Εθνικο Μετσοβιο Πολυτεχνειο

2η Γραπτή Άσκηση

Αλγόριθμοι & Πολυπλοκότητα

Σπουδαστής:
Διονύσης Ζηνδρος (06601)
<dionyziz@gmail.com>

Διδάσκοντες:
Στάθης Ζαχος
Δημήτρης Φωτακης

31 Δεκεμβρίου 2011

Άσκηση 1

Ο αλγόριθμος που χρησιμοποιούμε είναι άπληστος. Ταξινομούμε τα διαστήματα σε αύξουσα σειρά σύμφωνα με το χρόνο τερματισμού f_i. Κατά τη διάρκεια του αλγορίθμου, χρωματίζουμε όσα διαστήματα έχουμε, με κάποιον τρόπο, επικαλύψει. Θεωρούμε ότι αρχικά κανένα διάστημα δεν είναι χρωματισμενο. Επιλέγουμε κάθε φορά να καλύψουμε το διάστημα που βρίσκεται αριστερότερα, σύμφωνα με την ταξινόμησή μας, από όλα τα διαστήματα που δεν έχουμε χρωματίσει ακόμη, αρχικά δηλαδή το αριστερότερο από όλα. Στη συνέχεια, επιλέγουμε από όλα τα διαστήματα που το επικαλύπτουν εκείνο που βρίσκεται δεξιότερα έτσι ώστε να πετύχουμε το μέγιστο δυνατό εύρος επικάλυψης. Αυτό το διάστημα επιλέγεται αμετάκλητα ως μέρος της απάντησής μας.

Αφότου κάνουμε την επιλογή αυτού διαστηματος, χρωματίζουμε όλα τα διαστήματα που αυτό επικαλύπτει, και συνεχίζουμε την ίδια διαδικασία.

Algorithm 1 Άσκηση 1

```
1:  procedure COVERS(a, b)
2:      return a.s ≤ b.s ∧ a.f > b.f ∨ b.s ≤ a.s ∧ b.f > a.f
3:  procedure INTERVALS(s, f, n)
4:      for i ← 1 to n do
5:          A[i].s ← s_i
6:          A[i].f ← f_i
7:          A[i].covered ← ⊥
8:      A ← sort A by f
9:      cost ← 0
10:     for i ← 1 to n do
11:         if ¬A[i].covered then
12:             for j ← n downto 1 do
13:                 if COVERS(A[i], A[j]) then
14:                     cost ← cost + 1
15:                     for k ← 1 to n do
16:                         if COVERS(A[j], A[k]) then
17:                             A[k].covered ← ⊤
18:                     break
19:     return cost
```

Ο αλγόριθμος θα είναι $O(n^2)$ καθώς περνάμε από n διαστήματα για κάθε ένα από τα οποία βρίσκουμε το τελευταίο με το οποίο επικαλύπτεται σε $O(n)$ και στη συνέχεια, σε $O(n)$, χρωματίζουμε τα το πολύ n διαστήματα που η επιλογή μας επικαλύπτει.

Άσκηση 2

Διασχίζουμε όλα τα διαστήματα με τη σειρά από εκείνο με το μικρότερο όριο ταχύτητας έως εκείνο με το μεγαλύτερο όριο ταχύτητας προσθέτωντας κάθε φορά όλο το χρόνο που μας απομένει στο αντίστοιχο διάστημα.

Λήμμα. *Η λύση που επιλέγει τα διαστήματα σε αύξουσα σειρά με βάση το όριο ταχύτητας είναι ορθή.*

Απόδειξη.

Επιχείρημα διαχωρισμού: Το πρόβλημα παραμένει ισοδύναμο αν "σπάσουμε" οποιοδήποτε διάστημα του δρόμου σε μικρότερα.

Αν επιλέξουμε τα διαστήματα σε αύξουσα σειρά με βάση το όριο ταχύτητας και κατανείμουμε όσο χρόνο έχουμε διαθέσιμο όσο μπορούμε, ενδέχεται να υπάρχει ένα τελευταίο διάστημα στο οποίο θα πρέπει να κατανείμουμε το χρόνο που μας απομένει σε μέρος του. Αν υπάρχει, χρησιμοποιώντας το επιχείρημα διαχωρισμού, επιλέγουμε να "σπάσουμε" το διάστημα αυτό σε δύο τμήματα έτσι ώστε ο χρόνος που απομένει να μπορεί να κατανεμηθεί ολόκληρος στο πρώτο μισό, έτσι ώστε να μην υπάρχει άλλος χρόνος για το δεύτερο μισό. Αυτό οδηγεί σε ένα ισοδύναμο πρόβλημα και διευκολύνει την απόδειξη.

Έστω ότι η ορθή λύση, αν είχαμε μόνο τα πρώτα i διαστήματα με το μικρότερο όριο ταχύτητας στη διάθεσή μας, έχει κόστος $J'[i]$, δηλαδή συνολικό χρόνο που χρειάζεται ο μοτοσυκλετιστής για να τα διασχίσει όλα. Έστω ότι η λύση μας που περιλαμβάνει όσα διαστήματα μπορεί με τη σειρά από το μικρότερο προς το μεγαλύτερο έχει κόστος $J[i]$. Θα δείξουμε ότι τα κόστη αυτά ταυτίζονται.

Έστω $K[i]$ ο δείκτης του i-οστού διαστήματος σε αύξουσα σειρά ορίου ταχύτητος.

Θα είναι:

$$J[0] = J'[0]$$

Έστω, τώρα ότι:

$$\forall i < n : J[i] = J'[i]$$

Υποθέτουμε τώρα ότι:

$$\sum_{j=0}^{i} \frac{l_{K[j]}}{v + u_{K[j]}} \leq T$$

Δηλαδή ότι απομένει ακόμη χρόνος υπέρβασης ορίου ταχύτητας. Τότε επιλέγουμε να συμπεριλάβουμε ολόκληρο το i-οστό τμήμα. Αυτό μπορεί να συμπεριληφθεί ολόκληρο λόγω του επιχειρήματος διαχωρισμού διαστημάτων. Το να συμπεριλάβουμε το συγκεκριμένο τμήμα οδηγεί σε βέλτιστη λύση, διότι αν

δεν το είχαμε συμπεριλάβει, θα είχαμε κάποιο τμήμα χρόνου υπέρβασης ορίου ταχύτητας ανεκμετάλλευτο το οποίο θα μπορούσαμε να είχαμε εκμεταλλευτεί. Αντίστοιχα, η λύση για τα πρώτα i τμήματα θα πρέπει να περιλαμβάνει τη λύση για τα $i-1$ τμήματα, καθώς οποιαδήποτε άλλη λύση για $i-1$ τμήματα αφήνει ένα τμήμα χρόνου ανεκμετάλλευτο.

Συνεπώς:

$$J[i] = J'[i]$$

Έστω τώρα ότι, αντίθετα, είναι:

$$\sum_{j=0}^{i} \frac{l_{K[j]}}{v + u_{K[j]}} > T$$

Τότε για την J λύση θα έχουμε ήδη αξιοποιήσει όλο το διαθέσιμο χρόνο υπέρβασης. Εξετάζουμε τώρα αν το i-οστό τμήμα θα συμπεριληφθεί. Η λύση J δεν θα συμπεριλάβει το i-οστό τμήμα. Θα δείξουμε ότι ούτε η J' θα το συμπεριλαμβάνει. Έστω, τώρα, ότι η λύση $J'[i]$ επιλέγει να υπερβεί το όριο ταχύτητας σε ένα διαφορετικό σύνολο από τα $i-1$ από την $J[i]$ έτσι ώστε ο χρόνος διάσχισης να είναι $L > J'[i-1]$ και να απομένει κάποιος χρόνος κατανομής για το i-οστό τμήμα. Τότε ο χρόνος αυτός θα πρέπει να έχει αφαιρεθεί από ένα τμήμα από τα πρώτα $i-1$ τμήματα. Λόγω του επιχειρήματος διαχωρισμού διαστημάτων μπορούμε να θεωρήσουμε ότι η ανταλλαγή αφορά ένα ολόκληρο διάστημα από τα $i-1$ διαστήματα και χρησιμοποιείται για να αντικαταστήσουμε ολόκληρο το i-οστό διάστημα. Έστω ότι το i-οστό διάστημα επιλέγεται για υπέρβαση ταχύτητας αντί του j-οστού διαστήματος με $j < i$. Τότε θα είναι λόγω του επιχειρήματος διαχωρισμού:

$$\frac{l_{K[j]}}{v + u_{K[j]}} = \frac{l_{K[i]}}{v + u_{K[i]}}$$

Και έτσι λόγω της διάταξης έχουμε:

$$\begin{aligned} u_{K[i]} &> u_{K[j]} \\ \Rightarrow \frac{l_{K[i]}}{u_{K[i]}} &< \frac{l_{K[j]}}{u_{K[j]}} \\ \Rightarrow \frac{l_{K[i]}}{u_{K[i]}} - \frac{l_{K[i]}}{v + u_{K[i]}} &< \frac{l_{K[j]}}{u_{K[j]}} - \frac{l_{K[j]}}{v + u_{K[j]}} \end{aligned}$$

Από αυτό καταλήγουμε στο ότι η επιλογή του i-οστού διαστήματος είναι πιο επιθυμητή από την επιλογή του j-οστού διαστήματος και έτσι η λύση $J'[i-1]$ αποτελεί μέρος της λύσης $J'[i]$. Έτσι έχουμε αρχή της βελτιστότητας στο πρόβλημα. Συνεπώς προκύπτει ότι:

$$J[i] = J'[i]$$

Χρησιμοποιώντας την μαθηματική επαγωγή, αυτό ολοκληρώνει την απόδειξη. □

Algorithm 2 Άσκηση 2

```
1: procedure MOTORCYCLE(l, u, n, v, T)
2:     for i ← 1 to n do
3:         A[i].u ← u[i]
4:         A[i].l ← l[i]
5:         A[i].i ← i
6:     A ← sort A by u
7:     S ← ∅
8:     for i ← 1 to n do
9:         speed ← v + A[i].u
10:        t ← A[i].l / speed
11:        if T > t then
12:            S ← S ∪ {(A[i].i, A[i].l)}
13:            T ← T − t
14:        else
15:            S ← S ∪ {(A[i].i, T ∗ speed)}
16:            break
17:    return S
```

Ο αλγόριθμος θα είναι $\Theta(n \log n)$ καθώς περιλαμβάνει μία ταξινόμηση, ενώ η επόμενη επανάληψη είναι γραμμική.

Ας εξετάσουμε τώρα την περίπτωση που η παραβίαση του ορίου ταχύτητας γίνεται κατά έναν παράγοντα α.

Λήμμα. *Αν ο μοτοσυκλετιστής έχει τη δυνατότητα να παραβιάσει τα όρια ταχύτητας κατά έναν παραγοντα α, τότε η επιλογή των διαστημάτων είναι αδιάφορη.*

Απόδειξη. Έστω ότι ο μοτοσυκλετιστής θέλει να επιλέξει αν θα αφιερώσει χρόνο t για να παραβιάσει το όριο ταχύτητας στο τμήμα i ή στο τμήμα j. Τότε ο χρόνος που θα γλυτώσει θα είναι ανεξάρτητος αυτής της επιλογής.

Πράγματι, έστω ότι επιλέγει να αφιερώσει χρόνο t στο διάστημα i και έστω, χωρίς βλάβη της γενικότητας, ότι $t \leq l_i/u_i$. Τότε έστω t' ο χρόνος κατά τον οποίο ο μοτοσυκλετιστής διασχίζει ένα τμήμα του διαστήματος στην κανονική του ταχύτητα. Θα έχουμε:

$$
\begin{aligned}
l &= aut + ut' \\
t' &= \frac{l - aut}{u} \\
&= \frac{l}{u} - at \\
t + t' &= t + \frac{l}{u} - at \\
\frac{l}{u} - (t + t') &= at - t
\end{aligned}
$$

Αυτός είναι και ο χρόνος που θα έχει κερδίσει. Πράγματι, ο χρόνος αυτός είναι ανεξάρτητος της επιλογής i. □

Άσκηση 3

α)

Ένα παράδειγμα για το οποίο ο άπληστος αλγόριθμος δεν δίνει το βέλτιστο αποτέλεσμα είναι το εξής:

0	4999	0
4999	5000	4999
0	4999	0
0	0	0

Ενώ ο άπληστος αλγόριθμος θα επιλέξει να τοποθετήσει ένα βότσαλο στο κουτί με αξία 5000, η βέλτιστη λύση επιτυγχάνεται επιλέγοντας τα 4 κουτάκια που είναι γείτονές του και δίνουν συνολική αξία 19,996.

Οι χειρότερες περιπτώσεις για τον άπληστο αλγόριθμο είναι εκείνες που τον έξαναγκάζουν' να κάνει μία τοπικά βέλτιση επιλογή, όμως του στερούν μεγαλύτερο κέρδος συνολικά, όπως η παραπάνω. Τέτοιες περιπτώσεις περιλαμβάνουν ένα κεντρικό στοιχείο το οποίο έχει μία μεγάλη τιμή M και γύρω του, κάθετα και οριζόντια, στοιχεία που έχουν επίσης μεγάλες τιμές, αλλά κατά κάποιο ε μικρότερες από M. Έτσι ο αλγόριθμος επιλέγει το M αλλά αποκλείει την επιλογή 4 στοιχείων με συνολική αξία $4(M - \epsilon)$.

0	$M-\epsilon$	0
$M-\epsilon$	M	$M-\epsilon$
0	$M-\epsilon$	0
0	0	0

Έτσι, ο αλγόριθμος συγκεντρώνει πάντα περισσότερο από 20% της συνολικής αξίας, δηλαδή $\frac{M}{4(M-\epsilon)}$. Αθροιστικά, αν υπάρχουν και άλλες τέτοιες εμφανίσεις του μοτίβου μέσα στα δεδομένα, ο αλγόριθμος και πάλι θα συγκεντρώσει το 20%. Διαφορετικά, ο αλγόριθμος ενδέχεται να συγκεντρώσει και μεγαλύτερα ποσά.

β)

Η λύση δυναμικού προγραμματισμού εντοπίζει τη βέλτιστη διάταξη σε γραμμικό χρόνο διασχίζοντας τη σκακιέρα από τα αριστερά προς τα δεξιά.

Σε κάθε βήμα, όταν επεξεργαζόμαστε την i-οστή στήλη, αποθηκεύουμε το βέλτιστο ποσό $J[i,\mu]$ που μπορούμε να κερδίσουμε από το μέρος της σκακιέρας που αποτελείται από τις στήλες 1 έως i για κάθε πιθανό συνδυασμό βοτσάλων μ που μπορούμε να τοποθετήσουμε στην i-οστή στήλη. Έχοντας αυτά τα δεδομένα, ο υπολογισμός της βέλτιστης λύσης για την $(i+1)$-οστή στήλη προκύπτει από την εξής αναδρομική σχέση:

$$\Omega = \{x \in 2^{\{1,2,3,4\}} : \forall \alpha, \beta \in x : \alpha - 1 \neq \beta\}$$
$$J[i,\mu] = \max\{J[i-1,\mu'] + \sum_{q \in \mu} C[i,q] : \mu' \in \Omega \wedge \mu' \cap \mu = \emptyset\}$$

Algorithm 3 Άσκηση 3β

```
1:  procedure Chess(C, n)
2:      M ← {x ∈ 2^{1,2,3,4} : ∀α, β ∈ x : α − 1 ≠ β}
3:      for all μ ∈ M do
4:          J'[μ] ← 0
5:      for i ← 0 to n do
6:          for μ ∈ M do
7:              J[μ] ← 0
8:              for ν ∈ M do
9:                  if ν ∩ μ = ∅ then
10:                     J[μ] ← max(J[μ], J'[ν] + Σ_{q∈μ} C[i, q])
11:         J' ← J
12:     return max({J[μ] : μ ∈ M})
```

Άσκηση 4

Σε κάθε βήμα i του αλγορίθμου δυναμικού προγραμματισμού υπολογίζουμε το συνολικό κόστος $J[i]$ της βέλτιστης κατανομής λέξεων σε γραμμές μέχρι και την i-οστή λέξη. Στο τέλος του αλγορίθμου, το συνολικό κόστος δίνεται από το $J[n]$.

Η αρχή της βελτιστότητας που εμφανίζει το πρόβλημα είναι το γεγονός ότι όταν μία λύση είναι βέλτιστη, η κατανομή σε γραμμές όλων των λέξεων εκτός της τελευταίας γραμμής είναι βέλτιστη.

Λήμμα. *Το βέλτιστο κόστος κατανομής όλων των λέξεων μέχρι και την i-οστή δίνεται από την αναδρομική σχέση:*

$$J[0] = 0$$

$$J[i] = \min_{1 \le j \le i} \{J[j-1] + (C + 1 - \sum_{k=j}^{i} (l[j] + 1))^2\}$$

Απόδειξη. Έστω J' το πραγματικό βέλτιστο κόστος κατανομής όλων των λέξεων μέχρι και την i-οστή λέξη. Θα δείξουμε ότι $J = J'$.

Για $i = 0$ το κείμενο δεν έχει γραμμές και συνεπώς $J[0] = J'[0]$.

Έστω $\forall n < i : J[n] = J'[n]$. Έστω ότι η βέλτιστη λύση για τις i πρώτες λέξεις τοποθετεί τις λέξεις j έως i στην τελευταία γραμμή. Τότε το κόστος της τελευταίας γραμμής θα είναι:

$$(C + 1 - \sum_{k=j}^{i} (l[j] + 1))^2$$

Έστω τώρα ότι η βέλτιστη λύση των πρώτων i λέξεων περιλαμβάνει μία υποβέλτιστη λύση για την κατανομή των λέξεων 1 έως $j - 1$ με κόστος $K > J[j-1]$. Τότε το συνολικό κόστος της κατανομής έως και την i-οστή λέξη θα είναι:

$$J'[i] = K + (C + 1 - \sum_{k=j}^{i} (l[j] + 1))^2$$

Όμως θα είναι $J'[i] > J[i]$, το οποίο είναι αντίφαση καθώς το κόστος $J'[i]$ είναι ελαχιστο. Συνεπώς $K = J[j-1]$. Καθώς το J κρατά το ελάχιστο από όλα τα j, βελτιστοποιεί όσο αφορά το πλήθος των λέξεων που μπορεί να κρατήσει στην τελευταία γραμμή και άρα $J[i] = J'[i]$.

Χρησιμοποιώντας τη μαθηματική επαγωγή, αυτό ολοκληρώνει την απόδειξη. □

Έτσι επιτυγχάνουμε έναν πολυωνυμικό αλγόριθμο που για σταθερό C τρέχει σε $\Theta(N)$ με μία σταθερά που είναι ανάλογη του μέσου πλήθος λέξεων ανά γραμμή. Ο λόγος που ο αλγόριθμος είναι γραμμικός είναι το γεγονός ότι σταματά αφότου η γραμμή που εξετάζει σε κάθε επανάληψη ξεπεράσει το μέγιστο επιτρεπόμενο όριο γραμμής.

Algorithm 4 Άσκηση 4

```
1:  procedure LineSplit(l, n, C)
2:      J[0] ← 0
3:      for i ← 1 to n do
4:          J[i] ← ∞
5:          cost ← C + 1
6:          for j ← i downto 1 do
7:              cost ← cost − (l[j] + 1)
8:              if cost < 0 then
9:                  break
10:             J[i] ← min(J[i], J[j − 1] + cost^2)
11:     return J[n]
```

Ο αλγόριθμος εύκολα προσαρμόζεται έτσι ώστε για κάθε i να αποθηκεύει και το δείκτη της λέξης που βρίσκεται τελευταία στην αμέσως προηγούμενη γραμμή όταν βελτιστοποιούμε για τις i πρώτες λέξεις. Διασχίζοντας αυτό τον πίνακα, μπορούμε στη συνέχεια να οικοδομήσουμε την ομοιόμορφη κατανομή του όλου κειμένου σε γραμμές.

Άσκηση 5

Χρησιμοποιούμε δυναμικό προγραμματισμό για να πετύχουμε πολυωνυμικό χρόνο εκτέλεσης. Αν K είναι ο τελευταίος διακομιστής που δέχεται θετικό πλήθος αιτήσεων, η αρχή της βελτιστότητας του προβλήματος είναι το γεγονός ότι μία βέλτιστη λύση αν θεωρήσουμε ότι υπάρχουν μόνο οι διακομιστές i έως K της οποίας ο πρώτος διακομιστής που θα αποθηκεύσει το αρχείο είναι ο j συμπεριλαμβάνει τη βέλτιστη λύση για την περίπτωση που υπάρχουν μόνο οι διακομιστές j έως K.

Αυτό φαίνεται στην ακόλουθη αναδρομική σχέση. Με $JS[i]$ συμβολίζουμε το ελάχιστο κόστος που μπορούμε να πετύχουμε για το εύρος διακομιστών i έως K αν εγγυηθούμε ότι θα αποθηκεύσουμε στο διακομιστή i. Με $J[i]$ συμβολίζουμε το ελάχιστο κόστος που μπορούμε να πετύχουμε γενικά για το εύρος διακομιστών i έως K.

Λήμμα. *Το βέλτιστο κόστος αποθήκευσης για οποιοδήποτε εύρος διακομιστών i έως K δίνεται από την ακόλουθη αναδρομική σχέση:*

$$JS[K] = J[K] = c[K]$$
$$JS[i] = J[i+1] + C[i]$$
$$J[i] = \min_{i \le j \le K} \{JS[j] + \sum_{l=i}^{j} (j-l)B[l]\}$$

Απόδειξη. Έστω ότι JS' και J' τα πραγματικά ελάχιστα κόστη. Θα δείξουμε ότι $J = J'$ και $JS' = JS$.

Για $i = K$ η πρόταση ισχύει, αφού $b[K] \neq 0$ και άρα υποχρεωτικά θα αποθηκεύσουμε το αρχείο στο διακομιστή K.

Έστω ότι $\forall n : K \geq n > i \Rightarrow J[n] = J'[n] \wedge JS[n] = JS'[n]$.

Τώρα το κόστος αποθήκευσης για τον i-οστό διακομιστή σίγουρα θα συμπεριλαμβάνεται στο $JS'[i]$. Έστω, όμως, ότι οι διακομιστές στη συνέχεια δεν έχουν κατανεμηθεί βέλτιστα αλλα έχουν ένα κόστος $Q > J[i+1]$. Τότε θα είναι:

$$JS'[i] = C[i] + Q$$
$$\Rightarrow JS'[i] > JS[i]$$

Αυτό όμως είναι αδύνατο διότι η $JS'[i]$ είναι βέλτιστη. Άρα $Q = J[i+1]$ και $JS[i] = JS'[i]$.

Έστω τώρα ότι ο πρώτος διακομιστής που αποθηκεύει το αρχείο στη βέλτιστη λύση για τους διακομιστές i έως K είναι ο j-οστός. Τότε υποθέτουμε ότι η βέλτιστη λύση δεν περιλαμβάνει τη βέλτιστη λύση για τους διακομιστές j έως K, αλλά μία άλλη λύση με κόστος $Q > JS[j]$. Τότε θα είναι:

$$J'[i] = Q + \sum_{l=i}^{j} (j-l)B[l]$$
$$\Rightarrow J'[i] > J[i]$$

Αυτό όμως είναι αδύνατο διότι η $J[i]$ είναι βέλτιστη. Άρα $Q = JS[j]$ και $J[i] = J'[i]$.

Χρησιμοποιώντας τη μαθηματική επαγωγή, αυτό ολοκληρώνει την απόδειξη. □

Επιπλέον, αποθηκεύοντας τους διακομιστές σε έναν πίνακα W μπορούμε να κατασκευάσουμε τη λύση[1].

[1] Η υλοποίηση χρησιμοποιώντας τον πίνακα W για ανακατασκευή του αποτελέσματος, καθώς και οι λύσεις όλων των υπόλοιπων προβλημάτων στη γλώσσα C υπάρχουν στο `http://github.com/dionyziz/ntua-algo`.

Αυτό μας επιτρέπει να κάνουμε μία επιπλέον βελτιστοποίηση, καθώς ένας διακομιστής πιο κοντά στην αρχή δεν μπορεί ποτέ να οδηγήσει στο να αποφευχθεί μία αποθήκευση που από επόμενο διακομιστή θεωρήθηκε βέλτιστη σε πρωτίστερους υπολογισμούς.

Algorithm 5 Άσκηση 5

```
1:  procedure SERVERS(b, c, n)
2:      Ω ← {i : b[i] > 0}
3:      if Ω = ∅ then
4:          return ∅
5:      K ← max(Ω)
6:      JS[K] ← c[K]
7:      J[K] ← c[K]
8:      W[K] ← K
9:      for i ← K − 1 downto 1 do
10:         JS[i] ← J[i + 1] + C[i]
11:         J[i] ← ∞
12:         for j ← i to W[i + 1] do
13:             cost ← JS[j]
14:             for l = i to j do
15:                 cost ← cost + (j − l)B[l]
16:                 if cost >= J[i] then
17:                     break
18:             if cost < J[i] then
19:                 J[i] ← cost
20:                 W[i] ← j
21:     S ← {S_K}
22:     i ← W[1] + 1
23:     while i ≤ K do
24:         S ← S ∪ {S_{i−1}}
25:         i ← W[i] + 1
26:     return S
```

Ο αλγόριθμος θα είναι $O(n^3)$ καθώς για κάθε διακομιστή υπολογίζεται το κόστος JS και J σε αυτόν, κάτι το οποίο χρειάζεται χρόνο $O(n^2)$ για να ελεγχθούν όλοι οι πιθανοί επόμενοι διακομιστές, αφού για κάθε έλεγχο απαιτείται ο υπολογισμός ενός αθροίσματος.

Άσκηση 6

α)

β)

γ)