόθεση: Έστω ότι σε d – 1 διαστάσεις είναι 3^(d – 1) – 1 γείτονες. Επαγωγικό βήμα: Στις d διαστάσεις, η extra διάσταση είναι σαν να προστέθηκε μία έννοια του «επιπέδου» έχω, λοιπόν, τους γείτονες των d – 1 διαστάσεων στο επίπεδό μου, τόσους από κάτω μου και τόσους από πάνω μου. Βεβαίως, κάτω και πάνω μου έχω έναν επιπλέον γείτονα ο οποίος καταλαμβάνει στο επίπεδό του την ίδια θέση που καταλαμβάνω εγώ στο επίπεδό μου:

Συνολικά: 3^(d – 1) – 1 + 3^(d – 1) – 1 + 1 + 3^(d – 1) – 1 + 1 = 3^d – 1 γείτονες. Δηλαδή, πολυπλοκότητα στον χρόνο: (c, d είναι σταθερές)

$$O\left(n \cdot 3^d \cdot \left\lceil c\sqrt{d} \right\rceil^d\right) = O(n)$$

Δείχνουμε γραφικά την παραπάνω ιδέα με τα επίπεδα γειτόνων για το βήμα από d = 2 σε d = 3.



Συνεπώς, για d >= 2 μπορούμε να χρησιμοποιήσουμε τον ίδιο αλγόριθμο αλλάζοντας μόνο τον τρόπο που βρίσκουμε τα όρια του χώρου μας και κατασκευάζουμε το πλέγμα κελιών με κύβους d διαστάσεων με μήκος πλευράς c * l.

Άσκηση 2: Πόρτες Ασφαλείας στο Κάστρο

```
**************************************************************************
1   Αλγόριθμος:
2   # ως 'πρώτη πόρτα' θα αναφερόμαστε στην πρώτη πόρτα
3   # την οποία δεν έχουμε καταφέρει να αντιστοιχίσουμε με
4   # διακόπτη και βρίσκεται πιο κοντά σε εμάς στον διάδρομο
5
6   # όταν λέμε ότι αγνοούμε διακόπτες, τότε εννοούμε ότι
7   # με κάποιον τρόπο θυμόμαστε ότι αυτός ο διακόπτης έχει
8   # ήδη αντιστοιχιστεί με πόρτα και δεν τον συμπεριλαμβάνουμε
9   # στην υπόλοιπη διαδικασία (όταν κοιτάζουμε τους διακόπτες
10  # τον προσπερνάμε σαν να μην υπάρχει)
11
12  # Θεωρούμε ότι οι διακόπτες βρίσκονται σε μια οριζόντια σειρά
13  # οπότε οι εκφράσεις 'αριστερό μισό', 'μέσο', 'δεξί μισό' να
14  # βγάζουν νόημα
15  Αρχή:
16      δες την κατάσταση της πρώτης πόρτας και κράτα την στη μνήμη σου
17
18  Βρόγχος:
19      Βήμα 2: Αν έχει μείνει μόνο ένας διακόπτης που δεν αγνοείς τότε
20              πήγαινε στο Τέλος
21      Βήμα 1-3: Άλλαξε την κατάσταση στους αριστερούς μισούς διακόπτες
22                από το σύνολο διακοπτών που δεν αγνοείς
23      Βήμα 4: σύγκρινε την κατάσταση της πόρτας με αυτήν που είχες δει
24              προηγουμένως.
25              Αν οι δύο καταστάσεις ταυτίζονται τότε
26                  επανάλαβε τον Βρόγχο αγνοώντας τους αριστερούς μισούς
27                  διακόπτες
28              Αν οι δύο καταστάσεις δεν ταυτίζονται τότε
29                  επανάλαβε τον Βρόγχο αγνοώντας τους δεξιούς μισούς
30                  διακόπτες
31  Τέλος:
32      Ο διακόπτης που έχει απομείνει ταιριάζει με την πρώτη πόρτα.
33      Αν η κατάστασή της είναι ανοικτή (το θυμόμαστε από την τελευταία φορά
34      που το ελέγξαμε) τότε ξέρουμε ότι ο διακόπτης αυτός ανοίγει την πόρτα
35      αυτή στην θέση που βρίσκεται.
36      Αλλιώς, αλλάζουμε την κατάσταση του διακόπτη και, πλέον, ισχύει το
37      ίδιο με παραπάνω.
38
39      Αφού ταυτοποίησες αυτόν τον διακόπτη με μια πόρτα, αγνόησέ τον και
40      Αν δεν απομένει διακόπτης που να μην αγνοείς σταμάτα
41      Αλλιώς πήγαινε στην Αρχή
**************************************************************************
```

Η αρίθμηση των βημάτων στον Βρόγχο είναι κάπως ανακατεμένη ώστε να ταιριάζει με την αρίθμηση στον ακόλουθο ψευδοκώδικα. Θα διαλέγαμε τον παραπάνω τρόπο για να εξηγήσουμε τον αλγόριθμο σε κάποιον φίλο μας, ωστόσο, δείχνουμε στη συνέχεια και μια εκδοχή του που να μοιάζει περισσότερο με κάτι που θα εκτελούσε ένας υπολογιστής και αναλύουμε την ορθότητα εκείνου του ψευδοκώδικα.

```
**********************************************************************
1   Αλγόριθμος:
2
3   result
4   # result[s] = (d, st)
5   # στον πίνακα αυτόν γράφουμε τα αποτελέσματα, ο διακόπτης s
6   # ανοίγει την πόρτα d όταν έχει την κατάσταση st
7   switch_list
8   # η switch_list είναι μια λίστα από τούπλες (s,st)
9   # δηλαδή, το switch_list[x] = (s, st) μας λέει ότι στην
10  # θέση x της λίστας βρίσκεται ο διακόπτης με αριθμό s και
11  # κατάσταση st. Αρχικά το switch_list περιέχει τις τούπλες
12  # με την σειρά που βλέπουμε τους διακόπτες με την άφιξή στο κάστρο
13  k = n
14  unlocked = 0
15
16  Αρχή:
17  check_door = η κατάσταση της πόρτας (unlocked + 1)
18  min = 1, max = k
19
20  Βρόγχος:
21      Βήμα 1: mid = floor{(min + max)/2}
22      Βήμα 2: if min == max: Πάω στο Τέλος
23      Βήμα 3: for i in range(min, mid + 1):
24                  switch_list[i].change_state()
25      Βήμα 4:
26      door = η κατάσταση της πόρτας (unlocked + 1)
27      if (door == check_door):
28          min = mid + 1
29          # max stays the same
30      else:
31          # min stays the same
32          max = mid
33      check_door = door # ενημερώνω την προηγούμενη κατάσταση της
34                        # πόρτας
35      Πάω στο Βρόγχος
36
37  Τέλος:
38      unlocked += 1
39      # έχουμε βρει τον διακόπτη της πόρτας unlocked
40      s = switch_list[mid].s # η ταυτότητα του διακόπτη
41      if (door == closed):
42          switch_list[mid].change_state()
43      # κατάσταση διακόπτη + κατάσταση πόρτας μας βοηθούν να
44      # συμπαιράνουμε σε ποια θέση ο διακόπτης αυτός ανοίγει την πόρτα
45      st = switch_list[mid].st # η κατάσταση αυτής της πόρτας
46      result[s] = (unlocked, st) # αποθηκεύουμε τον διακόπτη
47      switch_list.remove(mid) # τον αφαιρούμε από τη λίστα, για
48      # να μην προκαλεί προβλήματα με τους δείκτες
49      k -= 1
50      if (k == 0): επιστρέφω result
51      else: Πάω στην Αρχή
**********************************************************************
```

Αναπαράσταση προβλήματος: Έστω ότι έχουμε ονοματίσει τους διακόπτες από 1 έως n (τους χαρακτήρες της συμβολοσειράς διακοπτών) χωρίς να αλλάξουμε ποτέ τα ονόματα και έστω η πρώτη πόρτα του διαδρόμου προς το μέρος μας είναι η 1 και κάθε επόμενη πόρτα έχει δείκτη αυξανόμενο κατά 1. Η συνάρτηση change_state() εναλλάσει το state από 0 (off) σε 1 (on) και αντίστροφα όταν καλείται.
Ορθότητα διαδικασίας ταυτοποίησης πορτών/διακοπτών:

Αρχή: αρχικοποιεί τις τιμές των min και max όπου [min, max] το διάστημα στο οποίο θα υπάρχει ο διακόπτης που αναζητούμε μέσα στην switch_list. Αρχικά, min = 1 και max = k και η λίστα διακοπτών έχει k στοιχεία σε αυτό το σημείο άρα κάποιο από αυτά αναγκαστικά ανοίγει την πόρτα unlocked + 1* (η πρώτη πόρτα που δεν έχουμε ανοίξει).

Βρόγχος: Παίρνει το μέσο του διαστήματος πιθανών διακοπτών και θέτει τους διακόπτες από την αρχή (min) έως αυτό το σημείο στη switch_list σε αντίθετη κατάσταση. Έπειτα, συγκρίνουμε το πριν και το μετά της πόρτας. Αν η κατάσταση παραμένει ίδια τότε αναγκαστικά κανένας από τους διακόπτες που αλλάξαμε δεν ελέγχει αυτήν την πόρτα, οπότε η λύση πρέπει να είναι στους άλλους διακόπτες οπότε min = mid + 1 και εξετάζουμε το διάστημα [min, max]. Αλλιώς, η πόρτα άλλαξε θέση άρα κάποιος από τους διακόπτες στο διάστημα [min, mid] την ελέγχει, οπότε ψάχνουμε για τη λύση μόνο σε αυτό το διάστημα θέτοντας max = mid. Αυτό μαζί με το γεγονός ότι για την πόρτα 1 ψάχνουμε σε όλους τους διαθέσιμους διακόπτες αποδεικνύουν και το επιχείρημά μας στο *, δηλαδή, ότι το διάστημα [min, max] έτσι όπως το θεωρούμε πάντα περιέχει τον διακόπτη που ανοίγει την πόρτα (unlocked + 1).
(Στο παραπάνω βήμα οι διακόπτες που θέτουμε δεν είναι οι min, …, mid, αλλά οι switch_list[min].s, …, switch_list[mid].s)

Τέλος: Οι γραμμές 41-42 μας επιτρέπουν να έχουμε ανοιχτή την πόρτα ώστε να μην εμποδίζει στην αναζήτηση του διακόπτης της επόμενής της. Αποθηκεύουμε την αντιστοιχία διακόπτη – πόρτας – θέσης που την ανοίγει και αφαιρούμε τον διακόπτη αυτόν από τη switch_list. Αυτό το κάνουμε γιατί μειώνουμε κατά 1 και τα δυνατά indexes (γραμμή 49) οπότε πρέπει κάθε φορά που βρίσκουμε μια αντιστοιχία διακόπτη – πόρτας να αγνοούμε τον διακόπτη αυτόν από την υπόλοιπη διαδικασία. Κάθε φορά που φτάνουμε στο Τέλος έχουμε βρει από έναν διαφορετικό διακόπτη. Συνεπώς, κάποια στιγμή η διαδικασία θα τελειώσει.

Πλήθος διαμορφώσεων που δοκιμάζω:

Έστω ότι έχω k = 2^logk διαθέσιμους διακόπτες και βρίσκομαι στην Αρχή τότε θα κάνω

Επανάληψη:	0 + 0	0 + 1	…	0 + logk
Πλήθος διακοπτών:	2^(logk – 0)	2^(logk – 1)	…	2^(logk – logk)

Σε κάθε επανάληψη ελέγχω μία μόνο διαμόρφωση. Άρα θα ελέγξω logk διαμορφώσεις.

Συνολικά έχω: $\sum_{i=1}^{n} logk = \log(n!)$ διαμορφώσεις που θα δοκιμαστούν, οι οποίες είναι και $\Theta(nlogn)$.

Θέλουμε η λύση μας να είναι αποδοτική οπότε μπορούμε να θεωρήσουμε ότι η switch_list είναι υλοποιημένη με τρόπο που να μας επιτρέπει να κάνουμε remove σε O(1) χρόνο και μπορούμε να φανταστούμε τα min, max, mid ως δείκτες σε στοιχεία της switch_list ώστε να μην χρειάζεται πάντοτε να τα αναζητούμε. Ενώ μπορούμε να σκεφτόμαστε τα for loops ως iterations που διατρέχουν διαδοχικά τα στοιχεία της switch_list.

Άσκηση 3: Φόρτιση Ηλεκτρικών Αυτοκινήτων

```
**********************************************************************
1     Αλγόριθμος:
2
3     Βήμα 1:
4
5     curr = 1
6     time[curr] = a1
7     for i in range(1, n + 1):
8         if (time[curr] != ai):
9             curr = curr + 1
10            time[curr] = ai
11        cars[curr] += 1
12    # time: έχει μόνο τις στιγμές κατά τις οποίες πραγματικά
13    # έρχεται αυτοκίνητο.
14    # Δηλαδή, το time[i] είναι η τιμή της i-οστη στιγμής κατά
15    # την οποία υπάρχει όντως άφιξη αυτοκινήτου, θα αναφερόμαστε
16    # σε αυτές τις στιγμές ως στιγμές αφίξεων.
17    # Το cars[i] έχει το πλήθος αυτοκινήτων που φτάνουν στο σύστημα
18    # την στιγμή άφιξης time[i].
19
20    l = curr # το curr αυτή την στιγμή είναι η τιμή των πόσων
21             # στιγμών αφίξεων υπάρχουν
22    min = 0, max = n # Αρχικοποίηση, θα χρειαστούμε τουλάχιστον
23                     # 0 και το πολύ 'n' εξυπηρετητές
24    Βρόγχος:
25        Βήμα 2.1:
26                servers = floor{(max + min)/2}
27                if (min == max): Πάω στο Τέλος
28                traffic = 0 # στην αρχή η κίνηση που μας
29    # παραδίδεται από την προηγούμενη στιγμή αφίξεων είναι 0
30    # γιατί δεν υπάρχει προηγούμενη τέτοια στιγμή
31        Βήμα 2.2:
32            for i in range(1, l + 1):
33                traffic += cars[i]
34    # κίνηση = παρελθοντικό περίσσευμα + τωρινή κίνηση
35                if (traffic > servers * d):
36                    min = servers + 1
37                    # max stays the same
38                    πάω στο Βήμα 2.1
39                else if (i != l): # στο τελευταίο δεν
40                                  # χρειάζεται να
41                                  # ελέγχουμε το traffic
42                    traffic = max{0, traffic -
43                        servers * (time[i + 1] - time[i])}
44                        # αφαιρώ τα εξυπηρετούμε αυτοκίνητα μέχρι
45                        # την επόμενη στιγμή αφίξεων
46        Βήμα 2.3:
47            # min stays the same
48            max = servers
49            πάω στο Βήμα 2.1
50
51    Τέλος: επιστρέφω servers
**********************************************************************
```

Τα time, cars μπορούμε να τα υλοποιήσουμε με διάφορες δομές, θα μπορούσαν να είναι και διασυνδεδεμένες λίστες, στον παραπάνω ψευδοκώδικα τα δείχνουμε σαν να ήταν πίνακες. Θεωρούμε ότι τα στοιχεία του πίνακα cars είναι αρχικοποιημένα στο μηδέν.

Αλγόριθμος: (με λόγια)

Βήμα 1: Φτιάχνουμε τους πίνακες time, cars που περιέχουν στιγμές αφίξεων και πόσες αφίξεις είχαμε εκείνη την στιγμή στο σύστημα αντίστοιχα. Στιγμή άφιξης είναι μια χρονική στιγμή κατά την οποία υπάρχει τουλάχιστον μία άφιξη.

Βρόγχος:
Βήμα 2.1: Παίρνουμε ως πλήθος servers την τιμή (περίπου) στο μέσο του επιτρεπτού εύρους. Αν το επιτρεπτό εύρος τιμών είναι μόνο μία τιμή τότε πάμε στο Τέλος. Θεωρούμε την κίνηση (traffic) από το παρελθόν ως 0 (αφού δεν υπάρχει παρελθόν).

Βήμα 2.2: Διατρέχουμε τις στιγμές αφίξεων και για κάθε μία από αυτές ενημερώνουμε την κίνηση και ελέγχουμε την ανισότητα στη γραμμή 35. Αν το σύστημά μας δεν αρκεί για τα αυτοκίνητα τότε μπαίνουμε στο if, θέτουμε min = mid + 1 και γυρίζουμε στο Βήμα 2.1 για να ξαναδοκιμάσουμε να εξυπηρετήσουμε τα αυτοκίνητα. Αν το σύστημά μας αρκεί τότε μπαίνουμε στο else if στο οποίο ενημερώνουμε την κίνηση που θα πρέπει να εξυπηρετηθεί την επόμενη στιγμή άφιξης αφαιρώντας τα αυτοκίνητα που προλαβαίνουμε να φορτίσουμε (time[i + 1] – time[i]) είναι ο χρόνος που μεσολαβεί και το πολλαπλασιάζουμε με το servers, που είναι οι υποδοχές φόρτισής μας. Αν είμαστε στην τελευταία στιγμή αφίξεων τότε δεν χρειάζεται να υπολογίσουμε το traffic για το μέλλον, οπότε για αυτό υπάρχει η συνθήκη στο else if.

Βήμα 2.3: Φτάνουμε εδώ αν έχουμε εξυπηρετήσει επιτυχώς όλα τα αυτοκίνητα. Θέτουμε max = servers και πάμε στο Βήμα 2.1 για να δοκιμάσουμε με λιγότερους servers.

Τέλος: επιστρέφω το servers.

Ορθότητα:

Βήμα 1: Στα στοιχεία του πίνακα cars πραγματοποιούμε μια απλή καταμέτρηση ενώ στα στοιχεία του πίνακα time καταγράφουμε τις στιγμές άφιξης με την σειρά εμφάνισής τους στην ακολουθία {an} της εισόδου. Το αποτέλεσμα είναι ορθό διότι η ακολουθία {an} είναι αύξουσα και άρα μόλις παρατηρήσουμε αλλαγή σε 2 διαδοχικές τιμές της αυτό σημαίνει ότι πάμε στην επόμενη (χρονικά) στιγμή άφιξης.

Βρόγχος:
Βήμα 2.1: Θεωρούμε ως επιτρεπτό εύρος πιθανών τιμών για το πλήθος των υποδοχών φόρτισης το [min, max]* και η τιμή servers που διαλέγουμε είναι μέσα σε αυτό το εύρος τιμών άρα είναι δεκτή. Θεωρούμε την κίνηση από το παρελθόν 0 και αν το εύρος επιτρεπτών τιμών
[min, max] έχει εκφυλιστεί σε μία μόνο τιμή τότε είναι σωστό να επιστρέψουμε αυτήν λόγω του τρόπου που κάνουμε τα “πάω στο Βήμα 2.1” στις γραμμές 38, 49 **.

Βήμα 2.2: Θέλουμε να ελέγξουμε την τωρινή κίνηση οπότε η αύξηση του traffic είναι σωστή.
Αν traffic > servers * d τότε δεν υπάρχει τρόπος να δρομολογήσουμε όλα τα αυτοκίνητα στους servers και να εξυπηρετηθούν σε το πολύ d χρονικές μονάδες. Θα έχουμε περίσσευμα traffic – servers * d. Άρα πρέπει να δοκιμάσουμε διαφορετική τιμή για τους servers.

Αιτιολόγηση για το ** όσον αφορά την γραμμή 38 και για το * όσον αφορά το ότι το εύρος επιτρεπτών δυνατών τιμών είναι κάτω φραγμένο από το min: min = servers + 1 και προσπερνάμε όλες τις ενδιάμεσες τιμές από το προηγούμενο min έως και το servers.

Έστω ότι δεν αρκούν s υποδοχές για το σύστημά μας, υπάρχει τρόπος να αρκούν z < s υποδοχές; Έστω μια στιγμή αφίξεων m κατά την οποία traffic_s > s * d ώστε να συμπεράνουμε ότι δεν αρκούν οι s υποδοχές τότε s > z, δηλαδή, s * d > s * z. Η τιμή της κίνησης (traffic) για τις z υποδοχές θα είναι διαφορετική αφού το traffic εξαρτάται από τους servers. Από τις γραμμές 42, 43 βλέπουμε ότι το traffic μεταξύ διαδοχικών στιγμών αφίξεων μειώνεται κατά servers * (διαφορά χρόνου). Για τις s, z η διαφορά χρόνου θα είναι ίδια ενώ s > z άρα στην περίπτωση του συστήματος με s servers το traffic που οφείλεται στο παρελθόν θα είναι μικρότερο από ότι στο άλλο σύστημα. Βέβαια, μπορεί να είναι και ίσα, αν είναι μηδέν. Συνεπώς traffic_z >= traffic_s > s * d > s * z άρα σε αυτό το σημείο θα αντιμετωπίσει πρόβλημα και το σύστημα με z υποδοχές φόρτισης.

Εδώ είδαμε ότι servers δεν αρκούν άρα όλες οι τιμές μικρότερες ή ίσες του servers αποκλείονται.

Τμήμα else if: Στη σημείο αυτό έχουμε traffic <= servers * d. Τότε σε d χρονικές μονάδες προλαβαίνουμε να τους εξυπηρετήσουμε όλους. Φροντίζουμε να εξυπηρετείται πάντα ο πιο παλιός στο σύστημα (με μια ουρά αναμονής για τα αυτοκίνητα για παράδειγμα) να μην ξεμένει μέσα στο traffic κάποιο αυτοκίνητο που συνεχώς του “κλέβουν” την σειρά και άρα θα περιμένει πολύ.

Βήμα 2.3: Υπόλοιπη αιτιολόγηση για τα * και **:

Το ότι φτάσαμε σε αυτό το σημείο του αλγορίθμου σημαίνει ότι η επιλογή μας για servers είναι σίγουρα αρκετή για να εξυπηρετήσει όλα τα αυτοκίνητα. Όταν μικραίνουμε το εύρος των δυνατών τιμών θέτοντας max = servers και πηγαίνοντας στο Βήμα 2.1 για να ξαναδοκιμάσουμε να τους εξυπηρετήσουμε αφήνουμε στο εύρος [min, max] την τιμή max (που είναι ίση με τους αρκούντες servers αυτής της επανάληψης). Συνεπώς, το max είναι πάντοτε μια τιμή servers που λύνει το πρόβλημα. Ο λόγος που διώχνουμε τιμές μεγαλύτερες από το max είναι γιατί ψάχνουμε τον ελάχιστο αριθμό servers που λύνει το πρόβλημα.

Το διάστημα επιτρεπτών τιμών μικραίνει από τις γραμμές 36, 48. Η γραμμή 48 δεν πρόκειται ποτέ, όπως είπαμε, να αφήσει εκτός του [min,max] όλες τις τιμές που λύνουν το πρόβλημα (το max πάντα το λύνει) και η γραμμή 36 πάντοτε αποκλείει μόνο μη αρκούσες τιμές για το servers. Άρα όταν στο διάστημα μείνει μόνο μια τιμή αυτή θα είναι ορθή.

Ξεκινάμε με [min, max] = [0, n] το οποίο σίγουρα περιέχει την λύση (n servers σίγουρα αρκούν).

Γιατί ο αλγόριθμος αυτός βρίσκει τον ελάχιστο αριθμό από servers που λύνουν το πρόβλημα;

Από τον τρόπο που αποκλείουμε τιμές βλέπουμε ότι:

[min, …, servers, …, max]: τιμές από το min έως και το servers αποκλείονται μόνο όταν είμαστε σίγουροι ότι καμία τους δεν λύνει το πρόβλημα και τιμές από το max έως και το servers + 1 αποκλείονται όταν είμαστε σίγουροι ότι το servers λύνει όντως το πρόβλημα. Άρα όταν φτάσουμε σε σημείο να έχουμε min = max = servers τότε όλες οι τιμές μικρότερες από το min εκείνη την στιγμή δεν λύνουν το πρόβλημα ενώ το max σίγουρα το λύνει. Άρα αυτή η τιμή του max είναι η s* που ψάχνουμε.

Υπολογιστική πολυπλοκότητα:

Χρονική πολυπλοκότητα:

Βήμα 1: Η εκτέλεση γίνεται σε O(n) χρόνο αφού διατρέχουμε τις n αφίξεις μια φορά την καθεμιά και οι πράξεις που κάνουμε για καθεμιά από αυτές είναι O(1).

Βρόγχος:

Πλήθος επαναλήψεων Βρόγχου: αρχίζουμε με n = 2^logn (n + 1 στην πραγματικότητα αλλά δεν μας πειράζει) πιθανές τιμές για τους servers. Σε κάθε βήμα προχωράμε το min ή το max στο servers που είναι το μέσο άρα μειώνουμε τις πιθανές τιμές στο μισό μέχρι να μείνει 1 μόνο τιμή.

Επανάληψη:	0 + 0	0 + 1	…	0 + logn
Δυνατές τιμές:	2^(logn – 0)	2^(logn – 1)	…	2^0 = 2^(logn – logn)

Άρα θα γίνουν logn επαναλήψεις του Βρόγχου.

Πλήθος επαναλήψεων ανά επανάληψη του Βρόγχου: Η χειρότερη περίπτωση είναι να γίνονται l επαναλήψεις στο Βήμα 2.2 σε κάθε επανάληψη του Βρόγχου (έστω ότι κάθε φορά η «εκτίμηση» αρκεί για να εξυπηρετήσουμε όλα τα αυτοκίνητα). Το l είναι το πλήθος των στοιχείων των λιστών/πινάκων (ανάλογα πώς το υλοποιούμε) time και cars από το Βήμα 1. Όπως είπαμε και στα σχόλια το l είναι το πλήθος των στιγμών αφίξεων. Κάθε επανάληψη του for loop στο Βήμα 2.2 γίνεται σε O(1).

Αν n > T τότε ανεξάρτητα από το ότι τα αυτοκίνητα είναι περισσότερα από τις στιγμές δεν γίνεται l > T λόγω του περιορισμού στις τιμές της {an} άρα l = O(T).

Αν n <= T τότε το πολύ κάθε αυτοκίνητο θα φτάνει σε διαφορετική στιγμή άρα l = O(n).

Συνολικά, βλέπουμε ότι l = O(min{n,T}) σε κάθε περίπτωση.

Για τον Βρόγχο, λοιπόν, γίνονται O(min{n, T} * logn) εντολές συνολικά.

Συνολικά έχουμε ότι η χρονική πολυπλοκότητα είναι:
$O(n + \min\{n, T\} * logn) = O(n + nlogn) = O(nlogn)$.

Χωρική πολυπλοκότητα:

Χρησιμοποιούμε δύο πίνακες μεγέθους l. Τα ai που είναι n στο πλήθος τα διαβάζουμε μια φορά ως είσοδο αλλά δεν μας νοιάζει να τα αποθηκεύσουμε κάπου, οπότε δεν θα τα συμπεριλάβουμε στην χωρική πολυπλοκότητα.

Άρα η χωρική πολυπλοκότητα είναι: $O(\min\{n, T\}) = O(n)$.

Οι τύποι με τα min{n,T} δίνουν καλύτερο άνω φράγμα της πολυπλοκότητας άρα συμπεριλαμβάνονται.

Άσκηση 4: Παραλαβή Πακέτων

1. Μας δίνεται η ακολουθία $S = ((w_1, p_1), (w_2, p_2), \dots, (w_n, p_n))$ και έστω μια αναδιάταξη της ακολουθίας αυτής την οποία συμβολίζουμε με $S' = ((w'_1, p'_1), (w'_2, p'_2), \dots, (w'_n, p'_n))$ τότε ο συνολικός βεβαρυμμένος χρόνος εξυπηρέτησης της S' είναι:

$$time(S') = \sum_{i=1}^{n} w'_i \sum_{j=1}^{i} p'_j = \sum_{i=1}^{n} p'_i \sum_{j=i}^{n} w'_j \ (1)$$

Από την δεύτερη έκφραση του $time(S')$ μπορούμε να δούμε ότι κάθε παραλαβή δέματος «επιβαρύνει» το αποτέλεσμα κατά το γινόμενο του χρόνου προετοιμασίας της επί το άθροισμα των βαρών αυτής και όλων των εργασιών που ο υπάλληλος θα εξυπηρετήσει μετά. Για να υπολογίσουμε το $\min\{time(S')\}$ για κάθε αναδιάταξη S' της S μπορούμε να χρησιμοποιήσουμε την εξής αναδρομή:

$$time(X, W) = \min_{\forall (w_k, p_k) \epsilon X} \{p_k(w_k + W) + time(X_k, W + w_k)\} \ όπου \ X_k = X \setminus \{(w_k, p_k)\} \ και \ X \neq \emptyset \ (2)$$

$$time(\emptyset, W) = 0$$

τότε $time(S, 0) = \min_{S' αναδιάταξη \ της \ S} \{time(S')\}$, αφού η αναδρομή κάθε φορά κρατάει το ελάχιστο από την εργασία που δρομολογεί τώρα (δείκτης k) και τις παραλαβές που δρομολογήθηκαν να συμβούν νωρίτερα (χρονικά) από αυτήν (time(Xk, W + wk)). Οι αναδρομές που θα χρησιμοποιήσουμε όπως φαίνεται και εδώ λειτουργούν σαν να διαλέγουν ποια παραλαβή θα μπει τελευταία στην ουρά. Χτίζουν την ουρά εξυπηρέτησης από το τέλος προς την αρχή.

Για να διευκολυνθούμε στην διατύπωση του αλγορίθμου μας υποθέτουμε ότι η ακολουθία S μας δίνεται ταξινομημένη ως προς την ποσότητα w/p σε φθίνουσα σειρά. Δηλαδή, υποθέτουμε ότι

$$\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \dots \geq \frac{w_n}{p_n}$$

Ως συνάρτηση χρόνου με άπλητο κριτήριο θεωρούμε την

$$myTime(X, W) = p_j(w_j + W) + myTime(X_j, W + w_j),$$

$$όπου \ X_j = X \setminus \{(w_j, p_j)\} και \ j \ τέτοιο \ ώστε \frac{w_j}{p_j} το \ ελάχιστο \ στα \ στοιχεία \ του \ X \ και \ X \neq \emptyset$$

$$myTime(\emptyset, W) = 0$$

Θέλουμε να δείξουμε ότι $myTime(S, 0) \leq time(S, 0)$.

Βάση επαγωγής: $myTime(X, W) \leq time(X, W), αν \ X \ έχει \ 0 \ ή \ 1 \ στοιχεία.$

Επαγωγική Υπόθεση: $myTime(X, W) \leq time(X, W)$ με $|X| \leq n - 1$

Επαγωγικό βήμα: Θέλω να δείξω ότι $myTime(S, W) \leq time(S, W), |S| = n$

Δηλαδή, θέλω να δείξω ότι

$$p_n(w_n + W) + myTime(S_n, w_n + W) \leq p_i(w_i + W) + time(S_i, w_i + W)\ για\ κάθε\ i = 1, \dots, n$$

Ωστόσο, γνωρίζω ότι $myTime(S_i, w_i + W) \leq time(S_i, w_i + W) \Leftrightarrow$
$p_i(w_i + W) + myTime(S_i, w_i + W) \leq p_i(w_i + W) + time(S_i, w_i + W)\ για\ κάθε\ i = 1, \dots, n\ (\alpha)$
από επαγωγική υπόθεση.
Θα προσπαθήσω να αποδείξω ότι
$p_n(w_n + W) + myTime(S_n, w_n + W) \leq p_i(w_i + W) + myTime(S_i, w_i + W)\ για\ κάθε\ i = 1, \dots, n (\beta)$
τότε από (α) και (β) θα έχω αποδείξει το επαγωγικό βήμα.

$$p_n(w_n + W) + myTime(S_n, w_n + W) \leq p_i(w_i + W) + myTime(S_i, w_i + W) \Leftrightarrow$$

$$W(p_n + \dots + p_i + \dots + p_1) + p_n w_n + \dots + p_{i+1}(w_n + \dots + w_{i+1}) + p_i(w_n + \dots + w_i) + OTHERS$$

$$\leq$$

$$W(p_n + \dots + p_i + \dots + p_1) + p_i w_i + p_n(w_i + w_n)^{***} + \dots + p_{i+1}(w_n + \dots + w_{i+1} + w_i) + OTHERS$$

Τα πάμε όλα στο πρώτο μέλος και βλέπουμε ότι ο όρος του W φεύγει και τα OTHERS είναι ίδια και στα δύο άρα απαλείφονται. Μένουν μόνο οι όροι μεταξύ των n και i.

$$-p_n w_i - \dots - p_{i+1} w_i + p_i(w_n + \dots + w_{i+1}) \leq 0 \Leftrightarrow$$

$$(w_i p_n - p_i w_n) + \dots + (w_i p_{i+1} - p_i w_{i+1}) \geq 0$$

Έστω i <= k <= n τότε λόγω του τρόπου που θεωρούμε ότι είναι ταξινομημένη η S συμπεραίνουμε ότι

$$\frac{w_i}{p_i} \geq \frac{w_k}{p_k} \Leftrightarrow w_i p_k \geq p_i w_k \Leftrightarrow w_i p_k - p_i w_k \geq 0$$

Άρα ισχύει η ανισότητα που θέλαμε να δείξουμε.

Συνεπώς δείξαμε ότι $myTime(S, W) \leq time(S, W)$

Και για W = 0 παίρνω το αρχικό ζητούμενο $myTime(S, 0) \leq time(S, 0)$.

***: Χρησιμοποιούμε την myTime η οποία διαλέγει πάντα το στοιχείο της S με ελάχιστο wi/pi και θεωρούμε ότι η S είναι ταξινομημένη με τον τρόπο που δείξαμε πάνω, οπότε για αυτό διαλέχτηκε το (wn,pn).

Μια βέλτιστη δρομολόγηση λοιπόν είναι: $S' = ((w'_1, p'_1), (w'_2, p'_2), \dots, (w'_n, p'_n))$ με $\frac{w'_1}{p'_1} \geq \frac{w'_2}{p'_2} \geq \dots \geq \frac{w'_n}{p'_n}$

Παρακάτω δείχνουμε τον αλγόριθμο στον οποίο ως sort μπορεί κανείς να χρησιμοποιήσει οποιαδήποτε από τις γνωστές συναρτήσεις ταξινόμησης. Η πολυπλοκότητα θα είναι ίδια με την πολυπλοκότητα της αντίστοιχης sort. Θεωρούμε ότι εξυπηρετούμε πρώτα τον πελάτη με τον μικρότερο δείκτη στον πίνακα/λίστα (ή οποιαδήποτε δομή χρησιμοποιούμε) του αποτελέσματος (Result).

```
*****************************************
1   Αλγόριθμος:
2
3   # Η είσοδος είναι αυτής της δομής
4   S = [(w1,p1),(w2,p2),...,(wn,pn)]
5
6   # μια συνάρτηση σύγκρισης 2 στοιχείων
7   # της εισόδου
8   fun mycomparison((a,b),(c,d)):
9       return (a/b) >= (c/d)
10
11  # θεωρούμε ότι όταν στην sort δώσουμε
12  # συνάρτηση σύγκρισης <= τότε ταξινομεί
13  # σε αύξουσα σειρά ενώ όταν δώσουμε
14  # >= ταξινομεί την S σε φθίνουσα σειρά
15  # ως προς το μέγεθος που έχουμε ορίσει
16  # ως κριτήριο ταξινόμησης
17  Result = sort(S, mycomparison)
18
19  return Result
*****************************************
```

2. Με βάση και την αναδρομική σχέση του πρώτου ερωτήματος μπορούμε να γράψουμε:

$$time(X,L,R) = \begin{cases} 0, \alpha\nu\ X = \emptyset \\ \min\limits_{\forall (w_k,p_k)\epsilon X} \begin{Bmatrix} p_k(w_k + L) + time(X_k, L + w_k, R) \\ p_k(w_k + R) + time(X_k, L, R + w_k) \end{Bmatrix} \end{cases} \text{όπου } X_k = X \setminus (w_k, p_k), \alpha\nu\ X \neq \emptyset$$

Θεωρούμε L = Left και R = Right (έστω ότι οι δύο υπάλληλοί μας είναι Left και Right). Ο ελάχιστος συνολικός βεβαρυμμένος χρόνος εξυπηρέτησης των πελατών θα είναι $time(S, 0,0)$ όπου $S = ((w_1, p_1), (w_2, p_2), \dots, (w_n, p_n))$.

Έστω ότι έχουμε πετύχει έναν διαχωρισμό των πελατών σε υπακολουθίες της S με ονόματα SL, SR που να δίνει τον ελάχιστο συνολικό βεβαρυμμένο χρόνο. Έστω ότι κάποια από τις SL, SR και υποθέτουμε χωρίς βλάβη της γενικότητας ότι είναι η SL έχει τους πελάτες της δρομολογημένους με τέτοιον τρόπο ώστε να μην τους εξυπηρετεί στον ελάχιστο δυνατό χρόνο. Ο υπάλληλος Left, τότε απλά αναδιατάσσοντας (εσωτερικά στην SL) τα στοιχεία της θα πετυχαίναμε μία SL' με μικρότερο βεβαρυμμένο χρόνο εξυπηρέτησης και, ως αποτέλεσμα, θα μειωνόταν ο συνολικός βεβαρυμμένος χρόνος εξυπηρέτησης για όλο το σύστημα. Πράγμα που οδηγεί σε άτοπο γιατί είχαμε υποθέσει ότι είχαμε καταφέρει να ελαχιστοποιήσουμε τον χρόνο αυτόν. Άρα οι SL, SR είναι τέτοιες ώστε να πετυχαίνουν για τον υπάλληλό τους τον ελάχιστο βεβαρυμμένο χρόνο εξυπηρέτησης. Όπως είδαμε από το ερώτημα 1 ένας τρόπος να το πετύχουμε αυτό είναι να ταξινομήσουμε τις SL, SR ως προς το wi/pi σε φθίνουσα σειρά.

$$t(\emptyset, L, R) = 0$$

$$t(X, L, R) = min\begin{Bmatrix} p_k(w_k + L) + t(X_k, L + w_k, R) \\ p_k(w_k + R) + t(X_k, L, R + w_k) \end{Bmatrix} \text{όπου } X_k = X \setminus \{(w_k, p_k)\} \text{ και}$$

$$(w_k, p_k) \text{ το στοιχείο με το μικρότερο } \frac{w_k}{p_k} \text{ στην } X, \text{αν } X \neq \emptyset$$

Ξέρουμε ότι στην σωστή λύση η ακολουθία πελάτων κάθε υπαλλήλου θα είναι φθίνουσα ως προς wi/pi και η παραπάνω αναδρομή θα ψάξει σε όλες τις περιπτώσεις στις οποίες συμβαίνει αυτή η διάταξη για λύση άρα θα ψάξει και τουλάχιστον μία περίπτωση που μας δίνει τον ελάχιστο χρόνο άρα θα βρει λύση. Ο ελάχιστος χρόνος θα είναι $t(S, 0,0)$ με $S = ((w_1, p_1), (w_2, p_2), \dots, (w_n, p_n))$.

Πρέπει να υλοποιήσουμε την αναδρομή με αποδοτικό τρόπο ο οποίος αποθηκεύει και την ακολουθία που τελικά οδήγησε στον βέλτιστο για εμάς χρόνο.

****

```
Αλγόριθμος:

# Η είσοδος είναι ένας τέτοιος πίνακας
S = [(w1,p1,1),(w2,p2,2),...,(wn,pn,n)]
# (wk,pk,k) είναι ένα struct με w = wk,
#                               p = pk,
#                               name = k

W = sum(S[:].w) # το άθροισμα όλων των βαρών

fun mycomparison((a,b,_),(c,d,_)):
    return (a/b) >= (c/d)

# Ο αλγόριθμός μας χρειάζεται την είσοδο
# ταξινομημένη κατά φθίνον wi/pi
S = sort(S,mycomparison)

Memo # είναι ένας πίνακας που θα χρησιμοποιήσουμε για να
     # κρατάμε τιμές του schedule που έχουμε ήδη βρει, ώστε
     # να αποφεύγουμε επανα-υπολογισμός. Το στοιχείο
     # Memo[k][L][R] = (value,side) μας λέει ότι η δουλειά
     # που βρίσκεται στην θέση k στην S, αν οι υπάλληλοι Left
     # και Right έχουν συσσωρευμένα βάρη L, R μέχρι στιγμής
     # θα δρομολογηθεί σε αυτόν που βρίσκεται στο side (left --> 0
     # και right --> 1) με τιμή συνολικού χρόνου για το σύστημα
     # ίση με value.
# ο Memo έχει διαστάσεις (n+1)x(W+1)x(W+1)
def schedule(k,L,R):
    if (k == 0):
        return 0
    if Memo[k,L,R] exists:
        return Memo[k,L,R].value
    w = S[k].w
    p = S[k].p

    left = p * (w + L) + schedule(k - 1,w + L, R)
    right = p * (w + R) + schedule(k - 1, L, w + R)

```

```
        if (left <= right):
            Memo[k,R,L].value = Memo[k,L,R].value = left
            Memo[k,R,L].side = not (Memo[k,L,R].side = 0)
            # το πρόβλημα είναι συμμετρικό ως προς τους 2
            # υπαλλήλους, οπότε αποθηκεύουμε την τιμή και
            # για την συμμετρική ως προς R, L κατάσταση
            return left
        else:
            Memo[k,R,L].value = Memo[k,L,R].value = right
            Memo[k,R,L].side = not (Memo[k,L,R].side = 1)
            return right

    min_time = schedule(n,0,0) # βρίσκει τον ελάχιστο χρόνο

    # θα βάλουμε στις λίστες αυτές ποιος πάει
    # αριστερά και ποιος δεξιά
    Left = []
    Right = []

    # Γεμίζουμε τις Left, Right από τον τελευταίο προς τον
    # πρώτο άρα στο τέλος θα περιέχουν τους πελάτες κάθε
    # υπαλλήλου με τη σωστή σειρά.
    L, R, side = 0, 0
    for i in range(n,0):
        side = Memo[i,L,R].side
        name = S[i].name
        w = S[i].w
        p = S[i].p
        if (side == 0):
            Left.push(name)
            L = L + w
        else:
            Right.push(name)
            R = R + w
    return Left, Right
    # Μια σημείωση είναι ότι στους πίνακες/λίστες έχουμε θεωρήσει εύρος
    # τιμών για τους δείκτες από 1 μέχρι κάποια τιμή, ενώ για τον Memo
    # επιτρέπουμε τα k,L,R να πάρουν και τιμές ίσες με το μηδέν.
**********************************************************************
```

Ο παραπάνω αλγόριθμος υλοποιεί την αναδρομή αυτή και χρησιμοποιεί memoisation. Στον πίνακα Memo κρατάμε σε κάθε στοιχείο του την επιλογή μας (αριστερά ή δεξιά) ώστε μετά να μπορούμε να βρούμε σε ποιον υπάλληλο πήγε ο κάθε πελάτης μας. Η πολυπλοκότητα είναι O(n*W^2) και το W, αφού είναι το άθροισμα n βαρών, θα είναι σίγουρα μεγαλύτερο ή ίσο του n. Ο αλγόριθμος είναι ψευδοπολυωνυμικός. Βέβαια, στον πίνακα Memo δεν θα γεμίσουμε όλες τις θέσεις του με τιμές αφού οι τριπλέτες (k,L,R) εξαρτώνται από τα βάρη των παραλαβών, οπότε μάλλον θα χρειαστούμε λιγότερες πράξεις από τις παραπάνω στη μέση περίπτωση.

Στις ενημερώσεις των τιμών του πίνακα Memo εκμεταλλευόμαστε την συμμετρία του προβλήματος και αποθηκεύουμε και τις αντίστοιχες τιμές για το πρόβλημα που ανταλλάσσει τις παραλαβές των δύο υπαλλήλων μας. Θεωρούμε ότι το not 1 = 0 και το not 0 = 1, ώστε εύκολα να αντιστρέφουμε την ανάθεση της παραλαβής σε κάποιον υπάλληλο αφού Left → 0 και Right → 1.

Αν έχουμε m > 2 υπαλλήλους τότε στην αναδρομή δίνουμε έναν πίνακα με τα μεγέθη των μέχρι τώρα συσσωρευμένων βαρών (θα είναι m αριθμοί) και μετά θα κρατάμε το ελάχιστο από τις αναθέσεις της παραλαβής αυτής σε οποιονδήποτε υπάλληλό μας.

$$time(X, W) = \min_{1 \le i \le m} \{p_k\big(w_k + W.get(i)\big) + time(X_k, W.update(i, w_k + W.get(i))\}$$

$$\text{όπου } X_k = X \setminus (w_k, p_k) \text{ και } (w_k, p_k) \text{ το στοιχείο με το μικρότερο } \frac{w_k}{p_k} \text{ στην } X, \text{αν } X \neq \emptyset$$

$$time(\emptyset, W) = 0$$

Για να πάρουμε την απάντηση για την ακολουθία S καλούμε time(S,W) με όλα τα στοιχεία του W να είναι μηδενικά.

Θεωρούμε ότι η W.get(i) επιστρέφει το i-οστό στοιχείο του W ενώ η W.update(i,Value) ενημερώνει την i-οστη τιμή του W στην τιμή Value επιστρέφοντας έναν καινούριο πίνακα W μετά την ενημέρωση χωρίς να επηρεάζει τον αρχικό. Στο πρόβλημα με τους 2 υπαλλήλους αποθηκεύαμε στον πίνακα memoisation και την επιλογή μας ώστε να μην χρειαστεί να κάνουμε backtracking εκ των υστέρων για να δούμε ποιος υπάλληλος πήρε ποια εργασία. Εδώ μάλλον δεν μας βοηθάει ιδιαίτερα αυτό. Συνεπώς, Θα εκτελέσουμε ένα απλό backtracking το οποίο κάθε φορά θα παίρνει την τιμή της βέλτισης λύσης την οποία γνωρίζουμε, θα αφαιρεί το στοιχείο που δρομολογήσαμε, έπειτα θα θεωρεί ως τιμή βέλτισης λύσης αυτήν που βρήκε. Θα εξετάζει στον πίνακα memo σε ποιο κελί βρίσκεται η τιμή αυτή και έτσι θα βρίσκει ποιος υπάλληλος εξυπηρετεί την εργασία αυτή.

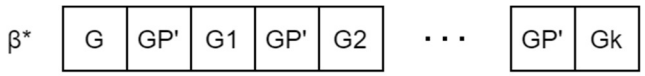
Άσκηση 5: Ελάχιστη Διαταραχή Ακολουθίας

1. Έστω ότι έχουμε βρει την βέλτιστη ακολουθία β*. Αν σε αυτήν την ακολουθία το τελευταίο στοιχείο είναι το μέγιστο (M) ή το ελάχιστό (m) της τότε δεν χρειάζεται να δείξουμε κάτι.

Έστω ότι το τελευταίο της στοιχείο δεν είναι το M ή το m. (A)

M – m είναι η μεγαλύτερη διαφορά 2 οποιονδήποτε στοιχείων της β*, αφού M, m είναι το μέγιστο και το ελάχιστο της β*. (B)

Έστω First το μέγιστο πρόθεμα της β* στο οποίο δεν περιέχεται m ή M. Τότε, αν το επόμενό του στοιχείο είναι το P αυτό θα είναι το είτε μέγιστο είτε ελάχιστο. Θεωρούμε P' = (P == m)?(M):(m). Διατρέχουμε την ακολουθία μέχρι το στοιχείο πριν την πρώτη εμφάνιση του P'. Ονομάζουμε G το πρόθεμα της β* που έχουμε διατρέξει μέχρι στιγμής. Διατρέχουμε την υπόλοιπη ακολουθία ονομάζοντας GP' (Groups of P') τα τμήματα της β* συνεχόμενων P' τα οποία συναντάμε και με G1, G2, …, Gk τα τμήματα στοιχείων της β* στα οποία θα ισχύει ότι κανένα τους στοιχείο δεν είναι ίσο με το P' και βρίσκονται ενδιάμεσα από τα GP', ενώ το Gk θα βρίσκεται στο τέλος της ακολουθίας β*. Παίρνουμε την ακολουθία γ από την β τοποθετώντας στο τέλος όλα τα GP':



Το G περιέχει το P αλλά όχι το P'. Τα G1, G2, …, Gk δεν περιέχουν το P'. Τα GP' είναι ομάδες αποκλειστικά από P'. Τα προθέματα που τελειώνουν μέσα στο G έχουν ίδιες διαταραχές και στην β* και στην γ. Τα προθέματα που τελειώνουν μέσα σε GP' έχουν διαταραχή M – m και στην β* και στην γ.

Έστω ένα πρόθεμα που τελειώνει μέσα σε κάποιο Gj. Στην β* έχει διαταραχή M – m αφού προηγούνται αυτού το G και τουλάχιστον ένα GP'. Στην γ το πρόθεμα αυτό δεν περιέχει το P' λόγω το τρόπου κατασκευής της γ άρα η διαταραχή του είναι μικρότερη από M – m (B). Συνεπώς, η γ έχει μικρότερη συνολική διαταραχή. Βέβαια, αυτό υποθέτει ότι υπάρχει τουλάχιστον ένα μη κενό GP' και μετά από αυτό υπάρχει τουλάχιστον ένα μη κενό Gj. Από (A) το Gk είναι αναγκαστικά μη κενό και επομένως το P' πρέπει να εμφανίζεται κάπου μέσα στην β* άρα θα υπάρχει και ένα GP'.

Καταφέραμε να φτιάξουμε μια ακολουθία με μικρότερη διαταραχή από την β* το οποίο είναι άτοπο. Δεν γίνεται το τελευταίο στοιχείο της β* να μην είναι το μέγιστο ή το ελάχιστο.

2. Θα δείξουμε ότι αν β* μια βέλτιστη ακολουθία με n στοιχεία τότε η δ = β* χωρίς το τελευταίο στοιχείο, έστω χ, πρέπει να είναι βέλτιστη. Έστω M, m το μέγιστο και ελάχιστο της β* αντίστοιχα.

Έστω δ1 ένας τρόπος να οργανώσουμε τα στοιχεία της δ ώστε να έχουμε την ελάχιστη συνολική διαταραχή (δ1 είναι μια βέλτιστη) και δ2 ένας άλλος τρόπος αναδιάταξής τους με μεγαλύτερη συνολική διαταραχή. Τώρα βάζουμε στο τέλος το στοιχείο χ και παίρνουμε τις β1 και β2. Η διαταραχή της β1 είναι η διαταραχή της δ1 + (M – m) και η διαταραχή της β2 είναι η διαταραχή της δ2 + (M – m). Άρα η διαταραχή της β1 είναι μικρότερη από την διαταραχή της β2. Η διαταραχή της β1 είναι όμως μικρότερη από την διαταραχή της β2 με οποιονδήποτε μη βέλτιστο τρόπο οργανώσουμε την δ2. Άρα η β1 πετυχαίνει την ελάχιστη διαταραχή και είναι βέλτιστη.

Αυτήν την πληροφορία την χρειαζόμαστε ώστε να μπορούμε να πούμε το εξής: Αν β* βέλτιστη ακολουθία τότε από ερώτημα 1 έχει τελευταίο της στοιχείο το μέγιστο ή το ελάχιστό της, αλλά και κάθε πρόθεμά της πρέπει να έχει ως τελευταίο του στοιχείο το μέγιστο ή το ελάχιστό του, γιατί είναι και αυτό βέλτιστο.

Μια αναδρομική σχέση που μας δίνει την ελάχιστη συνολική διαταραχή οποιασδήποτε αναδιάταξης των στοιχείων μιας ακολουθίας S είναι η παρακάτω:

$$Dist(S) = \begin{cases} 0, \alpha\nu\ |S| = 1 \\ \min\limits_{x \in \{minS, maxS\}} \{maxS - minS + Dist(S \setminus \{x\})\}, \alpha\nu\ |S| > 1 \end{cases}$$

Θα εκφράσουμε την αναδρομή αυτή με τρόπο που να μας διευκολύνει να γράψουμε αλγόριθμο:

$$S = (a1 \leq a2 \leq \cdots \leq an)\ \mu\iota\alpha\ \alpha\acute{\upsilon}\xi ουσα\ \alpha\kappa ολουθ\acute{\iota}\alpha\ n\ \sigma τοιχε\acute{\iota}ων$$

Τότε $Dist(S) = solv(1, n)$ όπου:

$$solv(i, j) = \begin{cases} 0, \alpha\nu\ i = j \\ aj - ai + \min\{solv(i, j-1), solv(i+1, j)\}, \alpha\nu\ i < j \end{cases}$$

Ο δείκτης i προσδιορίζει το ελάχιστο και ο δείκτης j το μέγιστο των στοιχείων της S που δεν έχουμε αποκλείσει ακόμα. «Αποκλείουμε» ένα στοιχείο ak σημαίνει ότι το βάλαμε στην κατάλληλη θέση στον ακολουθία β* που ψάχνουμε. Αν τα i και j είναι ίσα μεταξύ τους τότε έχει μείνει μόνο ένα στοιχείο στο σύνολο αριθμών που εξετάζουμε άρα επιστρέφουμε 0 σε συμφωνία και με την αναδρομή Dist(S). Ξέρουμε ότι κάποια στιγμή τα i και j θα γίνουν ίσα γιατί ξεκινούν με τιμές 1, n αντίστοιχα και το i αυξάνει μοναδιαία ή το j μειώνεται μοναδιαία σε κάθε βήμα (μόνο ένα από τα δύο θα συμβεί) κατά το οποίο είναι διαφορετικά μεταξύ τους. Δεν γίνεται το ένα να ξεπεράσει το άλλο.

Ο παρακάτω αλγόριθμος υλοποιεί την αναδρομή solv, η οποία έχουμε δείξει ότι δίνει σωστό αποτέλεσμα.

Εδώ κατασκευάζουμε την β* από το τέλος προς την αρχή, οπότε όπου αναφέρουμε ότι βάζουμε ένα στοιχείο στην β* εννοούμε ότι το κάνουμε σε συμφωνία με το παραπάνω.

```
**************************************************************
1       Αλγόριθμος:
2
3       S # ακολουθία εισόδου
4
5       S = sort(S)
6
7       Memo # πίνακας που χρησιμεύει για το memoisation
8       # Memo[i][j] = (value, k), όπου value = η τιμή
9       # της συνολικής διαταραχής της ακολουθίας με στοιχεία
10      # από το S[i] έως το S[j] και k σημαίνει ότι:
11      # αν k = -1 τότε έβαλα στην β* το μέγιστο ενώ αν
12      # k = 1 τότε έβαλα στην β* το ελάχιστο
13
14      def solv(i,j):
15          if (i == j): return 0
16          if(Memo[i][j] exists):
17              return Memo[i][j].value
18
19          # "Βάλαμε το μέγιστο στην ακολουθία"
20          Option_Max = S[j] - S[i] + solv(i,j-1)
21          # "Βάλαμε το ελάχιστο στην ακολουθία"
22          Option_Min = S[j] - S[i] + solv(i+1,j)
23
24          if (Option_Max <= Option_Min):
25              Memo[i][j].value = Option_Max
26              Memo[i][j].next_el = -1
27          else:
28              Memo[i][j].value = Option_Min
29              Memo[i][j].next_el = 1
30          return Memo[i][j].value
31
32      D = solv(1,n) # Το D θα είναι η καλύτερη διαταραχή
33      # και η κλήση αυτή θα μας φτιάξει τον πίνακα Memo.
34
35      b_star = [] # εδώ θα βάλουμε την απάντηση
36
37      # Στον παρακάτω βρόχο παιρνάμε τα κατάλληλα
38      # στοιχεία του πίνακα Memo και ενημερώνουμε
39      # την λίστα του αποτελέσματός μας
40      i, j = 1, n
41      while(True):
42          if (i == j):
43              b_star.push_front(S[i])
44              break
45          if(Memo[i][j].k == -1):
46              b_star.push_front(S[j])
47              j -= 1
48          else:
49              b_star.push_front(S[i])
50              i += 1
51
52      return D,b_star
**************************************************************
```

Ορθότητα:

Αρχικά κάνουμε ταξινόμηση στην είσοδο, ώστε να μπορούμε να εκτελέσουμε την αναδρομή solv που προϋποθέτει ταξινομημένα στοιχεία. Υποθέτουμε ότι η sort ταξινομεί σε αύξουσα σειρά.

Ο πίνακας Memo είναι διαστάσεων nxn, αφού θέλουμε να αποθηκεύουμε τιμές με Memo[i][j] με i < j.

Η solv: Συγκρίνει τις τιμές του αν αποκλείσουμε από τα υπολειπόμενα στοιχεία το μέγιστο (μείωση μονάδας από το j, αφού στην επόμενη κλήση το aj, το οποίο είναι το μέγιστο, δεν εμπεριέχεται στα στοιχεία που εξετάζουμε) ή αν αποκλείσουμε από τα υπολειπόμενα στοιχεία το ελάχιστο (προσθήκη μονάδας στο i, αφού στην επόμενη κλήση το ai, το οποίο είναι το ελάχιστο, δεν εμπεριέχεται στα στοιχεία που εξετάζουμε) και κρατάει την μικρότερη τιμή αποθηκεύοντας όμως στο κατάλληλο στοιχείο του πίνακα Memo και την επιλογή μας για αποκλεισμό του μεγίστου ή ελαχίστου.

Δείξαμε ότι η solv(i,j) υλοποιεί ορθά την αναδρομή. Έχουμε προσθέσει μόνο στον πίνακα Memo την ικανότητα να θυμάται την επιλογή μας μεταξύ μεγίστου και ελαχίστου. Έπειτα, υπολογίζουμε την ελάχιστη διαταραχή καλώντας την solv(1,n), δηλαδή, έχοντας αρχικά αποκλείσει κανένα στοιχείο του πίνακα S.

Όπως δείξαμε στην αιτιολόγηση της ορθότητας της solv η αφαίρεση μονάδας από το j ισοδυναμεί με προσθήκη του μεγίστου στην β* ενώ προσθήκη μονάδας στο i ισοδυναμεί με προσθήκη του ελαχίστου στην β*. Εδώ είναι σημαντικό να σημειώσουμε ότι η αναδρομή αυτή συμπεριφέρεται σαν να εξετάζει τα στοιχεία της β* από το τέλος της προς την αρχή της, οπότε όταν προσπαθήσουμε να την κατασκευάσουμε εκκινώντας από μια κενή λίστα τότε αν κάθε φορά τοποθετούμε το στοιχείο, που βρήκαμε ότι η solv απέκλεισε, στην κεφαλή της τότε θα την κατασκευάσουμε όντως από το τέλος προς την αρχή άρα θα πάρουμε όντως την β*.

Με βάση την παρατήρηση αυτή ο βρόχος επανάληψης που έχουμε στο τέλος του ψευδοκώδικα κατασκευάζει την ακολουθία β*.

Χωρική πολυπλοκότητα: S → O(n), Memo → O(n^2) άρα είναι $O(n^2)$

Χρονική πολυπλοκότητα:
Sorting → O(nlogn)
solv(1,n) → O(n^2)

Στην χειρότερη περίπτωση θα γίνουν αρκετές επαναλήψεις ώστε να γεμίσουμε με τιμές όλες τις θέσεις του πίνακα Memo (με i < j) δηλαδή O(n^2) επαναλήψεις. Σε κάθε επανάληψη οι υπολογισμοί των Option_Max και Option_Min γίνονται σε σταθερό χρόνο λόγω του memoisation και μετά εκτελούμε μια σύγκριση και αναθέσεις τιμών σε πίνακα που γίνονται σε σταθερό χρόνο άρα O(n^2) χρόνος.
Στο τέλος ο βρόγχος επανάληψης που εκτελούμε θα βάλει κάθε στοιχείο της S στην β* κάνοντας σταθερό χρόνο για την κάθε τοποθέτηση και εκτελώντας μία επανάληψη για καθένα από αυτά άρα O(n) χρόνος.

Συνολικά, η χρονική πολυπλοκότητα είναι, λοιπόν, $O(n^2)$.