1η Σειρά Ασκήσεων, Αλγόριθμοι και Πολυπλοκότητα

Πέτρος Αυγερίνος 03115074

7 Νοεμβρίου, 2023

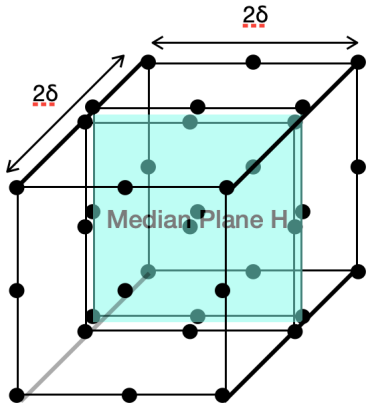
Contents

1 Πλησιέστερο Ζεύγος Σημείων

α. Για την εύρεση του πλησιέστερου ζέυγους σημείων στον τρισδιάστατο χώρο θα εργαστούμε παρόμοια με τις δύο διαστάσεις. Αρχικά θα ταξινομήσουμε κατά x, y, z με κόστος $O(3nlog(n))$. Αργότερα μπορούμε να βρούμε μια επιφάνεια η οποία χωρίζει τον τρισδιάστατο χώρο σε δύο set P, Q μέσω του median ενός από τις τρείς διαστάσεις για τα σημεία αυτά, έστω x_{median}.

Έστω η επιφάνεια αυτή H και αναδρομικά βρίσκουμε δ_P, δ_Q, τις μικρότερες αποστάσεις μέσα σε κάθε υποπρόβλημα.Από τον αλγόριθμο των δύο διαστάσεων είδαμε ότι μπορούμε με πεπερασμένο αριθμό συγκρίσεων για ένα σημείο να βρούμε όλα τα πιθανά πλησιέστερα σημεία με $d(p_i, p_j) \leq \delta, \forall i \neq j$ με την χρήση ενός τετραγώνου διαστάσεων $(2 \cdot \delta)^2$. Στην περίπτωση του τρισδιάστατου χώρου μπορούμε παίρνοντας το $\delta = min(\delta_P, \delta_Q)$ να κατασκευάσουμε ένα τρισδιάστατο slab με μήκος 2δ γύρω από το H.

Χωρίζοντας το παραπάνω slab σε κύβους με όγκο $(2 \cdot \delta)^3$, κατά το μέγιστο μπορούμε μέσα σε αυτόν τον κύβο να έχουμε 9 σημεία σε απόσταση δ από το H που ανήκουν στο P, 9 σημεία πάνω στο H τα οποία ανήκουν στο P, 9 σημεία σε απόσταση δ από το H που ανήκουν στο Q και 9 σημεία πάνω στο H τα οποία ανήκουν στο Q. Επομένως για ένα σημείο $p \in P$ θα πρέπει να γίνουν συνολικά 35 έλεγχοι. για κάθε διάσταση y, z άρα σύνολο 70 έλεγχοι, καθώς με τις ταξινομήσεις έχουμε ουσιαστικά προβάλει τα σημεία μας πάνω στο H. Άρα στην χειρότερη περίπτωση όπου το slab εμπερίεχει όλα τα στοιχεία, $\frac{n}{2}$ σε κάθε set P, Q η συνολική πολυπλοκότητα για το κομμάτι του βασίλευε προκύπτει $O(70 \cdot n)$. Μπορούμε επίσης να μειώσουμε τις συγκρίσεις αρκεί να ανατρέξουμε μόνο ένα από τα P, Q με τα αντίστοιχα της άλλης μεριάς του slab. Αυτό όμως δεν έχει πραγματικό αντίκτυπο στην πολυπλοκότητά του αλγορίθμου.



Η συνολική πολυπλοκότητα του αλγορίθμου μας προκύπτει $O(n \cdot log(n)^3)$. Από Master Theorem προκύπτει $T(n) = 2 \cdot T(\frac{n}{2}) + O(n) = O(nlog(n))$.

Algorithm 1 Closest_Pair_3D($P = [p_i(x_i, y_i, z_i) \in 3D] : i \in [0, n], n$)

1: Sort by each dimension and create arrays X_s, Y_s, Z_s ▷ $O(n \cdot log(n)^3)$
2: δ= Find_Closest_Distance(X_s, Y_s, Z_s, n)
3: **return** δ

Algorithm 2 Find_Closest_Distance(X_s, Y_s, Z_s, n)

if $n == 2$ **then return** $d(p1, p2)$
if $n == 3$ **then return** $min(d_1, d_2, d_3)$
else
    $x_{median} = X_s[n/2]$
    $X_p = X_s[1 : n/2]$, $X_q = X_s[n/2 + 1 : n]$
    $k = 0$
    for $i \in [0, n]$ **do**
        if $Y_s.x_i \leq x_{median}$ **then**
            $Y_p.insert(P.y_i)$
        else
            $Y_q.insert(P.y_i)$
        if $Z_s.x_i \leq x_{median}$ **then**
            $Z_p.insert(P.z_i)$
        else
            $Z_q.insert(P.z_i)$
    $\delta_p = Find\_Closest\_Distance(X_p, Y_p, Z_p, n/2)$
    $\delta_q = Find\_Closest\_Distance(X_q, Y_q, Z_q, n/2)$
    $\delta = min(\delta_p, \delta_q)$ ▷ $2 \cdot T(\frac{n}{2})$
    Construct S' containing all points that are in distance δ from H plane.(assume k points) ▷ $O(n)$
    for $i \in [0, k]$ **do**
        for $j \in [i + 1, i + 35]$ **do**
            if $j \leq k$ **then**
                if $d(p_i, p_j) < \delta$ **then** $\delta = d(p_i, p_j)$ ▷ Finite, as proven above
    return δ

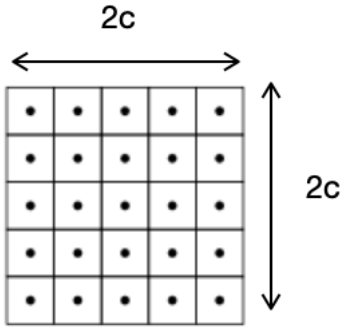
β. Γνωρίζουμε ότι

$$\delta^* \in [l, c \cdot l] \Rightarrow l \leq \delta^* \leq c \cdot l$$

όπου θα υπάρχουν δύο σημεία p_1^*, p_2^* για τα οποία θα ισχύει $d(p_1^*, p_2^*) = \delta^*$. Άρα:

$$\begin{gathered} l \leq d(p_1^*, p_2^*) \leq cl \Rightarrow \\ 1 \leq \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + \cdots + (d_1 - d_2)^2} \leq cl \Rightarrow \\ 1 \leq \sqrt{(\frac{x_1 - x_2}{l})^2 + (\frac{y_1 - y_2}{l})^2 + \cdots + (\frac{d_1 - d_2}{l})^2} \leq c \Rightarrow \\ 1 \leq d(p_1^l, p_2^l) \leq c. \end{gathered}$$

Αν $\forall p_i$ κανονικοποιήσουμε με τον παραπάνω τρόπο μπορούμε να δημιουργήσουμε ένα grid στο οποίο για τα σημεία μας θα ισχύει $1 \leq d(p_i^*, p_j^*) \forall i, j$ με $i \neq j$.



Έστω c μια ποσότητα ανάλογη της ποσότητας συγκρίσεων που θα κάνουμε για το παραπάνω grid για ένα σημείο p_i. Μπορούμε να εργαστούμε παρόμοια με το προηγούμενο ζητούμενο όπου ελέγχουμε υπερκύβους πλευράς $2 \cdot c$. Για δύο διαστάσεις είχαμε $(2 \cdot c)^2$, για τρεις $(2 \cdot c)^3$, για έναν υπερκύβο d διαστάσεων θα έχουμε τελικά $(2 \cdot c)^d$. Επομένως προκύπτει πεπερασμένος αριθμός συγκρίσεων για κάθε σημείο με πολυπλοκότητα $O((2 \cdot c)^d \cdot n)$. Έτσι μπορούμε σε γραμμικό χρόνο να βρούμε το πιο κοντινό ζευγάρι σημείων.

Algorithm 3 Closest_Pair_DD(P,n)

$\delta \to \infty$
for each dimension in a single P_i **do** ▷ $O(d \cdot n)$
    for $i \in [0, n]$ **do**
        Normalise p_i by l for said dimension
for $i \in [0, n]$ **do**
    Calculate $(2 \cdot c)^d$ distances for p_i ▷ $O((2 \cdot c)^d \cdot n)$
    if $d(p_i, p_j) < \delta$ **then**
        $\delta = d(p_i, p_j)$
return δ

2 Πόρτες Ασφαλείας στο Κάστρο

Ουσιαστικά αντιλαμβανόμαστε ότι αναζητούμε μια binary συμβολοσειρά n ψηφίων με την οποία θα είναι εφικτό να είναι ανοιχτές όλες οι πόρτες ταυτόχρονα ώστε να περάσουμε. Αν δοκιμάζαμε κάθε δυνατή επιλογή θα προέκυπταν 2^n συνδυασμοί το οποίο δεν είναι υπολογιστικά συνετό. Επομένως θα πρέπει να εργαστούμε διαφορετικά.
Αφού η μόνη γνώση που έχουμε στη διάθεση μας για τις διάφορες συμβολοσειρές που μπορούν να προκύψουν είναι για την τελευταία πόρτα που μένει κλειστή θα πέπει σειριακά για κάθε πόρτα να ψάξουμε την τιμή και τον διακοπτή για τον οποίο η πόρτα αυτή ανοίγει.
Για την εύρεση της πρώτης πόρτας, αρχικά θα οδηγήσουμε όλους τους διακόπτες σε μία τιμή, 0 ή 1, και αν η πόρτα ανοίξει με την κίνησή μας αυτή τότε γνωρίζουμε ότι ανοίγει με την τιμή που θέσαμε, αλλιώς με την άλλη τιμή. Πλέον λοιπόν γνωρίζουμε για ποια τιμή ανοίγει η πρώτη πόρτα. Αρκεί τώρα να βρούμε τον διακόπτη για τον οποίο συμβαίνει αυτό.
Για την εύρεση του διακόπτη ανοίγουμε $\frac{n}{2}$ διακόπτες και κλείνουμε τους υπόλοιπους. Αφού γνωρίζουμε για ποια τιμή ανοίγει η πόρτα με την διαμέριση των διακοπτών αυτή μπορούμε να αντιληφθούμε σε ποιό από τα δύο μισά θα πρέπει να εργαστούμε. Αν η πόρτα ανοίξει για τα μισά που θέσαμε στη τιμή για την οποία γνωρίζουμε ότι ανοίγει η πόρτα, τότε θα ψάξουμε αναδρομικά μέσα σε αυτό το σύνολο. Στην περίπτωση όπου κάνουμε λάθος ψάχνουμε το διπλανό σύνολο το οποίο έχει κόστος σταθερό. Επομένως με αυτή την αναδρομή προκύπτει κόστος για την εύρεση του διακόπτη της πρώτης πόρτας $\Theta(log(n))$.
Διατηρούμε τον διακόπτη της πρώτης πόρτας στη θέση που πρέπει για να ανοίγουν και προχωράμε αναδρομικά για την εύρεση της τιμή για την οποία ανοίγει η δεύτερη πόρτα, με την οδήγηση όλων των υπολοιπόμενων διακοπτών σε μία από τις τιμές 0 ή 1. Μόλις βρεθεί η τιμή μπορούμε πάλι με κόστος αυτή τη φορά $\Theta(log(n-1))$ να βρούμε ποιος διακόπτης είναι υπεύθυνος για την δεύτερη πόρτα. Αναδρομικά κάνω το ίδιο για την i-ιοστή πόρτα.

Συνολικά λοιπόν προκύπτει για τις συνολικές διαμορφώσεις: $\Delta(n) = \sum_{i=1}^{n} \Theta(log(n) - i + 1) = \Theta(n \cdot log(n))$

Algorithm 4 Castle_Doors(Doors,Switches)

for each Door in Doors **do** ▷ $\Theta(n)$
    Switch all Switches on (1) ▷ $\Theta(1)$
    if Door opens **then**
        Door.opens_at ← 1 **else**
        Door.opens_at ← 0 −
            Split the Switches in $\frac{n}{2}$ parts each
            Recursively check each part, given where the door opens ▷ $\Theta(log(n))$
            Save the position of the Switch that opened the Door

3 Κρυμμένος Θησαυρός

Αυτό το πρόβλημα είναι γνωστό και ως "Lost Cow Problem".

Ο τρόπος με τον οποίο θα λύσουμε το παρόν πρόβλημα είναι ο διπλασιασμός της απόστασης που διανύουμε από το 0 κάθε φορά, μία από τα δέξια και μία από τα αρίστερα σε σχηματισμό "zig zag". Ονομάζουμε την απόσταση που διανύω στην επανάληψη i του αλγορίθμου ως $r_i = 2^i$ όπου $i = \{1, 2, 3, \dots\}$ (Για το $i = 0$ καμία βάση ($a^0 = 1$) δεν μπορεί να αυξήσει την πληροφορία σχετικά με την εύρεση του θησαυρού, και δεν πρόκειται να βρεθεί στο διάστημα μέχρι το 1 και γι'αυτο δεν το συμπεριλαμβάνουμε στις επαναλήψεις μας).
Άρα είναι σαφές πως η απόσταση που θα διανύσω μέχρι να επιστρέψω στο 0 για εκείνη την επανάληψη θα είναι $2 \cdot r_i$. Για τα $imod2 = 1$ ακολουθώ το δεξί μονοπάτι και για τα $imod2 = 0$ το αριστερό. Έστω ότι βρίσκω τον θησαυρό σε απόσταση $|x|$ από το 0 στην $n + 2$ επανάληψη του αλγορίθμου. Επομένως η συνολική απόσταση που θα έχω διανύσει θα είναι:

$$D(n) = 2 + 2 + 4 + 4 + \cdots + r_{n+1} + r_{n+1} + |x| = \\ 2 \cdot \sum_{i=1}^{n+1} 2^i + |x| = \\ 2 \cdot (2^{n+2} - 2) + |x|$$

Όμως αφού ο θησαυρός βρέθηκε στην επανάληψη $n+2$, αυτό σημαίνει ότι η εύρεσή του διέκοψε τον αλγόριθμο σε κάποιο σημείο μεταξύ των αποστάσεων 2^n και 2^{n+2}, επομένως προκύπτει ότι $|x| = 2^n + d'$. Άρα:

$$D(n) = 2 \cdot (2^{n+2} - 2) + |x| = \\ 2 \cdot (2^{n+2} - 2) + 2^n + d' = \\ 2 \cdot (4 \cdot 2^n - 2) + 2^n + d' = \\ 9 \cdot 2^n - 4 + d' \leq 9 \cdot 2^n + d' \leq 9 \cdot (2^n + d') \Rightarrow \\ 9 \cdot (2^n + d') \leq 9 \cdot |x|$$

Βρήκαμε λοιπόν μία σταθερά $c = 9$ για την οποία ο αλγόριθμός μας έχει πολυπλοκότητα $O(|x|)$.
Για την επιλογή της βάσης της εκθετικής συνάρτησης βημάτων, θα αποδείξουμε ότι βέλτιστη βάση $a \in \mathbb{N}$ είναι για $a = 2$.
Έστω ότι η βάση αυτή είναι a. Επομένως το $D(a, n)$ για ένα $n >> a$ θα είναι της μορφής:

$$D(a, n) = 2 \cdot \sum_{i=1}^{n+1} a^i + |x| = 2 \cdot \sum_{i=1}^{n+1} a^i + a^n + d' = \\ 2 \cdot (\frac{a^{n+2} - 1}{a - 1} - 1) + a^n + d' \xRightarrow{n>>a} 2 \cdot a^{n+2} + a^n + d' = \\ (2a^2 + 1) \cdot a^n + d'$$

Η οποία είναι μια γνησίως αύξουσα συνάρτηση ως προς a με ελάχιστο ακέραιο δυνατό προς επιλογή $a = 2$ καθώς, για $a = 1$ θα κάναμε ταλάντωση γύρω από το μηδέν με μήκος 1.

Algorithm 5 Hidden_Treasure(X)

```
1: i ← 1
2: while Treasure is not found do
3:     Travel 2^i steps towards Path_{imod2}
4:     if Treasure is Found while travelling then return i        ▷ O(|x|)
5:     else
6:         Return to 0
7:         i ← i + 1
```

4 Μη Επικαλυπτόμενα Διαστήματα

α. Για την περίπτωση όπου χρησιμοποιούμε ως κριτήριο το μέτρο $|f_i - s_i|$ προκύπτει η εξής αναδρομική σχέση:

$$C^*(A) = max\{|f_i - s_i| + C^*(A_i)\},\ \text{όπου}\ A_i = \{j \in A : s_j \geq f_i\}$$

Ταξινομώντας λοιπόν τα $|f_i - s_i|$ σε φθήνουσα σειρά προκύπτει ότι:

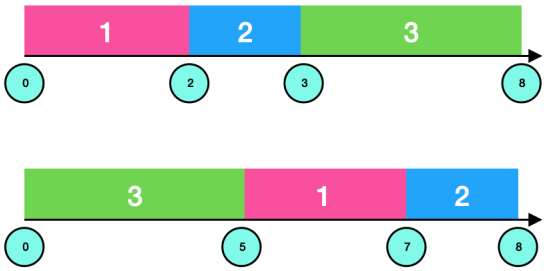
$$|f_1 - s_1| > |f_2 - s_2| > \cdots > |f_n - s_n| \text{ όπου } n \text{ το σύνολο των διαστημάτων}$$

Προκύπτει λοιπόν ο εξής αλγόριθμος:

Algorithm 6 *Sequences_1*(S)

```
Sort A via criterion |f_i − s_i|(max first)
j ← 1
C ← {1}
for i ← 2 to n do
    if s_i ≥ f_j then
        C ← C ∪ {i}
        j ← i
return C
```

Ας λάβουμε υπόψιν όμως το εξής παράδειγμα.



Ταξινομώντας τα 1,2,3 κατά φθήνουσα σειρά μέτρου, παίρνουμε πρώτο το μεγαλύτερο, στην πέριπτωσή μας το 3, το οποίο όμως είναι τελευταίο, επομένως ο αλγόριθμος θα τερματήσει χωρίς να συμπεριλάβει το 1 και 2 στην απόφασή του λόγω του ελέγχου $s_i \geq f_j$. Είναι φανερό όμως πως η βέλτιστη λύση δεν είναι μόνο το $\{3\}$ αλλά το $\{1, 2, 3\}$. Άρα είναι σαφές πως με αυτό το κριτήριο ο αγλόριθμός μας αποτυγχάνει.

Ταξινομώντας τώρα με βάση τα f_i σε αύξουσα σειρά, την σειρά δηλαδή με την οποία οι διεργασίες ολοκληρώνονται πρώτες κάθε φορά σε κάθε βήμα του αλγορίθμου, θα πάρουμε την βέλτιστη λύση η οποία είναι το σύνολο $\{1, 2, 3\}$. Ο αγλόριθμος λοιπόν παίρνει την εξής μορφή που είδαμε και στο μάθημα.

Algorithm 7 *Sequences_2*(S)

```
Sort A via criterion f_i(min first)
j ← 1
C ← {1}
for i ← 2 to n do
    if s_i ≥ f_j then
        C ← C ∪ {i}
        j ← i
return C
```

Και στις δύο περιπτώσεις λόγω ταξινόμησης θα προκύπτει πολυπλοκότητα $O(n \cdot log(n))$ λόγω ταξινόμησης.

β. Αρχικά θα ορίσουμε κάποια notations για το πρόβλημα ώστε να μπορούμε καλύτερα να το εκφράσουμε. Θέτω $\Delta(n)$ το σύνολο όλων των διαστημάτων για το πρόβλημα μου, όπου $\delta_i \in \Delta(n)$ είναι τα διαστήματα αυτά. Το δ_i εμπεριέχει πληροφορία σχετικά με την αρχή του sequence s_i, το τέλος f_i, το μέτρο $|f_i - s_i|$, αν χρειαστεί θα το επεκτείνουμε για τις ανάγκες του προβλήματός μας.

```
struct {
    int s; \\ start
    int f; \\ finish
    int m = f-s; \\ |f-s|
} sequence;
```

Ταξινομώ το σύνολο Δ κατά αύξουσα σειρά των f_i. Πλέον ο άπληστος αλγόριθμος δεν θα καταφέρει να μας δώσει τη λύση στο πρόβλημα, θα πρέπει να εργαστούμε με δυναμικό προγραμματισμό. Θα επιχειρήσουμε να κατασκευάσουμε μία αναδρομική σχέση η οποία για κάθε βήμα του αλγορίθμου θα εντάσει στο πρόβλημα ένα ακόμη διάστημα. Κάθε φορά που εντάσσουμε ένα διάστημα στο πρόβλημά μας θα επιλέγουμε την καλύτερη μεταξύ δύο περιπτώσεων, την περίπτωση αυτή δηλαδή που μεγιστοποιεί την συνάρτηση κόστους για το άθροισμα μη επικαλυπτώμενων διαστημάτων.

1. Αν το δ_i επιλέγεται στην λύση για i διαστήματα.
2. Αν το δ_i δεν επιλέγεται στην λύση για i διαστήματα.

Είναι σαφές πως για να διατηρώ την γνώση σχετικά με το πόσα διαστήματα έχουν εισαχθεί στο βήμα i και το συνολικό βάρος της βέλτιστης λύσης μέχρι εκείνο το σημείο θα πρέπει να αποθηκεύσω τη γνώση αυτή σε ένα πίνακα. Έστω ο πίνακας αυτός $A[n]$ ο οποίος σε κάθε θέση i περιέχει τη βέλτιστη λύση για την επιλογή διαστημάτων μέχρι εκείνο το σημείο. Θα χρειαστεί σε αυτό το σημείο μία προεργασία για την έυρεση των προηγούμενων μη επικαλυπτώμενων διαστημάτων $\forall \delta_i \in \Delta$ ώστε να μπορούμε να υλοποιήσουμε την αναδρομική σχέση που βρήκαμε παραπάνω. Για την εύρεση αυτών μπορούμε να κάνουμε δυαδική αναζήτηση για το s_i (την αρχή του διαστήματος για το οποίο ψάχνουμε τον αμέσως προηγούμενό του) μέσα στον ταξινομημένο κατά f_i πίνακα Δ και να πάρουμε το αμέσως μικρότερο στοιχείο από το s_i. Η πολυπλοκότητα αυτής της διαδικασίας είναι σαφώς $O(log(n))$, και για όλα τα n διαστήματα προκύπτει τελικά $O(n \cdot log(n))$. Μπορώ να αποθηκεύσω τον προηγούμενο αυτό μέσα στο ίδιο struct με πριν, επομένως το struct μου θα πάρει την εξής μορφή:

```
struct {
    int s; \\ start
    int f; \\ finish
    int m = f-s; \\ |f-s|
    int p; \\ index of previous
} sequence;
```

Με την παραπάνω διαδικασία μπορούμε τώρα να γράψουμε την αναδρομική σχέση του αλγορίθμου μας.

$$A[i] = \begin{cases} 0 & i = 0 \\ max\{\delta_i.m + A[\delta_i.p], A[i-1]\} & i \neq 0 \end{cases}$$

Με την παραπάνω αναδρομική σχέση γνωρίζουμε πως το τελευταίο στοιχείο του πίνακα A θα περιέχει την βέλτιστη λύση για την εισαγωγή όλων των διαστημάτων του Δ. Επομένως ο αλγόριθμός μας παίρνει την εξής μορφή.

Algorithm 8 *Optimal_Length*(Δ) ▷ $O(n \cdot log(n))$

1: $A[0] = 0$
2: Sort Δ by f_i in ascending order ▷ $O(n \cdot log(n))$
3: **for** $\delta \in \Delta$ **do** ▷ $O(n)$
4: Search for immediate previous f of $\delta.s$ ▷ $O(log(n))$
5: **for** $i \in [1, n]$ **do** $A[i] = max(s_i.m + A[s_i.p], A[i-1])$ ▷ $O(n)$
return $A[n]$

Καταφέραμε λοιπόν να βρούμε σε χρόνο $O(n \cdot log(n))$ το μέγιστο δυνατό μήκος για τα μη επικαλυπτώμενα διαστήματα. Θα αναζητήσουμε τώρα το σύνολο C των διαστημάτων του Δ για τα οποία ισχύει η βέλτιστη αυτή λύση.
Ξεκινώντας από το τέλος του πίνακα A για $i = n$ και αφαιρώντας ένα ένα τα διαστήματα μπορούμε, κάθε φορά που $A[i-1] < A[i]$ να συμπεράνουμε ότι το τελευταίο που αφαιρέσαμε συμμετέχει στην βέλτιστη λύση και έτσι να το τοποθετήσουμε στο σύνολο C. Όμως θα πρέπει να λάβουμε υπόψιν και το αμέσως προηγούμενο του διαστήματος δ_i καθώς η μείωση αυτή μπορεί να προκύψει και από αλλαγή συνόλου διαστήματων. Οπότε κάθε φορά που παρατηρείται μείωση, θα ελέγχουμε αν υπάρχει προηγούμενο μη επικαλυπτώμενο και αν υπάρχει θα συνεχίζουμε την αναζήτηση σε αυτό το προηγούμενο γλυτώνοντας ετσί κάποιους ελέγχους.

Algorithm 9 *Optimal_Solution*(n) ▷ $O(n)$

```
1: i = n
2: if i = 0 then return C
3: if s_i.m + A[s_i.p] > A[i − 1] then
4:     Append i to C
5:     i = i − 1
6:     Optimal_Solution(s_i.p)
7: else
8:     Optimal_Solution(i)
   return C
```

5 Παραλαβή Πακέτων

1. Για την επίλυση του παρόντος προβλήματος θα κάνουμε χρήση άπληστου αλγόριθμου με κριτήριο τον λόγο $\frac{w_i}{p_i}$. Ταξινομώντας λοιπόν $\forall i \in A$ τα $\frac{w_i}{p_i}$ σε φθήνουσα σειρά μπορούμε κάθε φορά να εξυπηρετούμε τον πελάτη εκείνο του οποίο ο λόγος σημαντικότητας προς χρόνο εξυπηρέτησης είναι μεγαλύτερος. Προκύπτει λοιπόν η εξής αναδρομική σχέση με την οποία θα εργαστούμε.

$$G(A) = (\text{Εξυπηρέτηση Πελάτη } i) : max\{\frac{w_i}{p_i} \forall i \in A\} + G(A' = A - \{i\})$$

Ας υποθέσουμε τώρα ότι η ταξινόμησή μας γίνεται ως εξής: $\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \cdots \geq \frac{w_n}{p_n}$. Άρα η λύση του άπληστου αλγορίθμου S θα είναι:

$$S = w_1 \cdot p_1 + w_2 \cdot (p_1 + p_2) + \cdots + w_n \cdot \sum_1^n p_i$$

Έστω ότι υπάρχει μία λύση η οποία είναι η βέλτιστη για την οποία μεταθέτω ένα w_i με το w_{i+1} όπου $i \in (1, n)$ και $\frac{w_i}{p_i} > \frac{w_{i+1}}{p_{i+1}}$. Έτσι προκύπτει ένα S' το οποίο θα είναι ως εξής:

$$S' = w_1 \cdot p_1 + w_2 \cdot (p_2 + p_2) + \cdots + w_{i+1} \cdot \sum_1^{i-1} p_k + w_{i+1} \cdot p_{i+1} + w_i \cdot \sum_1^{i+1} p_k + \cdots + w_n \cdot \sum_1^n p_i$$

Για τα S, S' μπορούμε να εργαστούμε ως εξής για να δούμε την ορθότητα του άπληστου αλγορίθμου:
Το κλάσμα $\frac{S}{S'} > 1$ αφού έχουμε υποθέσει ότι S' είναι βέλτιστη λύση άρα και ελάχιστη. Άρα:

$$S > S' \Rightarrow w_i \cdot \sum_1^i p_k + w_{i+1} \cdot \sum_1^{i+1} p_k > w_{i+1} \cdot \sum_1^{i-1} p_k + w_{i+1} \cdot p_{i+1} + w_i \cdot \sum_1^{i+i} p_k$$
$$\Rightarrow w_i \cdot (p_1 + p_2 + \ldots p_i) + w_{i+1} \cdot (p_1 + p_2 + \ldots p_{i+1}) > w_{i+1} \cdot (p_1 + p_2 + \ldots p_{i-1}) + w_i \cdot (p_1 + p_2 + \ldots p_{i+1}) + w_{i+1} \cdot p_{i+1}$$
$$\Rightarrow w_{i+1} \cdot p_i > w_i \cdot p_{i+1}$$
$$\Rightarrow \frac{w_{i+1}}{p_{i+1}} > \frac{w_i}{p_i}$$

Το οποίο σαφώς δεν ισχύει καθώς γνωρίζω ότι $\frac{w_i}{p_i} > \frac{w_{i+1}}{p_{i+1}}$, $\forall i \in A$. Άρα ο G(A) βρίσκει όντως βέλτιστη λύση. Η πολυπλοκότητα του συγκεκριμένου άπληστου αλγορίθμου προκύπτει $O(n \cdot log(n))$ λόγω ταξινόμησης.

2. Πλέον η χρήση άπληστου αλγορίθμου δεν θα καταφέρει να λύσει το πρόβλημα, επομένως θα εργαστούμε ξανά με δυναμικό προγραμματισμό. Ας δούμε αρχικά την περίπτωση δύο υπαλλήλων. Θα αποφασίζουμε κάθε φορά ποιος υπάλληλος θα εξυπηρετήσει τον επόμενο πελάτη ελέγχοντας τις τιμές για τις δύο περιπτώσεις και παίρνοντας κάθε φορά την μικρότερη τιμή, αφού πρώτα ταξινομήσουμε βέβαια τους πελάτες σύμφωνα με τον λόγο $\frac{w}{p}$ σε φθήνουσα σειρά όπως είδαμε στο προηγούμενο ερώτημα.

1. Η περίπτωση να αναλάβει ο υπάλληλος 1 τον επόμενο πελάτη.
2. Η περίπτωση να αναλάβει ο υπάλληλος 2 τον επόμενο πελάτη.

Έστω ότι ο υπάλληλος 1 θα εξυπηρετήσει ένα σύνολο C_1 πελατών. Αν οι πελάτες είναι στο σύνολο C τότε ο υπάλληλος 2 θα εξυπηρετήσει το σύνολο $C_2 = C - C_1$. Αφού θα διατάξουμε τους πελάτες πριν γίνει η εξυπηρέτηση μπορούμε να ελέγξουμε για οποιοδήποτε πιθανό χρόνο εξυπηρέτησης που θα κάνει ο υπάλληλος 1 (p_{c_1}) ποιος χρόνος θα προκύψει για τον υπάλληλο 2.

Επομένως προκύπτει το συμπέρασμα ότι χρειάζεται πάλι η χρήση μιας αναδρομικής συνάρτησης $A(n, p_{c_1})$ όπου στην εισαγωγή κάθε πελάτη σε κάποιο ταμείο να προστίθεται σε αυτή την συνάρτηση στη θέση i για το βήμα i (δεδομένου των προηγούμενων εισαγωγών και ενός χρόνου p_{c_1}, τον χρόνο δηλαδή που θα εξυπηρετεί ο Υπάλληλος 1) η βέλτιστη τιμή για εκείνο το βήμα. Η λύση με χρήση της αναδρομικής αυτής σχέσης θα είναι για το n-ιοστό βήμα του αλγορίθμου για κάποιο χρόνο p_{c_1} όπου προκύπτει η ελάχιστη τιμή για την συνάρτηση A.

Προκύπτει λοιπόν η εξής αναδρομική σχέση:

$$A(i, p_{c_1}) = \begin{cases} 0 & i = 0 \\ min(A(i-1, p_{c_1} - p_i) + w_i \cdot p_{c_1}, A(i-1, p_{c_1}) + w_i \cdot (\sum_0^i p_k - p_{c_1})) & i \neq 0 \end{cases}$$

Τελικά η λύση του αλγορίθμου μας θα είναι $\min_i \{A(n, p_{c_1|i})\}$

Να σημειωθεί ότι $\forall j > i \Rightarrow A(i, p_j) = 0$ γιατί δεν θα είναι εφικτό σε ένα βήμα i του αλγορίθμου να εισάγουμε έναν πελάτη j καθώς ακολουθούμε την ταξινόμηση κατά $\frac{w}{p}$. Αφού η συνάρτηση αυτή χρησιμοποιεί μνήμη θα χρειαστεί μόνο μία φορά να διατρέξουμε κάθε πελάτη, και μετά για κάθε πελάτη θα αυξόνται οι δυνατοί χρόνοι p_{c_1} για τους οποίους θα πρέπει να βρούμε κάθε δυνατή επιλογή. Η μνήμη του αλγορίθμου θα εξυπηρετήσει στο να μην κάνουμε πολλαπλές φορές αυτές τις πράξεις. Συνολικά θα βρούμε $Subset\_Sum(p_i)$ πιθανούς χρόνους για το p_{c_1}. Η πολυπλοκότητα του αλγορίθμου λοιπόν θα προκύψει $O(n \cdot Subset\_Sum(p_i))$.

Για m υπαλλήλους θα χρειαστεί να ελέγξω όλους τους πιθανούς χρόνους για $m-1$ υπαλλήλους επομένως η πολυπλοκότητα προκύπτει $O(n \cdot Subset\_Sum(p_i)^{m-1})$.