Θ.Ε. ΠΛΣ50 (2009-10) – ΓΡΑΠΤΗ ΕΡΓΑΣΙΑ Ε4

Ημερομηνία ανάρτησης	**15.11.2009**
Ημερομηνία αποστολής	**10.01.2010**
Ανακοίνωση ενδεικτικής επίλυσης	**13.01.2010**

Θεματολογία-στόχος

Στην εργασία αυτή θα ασχοληθούμε με θέματα σχεδίασης και πολυπλοκότητας αλγορίθμων.

Παρατηρήσεις

Περιμένουμε όλες οι εργασίες να αποσταλούν μέσω email και να είναι γραμμένες σε επεξεργαστή κειμένου σε μορφή doc ή odt. Αρχεία pdf γίνονται δεκτά μόνο όταν συνοδεύονται από το αντίστοιχο doc/odt. Τα τμήματα κώδικα και τα τμήματα δεδομένων θα βρίσκονται σε ξεχωριστά αρχεία και θα υπάρχουν αναφορές σ' αυτά από το κείμενο της εργασίας. Αν υποβάλλετε πηγαίο κώδικα, πρέπει να αναφέρετε το περιβάλλον ανάπτυξης C το οποίο χρησιμοποιήσατε. Δεν χρειάζεται να συμπεριλαμβάνετε στην αποστολή σας εκτελέσιμα αρχεία. Στον Καθηγητή-Σύμβουλό σας, σε κάθε περίπτωση, στέλνετε **ένα μόνο** αρχείο (συμπιεσμένο).

Θέμα 1: Σχεδίαση και Ανάλυση Αλγορίθμων

1α

Έστω Array ένας πίνακας n ακεραίων αριθμών και min, max, number τρεις ακέραιοι αριθμοί. Γράψτε σε ψευδοκώδικα μια συνάρτηση Function(Array, min, max, number) η οποία να επιστρέφει την τιμή 1 αν Array[i] = number για κάποιο 1 ≤ min ≤ i ≤ max ≤ n διαφορετικά να επιστρέφει την τιμή 0. Η χρονική πολυπλοκότητα της συνάρτησης που θα σχεδιάσετε θα πρέπει να είναι O(lg*n*).

```
Function(Array, min, max, number)
{
   if  min > max then return false
   else
    i ← ⌊(min+ max) / 2⌋
    if number = Array[i] then return true
    else if number < Array[i] then return Function(Array, min, i-1, number)
         else return Function(Array, i+1, max, number)
}
```

Η συνάρτηση Function(Array, min, max, number) πραγματοποιεί δυαδική αναζήτηση. Στη χειρότερη περίπτωση θα γίνουν O(lg*n*) αναδρομικές κλήσεις στη συνάρτηση Function αφού ο υποπίνακας Array[min max] διαιρείται στη μέση σε κάθε αναδρομική κλήση. Η χρονική πολυπλοκότητα της συνάρτησης Function, αν εξαιρέσουμε τις αναδρομικές κλήσεις, είναι O(1). Επομένως, η συνολική χρονική πολυπλοκότητα της Function είναι O(lg*n*) .

1β

Έστω ότι σας δίνεται ένας μονοδιάστατος πίνακας C ο οποίος περιέχει n μοναδικούς ακέραιους αριθμούς.

Σχεδιάστε έναν αλγόριθμο - γράψτε τον ψευδοκώδικά του - ο οποίος θα δέχεται ως είσοδο έναν τέτοιο πίνακα C και έναν ακέραιο αριθμό m και θα επιστρέφει ένα ζεύγος διαφορετικών ακέραιων αριθμών που θα βρίσκονται μέσα στον πίνακα C και θα έχουν άθροισμα ίσο με την τιμή του ακέραιου αριθμού m. Εάν δεν υπάρχουν δύο τέτοιοι ακέραιοι αριθμοί μέσα στον πίνακα C τότε ο αλγόριθμος θα πρέπει να επιστρέφει την τιμή -1.

```
Find_Sum(C, n, m)
{
for i ← 1 to n-1 do
    for j ← i+1 to n do
        if C[i] + C[j] = m then return(C[i], C[j])
return -1
}
```

Ο αλγόριθμος ελέγχει όλα τα ζεύγη τιμών (C[i], C[j]), $1 \le i < j \le n$, για το εάν έχουν άθροισμα ίσο με την τιμή του ακέραιου αριθμού m. Πρέπει υποχρεωτικά να ισχύει ότι $i < j$ ώστε τα C[i] και C[j] να είναι διαφορετικά. Αφού $j \le n$, τότε η μεγαλύτερη τιμή που μπορεί να πάρει ο μετρητής i ισούται με n-1.

1γ

Αναφέρετε σε ποια περίπτωση ο αλγόριθμος που σχεδιάσατε στο ερώτημα 1β εμφανίζει τη χειρότερη χρονική πολυπλοκότητά του και αποδείξτε ποια είναι η χρονική πολυπλοκότητα του αλγορίθμου σας στην περίπτωση αυτή.

Η χειρότερη χρονική πολυπλοκότητα του αλγορίθμου εμφανίζεται όταν δεν υπάρχει κανένα ζεύγος διαφορετικών ακέραιων αριθμών μέσα στον πίνακα C το οποίο να έχει άθροισμα ίσο με την τιμή του ακέραιου αριθμού m. Στην περίπτωση αυτή η εντολή return που βρίσκεται μέσα στον εσωτερικό βρόγχο for δε θα εκτελεστεί ποτέ και επομένως οι δύο βρόγχοι for θα εκτελέσουν το μέγιστο αριθμό επαναλήψεων.

Σε κάθε επανάληψη του εσωτερικού for βρόγχου εκτελείται ένας σταθερός αριθμός εντολών, έστω c, ενώ ο ίδιος ο βρόγχος επαναλαμβάνεται (n-1)-(i+1)+1 = n-i-1 φορές. Επομένως, κάθε φορά που εκτελείται ο βρόγχος αυτός ο αριθμός των εντολών που εκτελούνται ισούται με c·(n-i-1).

Ο εξωτερικός βρόγχος for επαναλαμβάνεται για όλες τις τιμές i=0,1,…,n-2.

Όταν i=0 ο εσωτερικός βρόγχος for εκτελεί c·(n-0-1) = c·(n-1) εντολές.

Όταν i=1 ο εσωτερικός βρόγχος for εκτελεί c·(n-1-1) = c·(n-2) εντολές.

...

Όταν i=n-2 ο εσωτερικός βρόγχος εκτελεί c·(n-(n-2)-1) = c·(1) εντολές.

Επομένως, ο συνολικός αριθμός των εντολών που εκτελούνται ισούται με:

$$c\cdot(n-1)+c\cdot(n-2)+\ldots+c\cdot(2)+c\cdot(1)=c\cdot\left(1+2+\ldots+(n-2)+(n-1)\right)=$$

$$c\sum_{k=1}^{n-1}k=c\frac{n(n-1)}{2}=\frac{c}{2}n^2-\frac{c}{2}n$$

Είναι προφανές ότι η χρονική πολυπλοκότητα του αλγορίθμου στη χειρότερη περίπτωση είναι $O(n^2)$.

Θέμα 2: Πολυπλοκότητα Αλγορίθμων

2α

Έστω ότι σας δίνεται ο ακόλουθος ψευδοκώδικας ενός αλγορίθμου:

```
Αλγόριθμος ΠΛΣ_50(n)
      y ← 0
      j ← n
      while j >= 1 do {
            k ← 0
            while k < n*n do {
                  y ← y + j - k
                  k ← k + 1
            }
      j ← j - 1
      }
      return y
```

Αποδείξτε ποια είναι η χρονική πολυπλοκότητα του παραπάνω αλγόριθμου στη χειρότερη περίπτωση.

Πρώτα θα αναλύσουμε τον εσωτερικό βρόγχο while. Σε κάθε επανάληψη του εσωτερικού βρόγχου while εκτελείται ένας σταθερός αριθμός εντολών, έστω c_1. Αυτός ο βρόγχος while εκτελείται $n \cdot n = n^2$ φορές (για όλες τις τιμές του k μεταξύ 0 και n^2-1). Επομένως, ο συνολικός αριθμός εντολών που εκτελούνται σε αυτό το βρόγχο ισούται με $c_1 \cdot n^2$. Αυτός ο εσωτερικός βρόγχος while βρίσκεται μέσα σε ένα άλλο βρόγχο while ο οποίος σε κάθε επανάληψη εκτελεί ένα σταθερό αριθμό εντολών, έστω c_2. Επομένως, ο συνολικός αριθμός εντολών που εκτελούνται σε κάθε επανάληψη του εξωτερικού βρόγχου while ισούται με $c_2 + c_1 \cdot n^2$. Ο εξωτερικός βρόγχος while επαναλαμβάνεται n φορές και άρα ο συνολικός αριθμός εντολών που εκτελούνται σε αυτόν ισούται με $n \cdot (c_2 + c_1 \cdot n^2)$. Έξω από τον εξωτερικό βρόγχο while εκτελείται ένας επίσης σταθερός αριθμός εντολών, έστω c_3. Επομένως, ο συνολικός αριθμός εντολών που εκτελούνται από τον παραπάνω αλγόριθμο ισούται με $n \cdot c_2 + c_1 \cdot n^3 + c_3$.

Είναι προφανές ότι η χρονική πολυπλοκότητα του αλγορίθμου στη χειρότερη περίπτωση είναι $O(n^3)$.

2β

Υπολογίστε μια συνάρτηση T(n) η οποία να δίνει το χρόνο εκτέλεσης του παρακάτω κώδικα ως προς το n. Θεωρείστε ότι ο χρόνος εκτέλεσης του παρακάτω κώδικα ισούται με τον αριθμό των εκτελέσεων της εντολής a ← a + 1. Υπολογίστε επίσης την πολυπλοκότητα της συνάρτησης T(n).

```
for i ← 1 to n do
      for j ← 1 to i do
            for k ← j to i+j do
                  a ← a + 1
            end for
      end for
end for
```

Ξεκινάμε από τον εσωτερικό βρόγχο for και προχωράμε προς τα έξω μέχρι τον εξωτερικό βρόγχο for.

Εσωτερικός βρόγχος for: $\sum_{k=j}^{i+j} 1 = (i+j) - j = i$ (1).

Μεσαίος βρόγχος for (κάνουμε αντικατάσταση αυτό που βρήκαμε στην (1)): $\sum_{j=1}^{i} i = i^2$ (2).

Εξωτερικός βρόγχος for (κάνουμε αντικατάσταση αυτό που βρήκαμε στη (2)): $\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$.

Επομένως, $T(n) = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$.

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης T(n) είναι $\Theta(n^3)$.

2γ

Υπολογίστε μια συνάρτηση S(m) η οποία να δίνει το χρόνο εκτέλεσης του παρακάτω κώδικα ως προς το m. Θεωρείστε ότι ο χρόνος εκτέλεσης του παρακάτω κώδικα ισούται με τον αριθμό των εκτελέσεων της εντολής b ← b + 1. Υπολογίστε επίσης την πολυπλοκότητα της συνάρτησης S(m).

```
for i ← 1 to m do
      for j ← 1 to i^2 do
            for k ← 1 to j do
                  b ← b + 1
            end for
      end for
end for
```

Ξεκινάμε από τον εσωτερικό βρόγχο for και προχωράμε προς τα έξω μέχρι τον εξωτερικό βρόγχο for.

Εσωτερικός βρόγχος for: $\sum_{k=1}^{j} 1 = j$ (1).

Μεσαίος βρόγχος for (κάνουμε αντικατάσταση αυτό που βρήκαμε στην (1)): $\sum_{j=1}^{i^2} j = \frac{i^2\left(i^2+1\right)}{2}$ (2).

Εξωτερικός βρόγχος for (κάνουμε αντικατάσταση αυτό που βρήκαμε στη (2)): $\sum_{i=1}^{n} \frac{i^2\left(i^2+1\right)}{2} = \frac{1}{2}\left(\sum_{i=1}^{n} i^4 + \sum_{i=1}^{n} i^2\right) = \frac{n^5}{10} + \frac{n^4}{4} + \frac{n^3}{3} + \frac{n^2}{4} + \frac{n}{15}$.

Επομένως, $T(n) = \frac{n^5}{10} + \frac{n^4}{4} + \frac{n^3}{3} + \frac{n^2}{4} + \frac{n}{15}$.

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης T(n) είναι $\Theta(n^5)$.

2δ

Έστω δύο αλγόριθμοι A_1 και A_2 οι οποίοι πραγματοποιούν την ίδια εργασία. Έστω ότι για είσοδο μεγέθους n ο αλγόριθμος A_1 χρειάζεται 6*n+24 βήματα ενώ ο αλγόριθμος A_2 χρειάζεται n^2+4*n βήματα. Ποιος από τους δύο αλγόριθμους είναι ο ταχύτερος; Για ποιες τιμές του n θα προτιμούσατε τον αλγόριθμο A_1 και για ποιες τον αλγόριθμο A_2;

Εξισώνουμε τα βήματα που απαιτεί ο αλγόριθμος A_1 με τα βήματα που απαιτεί ο αλγόριθμος A_2 και επιλύουμε ως προς το n:

$6n + 24 = n^2 + 4n \Rightarrow n^2 - 2n - 24 = 0$

Η θετική ρίζα του παραπάνω τριωνύμου ισούται με 6.

Επομένως, ο αλγόριθμος A_1 είναι προτιμότερος (ταχύτερος) για $n \in (6, \infty)$ ενώ ο αλγόριθμος A_2 είναι προτιμότερος (ταχύτερος) για $n \in [0, 6)$.

Θέμα 3: Αναδρομικοί Αλγόριθμοι

.

3α

Έστω ότι ο χρόνος εκτέλεσης ενός αλγορίθμου Α για την επεξεργασία n στοιχείων δίνεται από την παρακάτω αναδρομική εξίσωση:

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + c \cdot n,\ T(1) = 0$$

Βρείτε μια μη αναδρομική εξίσωση η οποία θα περιγράφει το χρόνο εκτέλεσης του αλγορίθμου Α ως προς τους ακέραιους αριθμούς n, k και c και υπολογίστε την πολυπλοκότητα της στη χειρότερη περίπτωση.

Υπόδειξη: Θεωρείστε ότι $n=k^m$, όπου ο m είναι ένας ακέραιος αριθμός που ισούται με $\log_k n$.

Υπάρχουν διάφοροι τρόποι επίλυσης για τέτοιου είδους θέματα. Στη συνέχεια θα παρουσιάσουμε ενδεικτικά δύο διαφορετικές προσεγγίσεις.

1η προσέγγιση:

Η αναδρομική εξίσωση $T\left(k^m\right) = k \cdot T\left(k^{m-1}\right) + c \cdot k^m,\ T(1) = 0$ μπορεί να αναλυθεί ως ακολούθως:

$$\frac{T\left(k^m\right)}{k^m} = \frac{T\left(k^{m-1}\right)}{k^{m-1}} + c$$

$$\frac{T\left(k^{m-1}\right)}{k^{m-1}} = \frac{T\left(k^{m-2}\right)}{k^{m-2}} + c$$

...

$$\frac{T(k)}{k} = \frac{T(1)}{1} + c$$

Προσθέτωντας κατά μέλη τις παραπάνω εξισώσεις προκύπτει ότι:

$$\frac{T\left(k^m\right)}{k^m} = c \cdot m \Rightarrow T\left(k^m\right) = c \cdot k^m \cdot m \Rightarrow T(n) = c \cdot n \cdot \log_k n\ .$$

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης T(n) είναι O(n·logn).

2η προσέγγιση:

Αρχικά υπολογίζουμε κάποιες αρχικές τιμές της αναδρομικής συνάρτησης T(n):

$$T(1)=0$$
$$T(k)=k\cdot T(1)+c\cdot k=c\cdot k$$
$$T(k^2)=k\cdot T(k)+c\cdot k^2=2\cdot c\cdot k^2$$
$$T(k^3)=k\cdot T(k^2)+c\cdot k^3=3\cdot c\cdot k^3$$

Η παραπάνω ακολουθία μας οδηγεί στο να υποθέσουμε ότι ισχύει η εξίσωση $T(k^m)=c\cdot m\cdot k^m$.

Θα το αποδείξουμε στη συνέχεια χρησιμοποιώντας μαθηματική επαγωγή.

α. Για m=0 η αρχική συνθήκη $T(1)=T(k^0)=c\cdot 0\cdot k^0=0$ ισχύει.

β. Έστω ότι ισχύει η εξίσωση για l=1,…,m-1. Τότε για l=m έχουμε:

$$T(k^m)=k\cdot T(k^{m-1})+c\cdot k^m \Rightarrow$$
$$T(k^m)=k\cdot c\cdot(m-1)\cdot k^{m-1}+c\cdot k^m \Rightarrow$$
$$T(k^m)=c\cdot(m-1+1)\cdot k^m \Rightarrow$$
$$T(k^m)=c\cdot m\cdot k^m$$

Άρα η υπόθεση που κάναμε για την T(n) ισχύει για όλες τις τιμές m=0,1,…,∞.

Επομένως, $T(n)=c\cdot n\cdot \log_k n$.

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης T(n) είναι O(n·logn).

3β

Έστω ότι ο χρόνος εκτέλεσης ενός αλγορίθμου Β για την επεξεργασία n στοιχείων δίνεται από την παρακάτω αναδρομική εξίσωση:

$$T(n)=\frac{2}{n}\left(T(0)+\ldots+T(n-1)\right)+c,\ T(0)=0$$

Βρείτε μια μη αναδρομική εξίσωση η οποία θα περιγράφει το χρόνο εκτέλεσης του αλγορίθμου Β ως προς τους ακέραιους αριθμούς n και c και υπολογίστε την πολυπλοκότητα της στη χειρότερη περίπτωση.

Υπόδειξη: Για να υπολογίσετε την ακριβή μη αναδρομική μορφή της συνάρτησης T(n) είναι πιθανόν να χρειαστείτε τη σχέση $\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+\ldots+\frac{1}{n(n+1)}=\frac{n}{n+1}$.

Υπάρχουν διάφοροι τρόποι επίλυσης για τέτοιου είδους θέματα. Στη συνέχεια θα παρουσιάσουμε ενδεικτικά δύο διαφορετικές προσεγγίσεις.

1η προσέγγιση:

Η αναδρομική συνάρτηση υποδηλώνει ότι:

$$n\cdot T(n)=2\left(T(0)+\ldots+T(n-2)+T(n-1)\right)+c\cdot n \quad (1)$$

Ομοίως, $(n-1)\cdot T(n-1)=2\left(T(0)+\ldots+T(n-2)\right)+c\cdot(n-1)$ (2)

Αφαιρώντας την (2) από την (1) κατά μέλη προκύπτει:

$$n\cdot T(n)-(n-1)T(n-1)=2\cdot T(n-1)+c \quad (3)$$

Η (3) μπορεί να γραφεί και ως:

$$n\cdot T(n)=(n+1)T(n-1)+c \Rightarrow \frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{c}{n(n+1)} \quad (4).$$

Η αναδρομική εξίσωση (4) μπορεί να αναλυθεί ως ακολούθως:

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{c}{n(n+1)}$$

$$\frac{T(n-1)}{n}=\frac{T(n-2)}{n-1}+\frac{c}{(n-1)n}$$

...

$$\frac{T(2)}{3}=\frac{T(1)}{2}+\frac{c}{2\cdot 3}$$

$$\frac{T(1)}{2}=\frac{T(0)}{1}+\frac{c}{1\cdot 2}$$

Προσθέτωντας κατά μέλη τις παραπάνω εξισώσεις και επειδή Τ(0)=0 προκύπτει ότι:

$$\frac{T(n)}{n+1}=\frac{c}{1\cdot 2}+\frac{c}{2\cdot 3}+\ldots+\frac{c}{(n-1)n}+\frac{c}{n(n+1)}=c\cdot\frac{n}{n+1}$$

Επομένως, $T(n)=c\cdot n$.

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης Τ(n) είναι O(n)

<u>2ⁿ προσέγγιση:</u>

Αρχικά υπολογίζουμε κάποιες αρχικές τιμές της αναδρομικής συνάρτησης Τ(n):

$$T(0)=0$$

$$T(1)=\frac{2}{1}\cdot 0+c=c$$

$$T(2)=\frac{2}{2}\cdot(0+c)+c=2c$$

$$T(3)=\frac{2}{3}\cdot(0+c+2c)+c=3c$$

$$T(4)=\frac{2}{4}\cdot(0+c+2c+3c)+c=4c$$

Η παραπάνω ακολουθία μας οδηγεί στο να υποθέσουμε ότι ισχύει η εξίσωση $T(n)=c\cdot n$.

Θα το αποδείξουμε στη συνέχεια χρησιμοποιώντας μαθηματική επαγωγή.

α. Για k=0 η αρχική συνθήκη $T(0)=c\cdot 0=0$ ισχύει.

β. Έστω ότι ισχύει η εξίσωση για k=1,…,n-1. Τότε για k=n έχουμε:

$$T(n)=\frac{2}{n}\cdot\left(0+c+2c+...+(n-1)c\right)+c\Rightarrow$$

$$T(n)=\frac{2}{n}\cdot\frac{(n-1)n}{2}c+c$$

$$T(n)=(n-1)c+c=c\cdot n$$

Άρα η υπόθεση που κάναμε για την T(n) ισχύει για όλες τις τιμές k=0,1,…,∞.

Επομένως, $T(n)=c\cdot n$.

Είναι προφανές ότι η πολυπλοκότητα της συνάρτησης T(n) είναι O(n).

ΚΡΙΤΗΡΙΑ ΑΞΙΟΛΟΓΗΣΗΣ	
Θέμα 1: Σχεδίαση και Ανάλυση Αλγορίθμων	**30**
Θέμα 1α	10
Θέμα 1β	10
Θέμα 1γ	10
Θέμα 2: Πολυπλοκότητα	**40**
Θέμα 2α	10
Θέμα 2β	10
Θέμα 2γ	10
Θέμα 2δ	10
Θέμα 3: Αναδρομικοί Αλγόριθμοι	**30**
Θέμα 3α	15
Θέμα 3β	15
ΣΥΝΟΛΟ	**100**
Ο συνολικός βαθμός θα διαιρεθεί δια 10, ώστε να προκύψει ο τελικός βαθμός της εργασίας.	

Καλή Επιτυχία!!!