Αλγόριθμοι και Πολυπλοκότητα
1η Σειρά Γραπτών Ασκήσεων

Ελισάβετ Παπαδοπούλου,
Α.Μ.: 03120190

Άσκηση 1: Πλησιέστερο Ζεύγος Σημείων

(a) Το πρόβλημα του πλησιέστερου ζεύγους n σημείων $(x_1, y_1, z_1), ..., (x_n, y_n, z_n)$ στις 3 διαστάσεις προκύπτει από την γενίκευση του αλγορίθμου στις 2 διαστάσεις ως εξής:

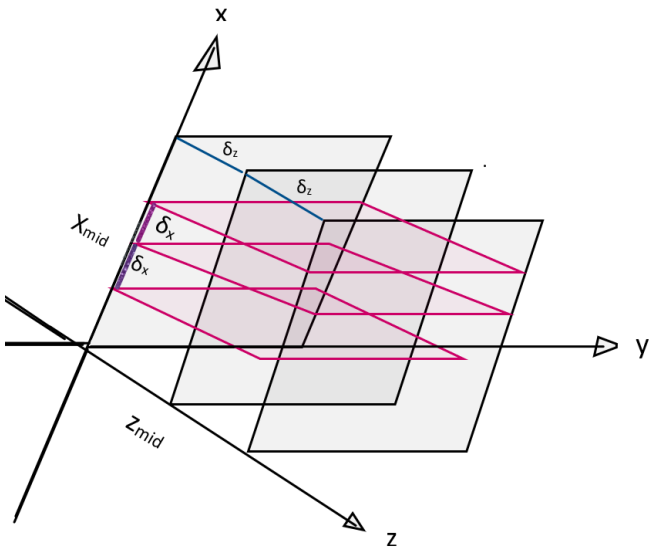
(i) Αρχικά, επιλέγουμε κάποια από τις μεταβλητές x, y, z, έστω εδώ την z, και ταξινομούμε όλα τα σημεία ως προς αυτήν, όπως ακριβώς και στις δύο διαστάσεις. Το βήμα αυτό κοστίζει $O(nlogn)$.

(ii) Βρίσκουμε το *median* των σημείων z_m, και τα χωρίζουμε σε μεγαλύτερα και μικρότερα του, φτιάχνουμε δηλαδή δύο υποπροβλήματα.

(iii) Αναδρομικά, βρίσκουμε τα ζευγάρια με την ελάχιστη απόσταση δεξιά και αριστερά της διαχωριστικής επιφάνειας του *median* ως εξής:

(iv) Για καθένα από τα δύο υποπροβλήματα, συνεχίζοτμε να τα χωρίζουμε στην μέση με τα δικά τους *median*, εώς ότου το πλήθος των σημείων σε τουλάχιστον ένα από τα σετ που έχουν προκύψει στο τελευταίο επίπεδο της αναδρομής να είναι **2**. Έστω τα σημεία p_{a_1}, p_{a_2} στο ένα σετ, με $d_1 = distance(p_{a_1}, p_{a_2})$. Σε αυτή την περίπτωση, ορίζουμε την ελάχιστη απόσταση $\delta_z = d_1$. Αν και τα δύο σετ περιέχουν δύο σημεία, τότε όρισε την ελάχιστη απόσταση $\delta_z = min\{d_1, d_2\}$.

(v) Για τα ζευγάρια με σημεία εκατέρωθεν της διαχωριστικής γραμμής, όρισε έναν χώρο ενός παραλληλεπιπέδου γύρω από την διαχωριστική επιφάνεια του επιπέδου αναδρομής στο οποίο βρισκόμαστε, του οποίου οι δύο παράλληλες στην διαχωριστική επιφάνεια πλευρές απέχουν κάθετη απόσταση δ_z από αυτήν, όπως φαίνεται και στο σχήμα.



(vi) Σε αυτό το βήμα, μπορούμε εύκολα να παρατηρήσουμε πως οι συγκρίσεις που θα χρειαστεί να γίνουν για τα σημεία μέσα στον χώρο που ορίζουν οι δύο επίπεδες πλάκες είναι σημαντικά λιγότερες.

(vii) Ταξινομούμε τα -πιθανώς πολύ λίγα- στοιχεία κατά άλλη μια μεταβλητή, έστω την x.

(viii) Πλέον, για την διάσταση αυτή μπορώ να χωρίσω τον χώρο μου σε κυβάκια διαστάσεων $\frac{1}{2}\delta_z$.

(ix) Θεωρώντας πως σε κάθε κύβο χωράει ακριβώς ένα σημείο, ξεκινάμε με εκείνο με το μικρότερο x. Το μακρινότερο σε αυτό σημείο που δεν ξεπερνάει την απόσταση δ_z δεν μπορεί να ανήκει πάνω από δύο κύβους προς κάθε κατεύθυνση, τοποθετημένους τον έναν δίπλα στον άλλο.

(x) Έτσι καταλήγουμε λοιπόν, πως για κάθε σημείο θα κάνουμε το πολύ 34 συγκρίσεις. Επομένως η πολυπλοκότητα του βήματος αυτού είναι O(n)

(xi) Στην συνέχεια, ξαναχωρίζουμε τα στοιχεία με το *median* της διάστασης x, x_m, και δημιουργούμε ένα παραλληλεπίπεδο χώρο με πλάτος δύο φορές την ελάχιστη απόσταση που προέκυψε από τις ακριβώς προηγούμενες συγκρίσεις μας, δ_x.

(xii) Ξανακόβουμε τον εναπομείναντα χώρο, σε κύβους διαστάσεων $\frac{1}{2}\delta_x$ αυτή τη φορά, και ταξινομώντας τα σημεία κατά την μόνη διάσταση που δεν έχουμε εξετάσει, εδώ y, επαναλαμβάνουμε το προηγούμενο βήμα των συγκρίσεων, που ξαναείναι 34 ανά σημείο το πολύ. Ξανά, η πολυπλοκότητα του βήματος αυτού είναι O(n)

(xiii) Είναι σημαντικό να σημειωθεί πως "κόβοντας" κάθε φορά τον χώρο στο μέσο μιας διάστασης, εξασφαλίζουμε πως η φλοίδα του χώρου που κρατάμε περιέχει τα κοντινότερα ως προς αυτή την διάσταση σημεία. Έτσι, μετά το δεύτερο κόψιμο, τα σημεία που απομένουν είναι κοντά τόσο κατά z αλλά και κατά x. Μένει λοιπόν να συγκριθούν με την ταξινόμηση ως προς την εναπομείνουσα διάσταση, και να τοποθετηθούν σε κυβάκια διαστάσεων $\frac{1}{2}\delta_x$, και να συγκριθούν με έναν συγκεκριμένο αριθμό γειτονικών τους πιθανών σημείων, για να έχουμε πλέον βρει την ελάχιστη απόσταση σημείων σε όλο το χώρο.

(xiv) Ακόμη, προκειμένου να μειώσουμε την πολυπλοκότητα του αλγορίθμου, η ταξινόμηση των σημειών σε κάθε διάσταση δεν χρειάζεται να γίνεται εκ νέου για κάθε παραλληλεπίπεδο χώρο πλάτους δ, αλλά με κάθε αναδρομή να επιστρέφονται όλα τα σημεία ταξινομημένα ήδη κατά τις άλλες δύο συντεταγμένες. Έτσι, αντί για ταξινόμηση, μπορούμε απλά να συγχωνεύουμε δύο ήδη ταξινομημένες λίστες.

Τελικά, ο αλγόριθμος που περιγράφηκε έχει αναδρομική σχέση

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n) \tag{1}$$

$$\Rightarrow\ T(n) = O(n \log n) \tag{2}$$

(b) Στην περίπτωση αυτή, γνωρίζουμε ήδη μια αρκετά καλή προσέγγιση της βέλτιστης απόστασης d^*, η οποία ανήκει στο διάστημα *[l, cl]*, με $c > 1$,

$$l > 0$$

.

(i) Με την γνώση αυτή, θεωρούμε δύο σημεία p_1, p_2.

(ii) Είτε πρόκειται για το ζευγάρι σημείων με την βέλτιστη απόσταση είτε όχι, γνωρίζουμε πως η απόστασή τους αποκλείεται να είναι μικρότερη από l.

(iii) Κατασκευάζουμε λοιπόν ένα πλέγμα ακτίνας l, στου οποίου τα κουτάκια μπορεί να ανήκει το πολύ ένα σημείο.

(iv) Θεωρόντας αρχικά ότι βρισκόμαστε στις 2 διαστάσεις, είναι φανερό από το σχήμα παρακάτω πως για κάποιο σημείο μέσα στο πλέγμα, οι μόνες συγκρίσεις που θα χρειαστεί να γίνουν είναι εκείνες για τα c γειτονικά κουτάκια προς κάθε κατεύθυνση. Επομένως στις δύο διαστάσεις το πλήθος των συγκρίσεων ανά σημείο είναι ανάλογο του $(2 \cdot c)^2$

(v) Γενικεύοντας λοιπόν στις d διαστάσεις, προκύπτουν $(2 \cdot c)^d$ συγκρίσεις ανά σημείο, το οποίο αποδεικνύει πως πρόκειται για γραμμικό χρόνο.

Άσκηση 2: Πόρτες Ασφαλείας στο Κάστρο

Για τον πρόβλημα θεωρούμε πως οι διακόπτες αποτελούν μια δυαδική συμβολοσειρά, με το 1 να συμβολίζει την πάνω και το 0 την κάτω θέση την οποία μπορούν να πάρουν. Για ίδιο πλήθος διακοπτών και πορτών αριθμού n, και χωρίς να γνωρίζουμε ποια θέση του κάθε διακόπτη ανοίγει ή κλείνει την πόρτα στην οποία αντιστοιχεί, προκείπτουν συνολικά 2^n συνδυασμοί, όπως φαίνεται παρακάτω:

s_1	s_2	...	s_i	...	s_{n-1}	s_n
0	0	...	0	...	0	0
0	0	...	0	...	0	1
⋮	⋮	...	⋮	...	⋮	⋮
1	1	...	1	...	1	0
1	1	...	1	...	1	1

Προφανώς, η πολυπλοκότητα του ελέγχου καθενός από τους πιθανούς συνδυασμούς θα ήταν υπερβολικά μεγάλη. Προκειμένου λοιπόν να γίνει η κατάλληλη αντιστοίχηση διακοπτών και πορτών, στις σωστές θέσεις, θα εφαρμόσουμε τον παρακάτω αλγόριθμο:

Algorithm 1 Switch-Port-Position Matching Algorithm

Require: n, $[s_1, s_2, ..., s_n]$, $[d_1, d_2, ..., d_n]$

1: **for** i from 1 to n **do**
2: Initialize all switches to 0 and check if door i is open.
3: Move the first half of the switches to 1 and leave the rest at 0.
4: **if** Door i closed after being open **then**
5: Indicates that the switch belongs to the first half and opens when set to 0.
6: **else if** Door i opened while it was closed **then**
7: Indicates that the switch belongs to the first half and opens when set to 1.
8: **else if** It remained closed and unchanged **then**
9: The switch belongs to the second half and opens when set to 1.
10: **else if** It remained open and unchanged **then**
11: The switch belongs to the second half and opens when set to 0.
12: **end if**
13: **end for**
14: Choose one of the halves that includes the switch of the first door and repeat the process until only one switch remains. It takes on the value that opens the door.

Όπως φαίνεται, ο αλγόριθμος αυτός εκτελεί Δυαδική Αναζήτηση για κάθε διακόπτη. Έτσι, εύκολα δείχνεται πως η συνολική του πολυπλοκότητα είναι

$$O(nlogn) \tag{3}$$

Άσκηση 3: Κρυμμένος Θησαυρός

Σε αυτό το πρόβλημα, καλούμαστε να αποφασίσουμε ποιος είναι ο βέλτιστος αλγόριθμος μετακίνησης πάνω στον άξονα των ακεραίων, προκειμένου να βρούμε πιο αποδοτικά τον θησαυρό στην θέση x.
Αρχικά, είναι προφανές πως ο θησαυρός θα βρίσκεται είτε στον θετικό, είτε στον αρνητικό ημιάξονα. Αυτό σημαίνει πως όλες οι μετακινήσεις που θα πραγματοποιήσουμε προς τον έναν από τους δύο θα είναι εξ' ολοκλήρου λανθασμένες.
Επιθυμούμε να μειώσουμε όσο μπορούμε αυτό το λάθος. Έτσι, θα επιλέξουμε ένα βήμα d, του οποίου η δύναμη στην οποία θα υψώνεται θα αυξάνεται κατά ένα μετά την επιστροφή στο 0.
Ισχύει πως κάθε αριθμός x βρίσκεται ανάμεσα σε έναν αριθμό d υψωμένο σε μια δύναμη k, και τον ίδιο αριθμό υψωμένο στο k+1, δηλαδή:

$$d^k \leq x \leq d^{k+1}$$

Φαίνεται λοιπόν, πως για βήμα d, ο αλγόριθμός μας θα εκτελέσει στην χειρότερη περίπτωση d^k βήματα από την σωστή, και επομένως d^{k+1} βήματα προς την λάθος μεριά, πριν γυρίσει στην σωστή για να βρει τον στόχο.

Τα βήματα του αλγορίθμου μπορούν να συμβολιστούν με την βοήθεια μιας σειράς, η οποία θα είναι κάπως έτσι:

$$\begin{aligned} 2 \cdot d^0 + 2 \cdot d^1 + \ldots + 2 \cdot d^k + 2 \cdot d^{k+1} + x &= \sum_{i=0}^{k+1} (2 \cdot d^i) + x \\ &= 2 \cdot (d^{k+1+1} - 1) + x \\ &= 2 \cdot (d^{k+2} - 1) + x \end{aligned}$$

Όπως φαίνεται, για κάθε δύναμη i στην οποία υψώνεται το βήμα d, πρέπει να προσθέσουμε το d^i δύο φορές, μία για να φύγει από το 0 και μία για να γυρίσει. Το τελευταίο βήμα του αλγορίθμου είναι, προφανώς, το x, καθώς ο αλγόριθμος τερματίζει με το που θα περάσει πάνω από τον στόχο.
Σε αυτό το σημείο, είναι φανερό πως πρόκειται για μια αύξουσα σειρά. Επομένως, όσο μικρότερο είναι το βήμα μου, τόσο μικρότερη θα είναι και η αύξηση των βημάτων μου. Γνωρίζω πως το βήμα πρέπει να είναι φυσικός αριθμός, και πως το 1 θα επιφέρει την ταλάντωση γύρω από το 0 από το 1 στο -1 για πάντα. Επομένως, θα επιλέξω για βήμα το 2. Η σειρά μου τώρα γίνεται:

$$\begin{aligned} 2 \cdot (d^{k+2} - 1) + x &= 8 \cdot (2^k) - 8 + x \\ &\leq 8 \cdot x - 8 + x = 9 \cdot x - 8 \end{aligned}$$

Άρα ο αλγόριθμος μας έχει πράγματι πολυπλοκότητα

$$O(n)$$

ενώ θα εκτελέσει βήματα

$$c \cdot |x|, \text{ με } c \leq 9$$

Άσκηση 4: Μη Επικαλυπτόμενα Διαστήματα Μέγιστου Συνολικού Μήκους

1. Προκειμένου να αποδείξουμε πως οι δύο αλγόριθμοι δεν βρίσκουν την βέλτιστη λύση, θα βρούμε από ένα αντιπαράδειγμα. Έστω η είσοδος $n = 5$ διαστημάτων, $[1,3), [2,6), [4,7), [5,8), [7,8)$. Γνωρίζουμε πως η βέλτιστη λύση του συγκεκριμένου προβλήματος είναι τα διαστήματα $([1,3), [4,7), [7,8))$.

   (a) Έστω ο αλγόριθμος που επιλέγει κάθε φορά το διαθέσιμο διάστημα $[s_i, f_i)$ με το μεγαλύτερο μήκος $f_i - s_i$. Στην περίπτωση μας, η επιλογή θα γινόταν διαδοχικά $[2,6), [7,8)$, καθώς προκειμένου να επιλεγεί ένα διάστημα πρέπει η αρχή του να είναι μετά το τέλος του προηγούμενου επιλεγμένου. Έτσι, ενώ ο αλγόριθμος βρήκε μια λύση με διαστήματα που δεν επικαλύπτονται μεταξύ τους, δεν έχει βρει την βέλτιστη.

   (b) Έστω τώρα ο αλγόριθμος που επιλέγει κάθε φορά το διαθέσιμο διάστημα $[s_i, f_i)$ με τον ελάχιστο χρόνο ολοκλήρωσης f_i. Σε αυτή την περίπτωση, οι επιλογές του αλγορίθμου θα ήταν διαδοχικά $([1,3), [4,7), [7,8))$, η οποία τυχαίνει να είναι και η σωστή λύση. Αν ωστόσο κάνουμε μια μικρή παραλλαγή στα αρχικά μας δεδομένα, προσθέτοντας το διάστημα $[4,6)$, η οποία ωστόσο δεν επηρεάζει την βέλτιστη λύση, τότε η επιλογή του αλγορίθμου θα ήταν η $([1,3), [4,6), [7,8))$. Ο αλγόριθμός μας λοιπόν, έχει κάνει λάθος.

2. Προκειμένου να διατυπώσουμε αποδοτικό αλγόριθμο για το πρόβλημα της επιλογής μη επικαλυπτόμενων διαστημάτων, θα χρειαστεί να κάνουμε κάποιες παραδοχές:
   Αρχικά, παρατηρούμε πως η χρήση αναδρομής για την επίλυση του προβλήματος θα επιφέρει ένα κόστος 2^n πράξεων, και επομένως δεν θα την προτιμήσουμε. Στην συνέχεια εξετάζουμε την πιθανότητα μιας λύσης με γραμμική αναζήτηση, η οποία ωστόσο και πάλι επιφέρει έναν αριθμό n^2 πράξεων. Τελικά, καταλήγουμε στον δυναμικό προγραμματισμό. Παρακάτω περιγράφεται και στην συνέχεια παρατίθεται ο αλγόριθμος.

- Θα βρούμε την τιμή της βέλτιστης λύσης.
- Αρχικά, ταξινομούμε τα διαστήματα κατά αύξοντα χρόνο ολοκλήρωσης $f_1 \leq f_2 \leq .... \leq f_n$. Το βήμα αυτό έχει πολυπλοκότητα $O(n \cdot \log n)$
- Στην συνέχεια, για κάθε διάστημα i βρίσκουμε το κοντινότερο προηγούμενό του j, για το οποίο ισχύει $f_j \leq s_i$, και αποθηκεύουμε αυτή την πληροφορία σε έναν μονοδιάστατο πίνακα $prev[n]$. Το βήμα αυτό έχει πολυπλοκότητα $O(n \cdot \log n)$, καθώς ο πίνακας των χρόνων ολοκλήρωσης f_i είναι ήδη ταξινομημένος. Επομένως, το μόνο που απομένει για να βρούμε τα κατάλληλα προηγούμενα διαστήματα, είναι να εκτελέσουμε δυαδική αναζήτηση στον πίνακα αυτό n φορές.
- Υπολογίζουμε έπειτα την διάρκεια της βέλτιστης λύσης με την ύπαρξη κάποιων από όλα τα διαστήματα, προσθέτοντας κάθε φορά ένα με την σειρά ταξινόμησής τους, και αποθηκεύοντας την τιμή τους σε έναν μονοδιάστατο πίνακα $opt[n]$. Το $opt[i] = k$ για παράδειγμα, σημαίνει πως, αν έχω τα διαστήματα 1 ως j, η καλύτερη δυνατή λύση έχει μέγεθος k.
  Ο υπολογισμός αυτός γίνεται αξιοποιώντας την πληροφορία του πίνακα $prev[n]$ με τον εξής τρόπο:
  (a) Αρχικοποιούμε όλα τα κελιά του πίνακα $opt[n]$ στο 0
  (b) Για όλα τα διαστήματα i από το 1 μέχρι το n, θέτουμε το αντίστοιχο κελί τους στον πίνακα $opt[n]$ ίσο με το μέγιστο μεταξύ του αθροίσματος του μήκους τους $f_i - s_i$ και της βέλτιστης λύσης για τα διαστήματα από την αρχή μέχρι το κοντινότερο κατάλληλο προηγούμενό τους από τον πίνακα $prev[n]$, και την καλύτερη λύση για τα διαστήματα μέχρι το αμέσως προηγούμενο στην ταξινόμηση. Το μέγιστο αυτό θα είναι η βέλτιστη λύση για τα διαστήματα μέχρι και το i.
  (c) Με αυτόν τον τρόπο προκύπτει πως η καλύτερη λύση για το πρόβλημα θα είναι το τελευταίο κελί του πίνακα $opt[n]$.
- Έχοντας λοιπόν την τιμή της βέλτιστης λύσης, πρέπει να βρούμε ποια είναι τα διαστήματα που την συνθέτουν.
  (a) Ξεκινώντας από το τελευταίο διάστημα της ταξινόμησης, υπολογίζουμε κάθε φορά αν το άθροισμα του μήκους του $f_i - s_i$ και της βέλτιστης λύσης των διαστημάτων από την αρχή μέχρι και το κοντινότερο κατάλληλο προηγούμενό του είναι μεγαλύτερο από την βέλτιστη λύση των διαστημάτων μέχρι και πριν από αυτό. Με άλλα λόγια, ελέγχουμε αν η προσθήκη του διαστήματος αύξησε την τιμή της βέλτιστης λύσης, και άρα αν συμμετείχε σε αυτήν, συμβουλευόμενοι κάθε φορά τον πίνακα $prev[n]$.
  (b) Αν η παραπάνω συνθήκη ισχύει, τότε τυπώνουμε το διάστημα i και καλούμε την συνάρτηση αναδρομικά για το $prev[i]$.
  (c) Όταν το $i = 0$, η συνάρτηση τερματίζει.
- Η συνάρτηση αυτή κάνει λιγότερες από n αναδρομές, καθώς δεν καλείται καν για όλα τα διαστήματα (εκτός αν κανένα δεν επικαλύπτεται με κάποιο). Επομένως η πολυπλοκότητα της είναι $O(n)$

Algorithm 2 Job Scheduling with Maximum Value

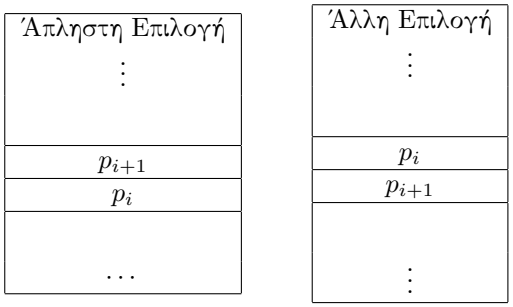
```
1: procedure ITERATIVE-COMPUTE-OPT(n, s, f, v)
2:     Sort jobs by finish times such that f_1 ≤ f_2 ≤ ... ≤ f_n
3:     for j = 1 to n do
4:         OPT[0] = 0
5:         OPT[j] = max(v_j + OPT[p(j)], OPT[j − 1])
6:     end for
7:     return OPT[n]
8: end procedure
9: procedure FIND-SOLUTION(n, j)
10:    if j = 0 then
11:        return (output nothing)
12:    else if v_j + OPT[p(j)] > OPT[j − 1] then
13:        print j
14:        FIND-SOLUTION(n, p(j))
15:    else
16:        FIND-SOLUTION(n, j − 1)
17:    end if
18: end procedure
```

Άσκηση 5: Παραλαβή Πακέτων

1. Για το πρόβλημα με μόνο έναν υπάλληλο διαθέσιμο, υποθέτουμε κατευθείαν πως η λύση βρίσκεται σε κάποιον άπληστο αλγόριθμο. Προκειμένου ωστόσο κάτι τέτοιο να ισχύει, θα πρέπει να βρούμε την κατάλληλη ταξινόμηση, προκειμένου να μπορεί ο άπληστος αλγόριθμος να ελαχιστοποιήσει την συνάρτηση βεβαρυμένου χρόνου εξυπηρέτησης.
   Με μια γρήγορη ματιά παρατηρούμε πως, αφού η συνάρτηση εξαρτάται τόσο από τα βάρη, αλλά και τους χρόνους εξυπηρέτησης των πελατών, η ορθότερη ταξινόμηση θα μοιάζει με εκείνη του $Knapsack$ με ποσοστά. Έτσι, ταξινομούμε τους πελάτες σε φθίνουσα σειρά πηλίκου σημασίας του πελάτη i προς τον χρόνο εξυπηρέτησης του. Ισχύει δηλαδή πως:

$$\frac{w_1}{p_1} \geq \frac{w_1}{p_2} \geq \ldots \geq \frac{w_n}{p_n}$$

   Προκειμένου να αποδείξουμε τον ισχυρισμό μας, θα χρησιμοποιήσουμε το Θεώρημα Ανταλλαγής. Έστω λοιπόν η επιλογή σύμφωνα με την Άπληστη στον πρώτο πίνακα, και κάποιο άλλο κριτήριο επιλογής στον δεύτερο. Σκοπός μας είναι να αποδείξουμε πως η άπληστη επιλογή θα παρέχει μια καλύτερη λύση από οποιαδήποτε άλλη.
   Όπως είναι λογικό, αφού η άλλη επιλογή δεν χρησιμοποιεί το συγκεκριμένο άπληστο κριτήριο, θα υπάρχουν δύο διαδοχικά i, i+1 τέτοια ώστε τα $\frac{w_i}{p_i} \geq \frac{w_{i+1}}{p_{i+1}}$ και παρόλαυτα θα είναι ταξινομημένα όπως φαίνεται παρακάτω:



   Για τις δύο παραπάνω ταξινομήσεις, και για να στηρίξουμε τον ισχυρισμό μας θα θεωρήσουμε πως η

συνάρτηση βεβαρυμένου χρόνου είναι μικρότερη για τον άπληστο. Έτσι, γράφουμε:

$$\begin{aligned}
&\cdots + w_i \cdot (p_1 + p_2 + \cdots + p_i) + w_{i+1} \cdot (p_1 + p_2 + \cdots + p_i + p_{i+1}) \\
&\leq \cdots + w_{i+1} \cdot (p_1 + p_2 + \cdots + p_{i-1} + p_{i+1}) + w_i \cdot (p_1 + p_2 + \cdots + p_{i+1} + p_i) \\
&\Rightarrow \ w_{i+1} \cdot p_i \leq w_i \cdot p_{i+1} \\
&\Rightarrow \ \frac{w_{i+1}}{p_{i+1}} \leq \frac{w_i}{p_i}
\end{aligned}$$

Το οποίο αποτελεί την αρχική μας ταξινόμηση. Επομένως, όντως η άπληστη επιλογή δίνει την καλύτερη λύση, και άρα αποδεικνύεται η ορθότητα του αλγορίθμου. Η πολυπλοκότητα του αλγορίθμου λοιπόν, προκύπτει από την πολυπλοκότητα της ταξινόμησης, η οποία είναι

$$O(n \cdot \log n)$$

2. Για το δεύτερο ερώτημα, η βασική απόφαση που καλείται να πάρει ο αλγόριθμός μας είναι σε ποιον από τους δύο εξυπηρετητές θα αναθέσει την κάθε πελάτη. Επειδή πλέον δεν μπορούμε να χρησιμοποιήσουμε την άπληστη επιλογή, θα καταφύγουμε στον Δυναμικό Προγραμματισμό.
Θεωρούμε πως οι πελάτες μας είναι ταξινομημένοι όπως βρέθηκε βέλτιστο στο προηγούμενο ερώτημα, δηλαδή πως

$$\frac{p_1}{w_1} \leq \frac{p_2}{w_2} \leq \ldots \leq \frac{p_n}{w_n}$$

Αφού για κάθε επόμενο πελάτη της λίστας μας, ο αλγόριθμος θα πρέπει να επιλέγει τον κατάλληλο εξυπηρετητή για να τον αναλάβει, το μόνο που χρειάζεται να γνωρίζουμε σε κάθε στάδιο πέρα του πελάτη, είναι ο χρόνος εξυπηρέτησης που έχει καλύψει καθένας εξυπηρετητής μέχρι τότε.
Ενδεικτικά, θεωρούμε ως T_A τον χρόνο που έχει καλύψει ο ταξινομητής Α μέχρι και την σειρά του i-οστού πελάτη. Όπως είναι λογικό, ο χρόνος του ταξινομητή Β θα είναι

$$T_B = \sum_{j=0}^{i} p_i - T_A$$

. Από τα παραπάνω δεδομένα, προκύπτει η παρακάτω αναδρομική σχέση για την συνάρτηση βέλτιστου βεβαρυμένου χρόνου, την οποία και προσπαθούμε να ελαχιστοποιήσουμε:

$$C^*(i, T_A) = \min\{C^*(i-1, T_A) + (\sum_{j=0}^{i}(p_j) - T_A) \cdot w_i, C^*(i-1, T_A - p_i) + p_i \cdot T_A\}$$

Όπως συμβολίζεται και στην αναδρομική σχέση παραπάνω, η επιλογή του αλγορίθμου θα είναι ο εξυπηρετητής που θα επιφέρει το μικρότερο χρονικό κόστος. Συμβολίζουμε την επιλογή του Β εξυπηρετητή πρώτη, με την απλή παραδοχή πως, αν για τον πελάτη i δεν επιλεχθεί ο πρώτος, τότε ο χρόνος του πρώτου έχει παραμείνει ίδιος από τουλάχιστον τον προηγούμενο πελάτη, ενώ το έξτρα άθροισμα που πρέπει να ληφθεί υπόψην είναι ο χρόνος του Β εξυπηρετητή T_B, επί το βάρος του πελάτη i. Στην επιλογή, αντίστοιχα, του εξυπηρετητή Α, συμβολίζεται θεωρόντας ότι ο χρόνος T_A έγινε με τον τελευταίο πελάτη, και άρα στην προηγούμενη επιλογή πελάτη ο Α είχε χρόνο $T_A - p_i$. Η νέα επιβάρυνση θα είναι λοιπόν $p_i \cdot w_i$

Ο αλγόριθμος αυτός καλείται να ελέγξει, για κάθε νέο πελάτη i, την βέλτιστη λύση, για κάθε χρόνο $0 \leq T_A \leq \sum_{j=0}^{i}(p_j)$. Επομένως η λύση του προβλήματος θα είναι τελικά:

$$\min\{C^*(n, T_A)\}$$

Για να βρεθούν όλες οι λύσεις, και να συγκριθούν, απαιτείται ένας μεγάλος αριθμός υπολογισμών για κάθε n. Η τελική πολυπλοκότητα του αλγορίθμου βρίσκεται

$$O(n \cdot T_A) = O(n^2),$$

καθώς ο χρόνος του πρώτου επεξεργαστή εξαρτάται από το n.
Όσο για την γενίκευση σε περισσότερους υπαλλήλους, έστω k, το πρόβλημά μας θα απαιτούσε πλέον την ύπαρξη k-1 μεταβλητών για τους εξυπηρετητές, μίας για καθέναν εκτός του τελευταίου, που προκύπτει από αφαίρεση των προηγούμενων από το σύνολο. Επομένως, η αναδρομική σχέση πλέον προκύπτει:

$$C^*(i, T_1, ..., T_{m-1}) = \min\{(C^*(i-1, T_1 - p_1, ..., T_{m-1}) + w_i \cdot T_1), ..., (C^*(i-1, T_1, ..., T_{m-1} - p_i) + w_i \cdot T_{m-1})),$$
$$C^*(i-1, T_1, ..., T_{m-1}) + w_i \cdot (\sum_{j=0}^{i}(p_j) - T_1 - T_2 - .... - T_{m-1})\}$$

Με πολυπλοκότητα:

$$O(n^m),$$