Ιούνιος 2015

Θέμα 1ο

Η αναδρομική σχέση $T(n)=7T(n/2)+n^2$ περιγράφει το χρόνο εκτέλεσης ενός αλγορίθμου. Ένας ανταγωνιστής αλγόριθμος έστω Α' έχει χρόνο εκτέλεσης $T'(n)=aT(n/2)+n^2$. Ποιά είναι η μεγαλύτερη ακέραια τιμή του a γιατ την οποία ο Α' είναι ασυμπτωτικά ταχύτερος του Α; Αποδείξτε την απάντησή σας.

$T(n)=aT(n/b)+f(n)$

Θεωρία: Master Theorem
case 1: if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
case 2: if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
case 3: if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b)\le\delta f(n)$ for some $\delta<1$.

Λύση

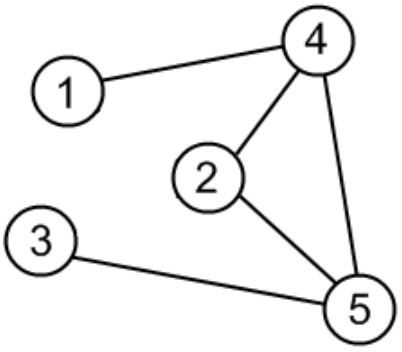
Από τη 1η περίπτωση του **Master Theorem** έχω ότι για τον Α ισχύει $T(n)=O(n^{\log_2 7})$ και ότι για τον Α' ισχύει: $T'(n)=O(n^{\log_2 a})$. Ο Α' θα είναι ασυμπτωτικά ταχύτερος του Α άν και μόνο άν $\log_2 a<\log_2 7$ δηλαδή, $\alpha<7$.

Θέμα 2ο

Περιγράψτε αλγορίθμους που να βρίσκουν όσο πιό γρήγορα γίνεται άν ένας:
(α) ένας μή-κατευθυνόμενος γράφος περιέχει κύκλο
(β) ένας κατευθυνόμενος περιέχει κύκλο
(γ) ένας κατευθυνόμενος γράφος περιέχει **αρνητικό** κύκλο
Αποδείξτε ποιά η πολυπλοκότητά τους.

Λύση

Θεωρία: Συνοπτικά ο DFS
Αρχικά θεωρούμε ότι **όλοι** οι κόμβοι του γράφου είναι white(unvisited).
1. Μάρκαρε το κόμβο u ώς **gray**(visited)
2. Για κάθε ακμή (u, v), όπου το u είναι **white**, τρέξε τον DFS για το u αναδρομικά.
3. Μάρκαρε τον u ώς **black** και επέστρεψε στο πατέρα.



Παράδειγμα
Γι τον γράφο της εικόνας, ακολουθόντας τα βήματα του DFS, οι κόμβοι οι οποίοι θα χρωματιστούν gray κατά σειρά θα είναι οι 1, 4, 2, 5, 3. Και κατά τη διάρκεια του backtrack θα επισκεφθούμε όλους τους κόμβους ξανά αλλά με την ανάποδη σειρά για να τους μαρκάρουμε ώς black. Δηλαδή 3, 5, 2, 4, 1.

(α) Μπορούμε να χρησημοποιήσουμε τον αλγόριθμο DFS για να ανιχνεύσουμε κύκλο σε έναν μη-κατευθυνόμενο γράφο σε χρόνο O(V+E). Κάνουμε ένα DFS traversal του γράφου. Για κάθε κόμβο ‘v’ που έχουμε επισκεφθεί, εαν υπάρχει ένας γειτονικός κόμβος ‘u’ που έχει ήδη επισκεφθεί και ο ‘u’ δεν είναι πατέρας του ‘v’, τότε υπάρχει κύκλος στον γράφο. Έαν τώρα δεν βρούμε κάποιον τέτοιον γειτονικό για κάθε κόμβο τότε λέμε οτι δεν υπάρχει κύκλος στον γράφο.
Συνεπώς κατά τη διάρκεια της αναζήτης (βήμα 2), σε περίπτωση που ο κόμβος που βρούμε είναι μαρκαρισμένος ώς **gray**, τότε βρέθηκε ο κύκλος και ο αλγόριθμος τερματίζει.

(β) Ο DFS για έναν κατευθυνόμενο γράφο παράγει ένα δέντρο. Υπάρχει κύκλος στον γράφο μόνο εαν υπάρχει μια “back edge” παρούσα στον γράφο.
“Back edge” ονομάζουμε μια ακμή που ξεκινάει απο έναν κόμβο και καταλήγει στον ίδιο κόμβο (selfloop) ή σε κάποιον από τους προγόνους στο δέντρο που παρήγαγε ο DFS.
Για να ανιχνεύσουμε μια “back edge”, μπορούμε να κρατήσουμε τους επισκεπτόμενους κόμβους σε μια στοίβα για το traversal του DFS.
Η λύση είναι περίπου ίδια με την παραλλαγή ότι για τους κόμβους που έχουν επισκεφθεί, πρέπει να κρατάμε τις τιμές τους σ ένα stack (Διότι, όντας πλέον directed graph δέν μπορούμε να πάμε πίσω χωρίς ένα stack.) Σε περίπτωση που συναντήσουμε ένα κόμβο ο οποίος είναι ήδη στο stack, τότε βρέθηκε κύκλος και ο αλγόριθμος τερματίζει. Η ακμή που ενώνει τον τωρινό κόμβο με εκείνον που βρίσκεται στην στοίβα είναι μια “back edge”.

Πολυπλοκότητα DFS: O(V+E)

(γ) Αρκεί η χρησιμοποίηση του αλγορίθμου Bellman-Ford.

Θεωρία: Αλγόριθμος Bellman-Ford

```
for all u ε V:
        dist(u) =
        prev(u) = nil
dist(s) = 0;
for (i = 1; i < |V| - 1; i++) {
        for (u,v) in E:{
                relax(u,v)
        }
}
for (u,v) in E: {
        if (v.d > u.d + w(u,v):{
                report negative circle
        }
}
```

Πολυπλοκότητα Bellman-Ford: O(VE)

Θέμα 3ο

Περιγράψτε έναν διαίρει-και-βασίλευε αλγόριθμο για πολλαπλασιασμό δύο ακεραίων Α και Β με βάση β με k ψηφία ο καθένας. Αποδείξτε ποιά είναι η πολυπλοκότητά του.

Λύση

Input: two n-digit integers a and b
Output: product of a and b

π.χ.

```
        1980   = a
x       2315   = b
----------------------
         9900
        1980
       5940
 +    3960
----------------------
      4573700 = a x b
```

Αυτός είναι ο αλγόριθμος που μάθαμε στο σχολείο. Είναι της τάξεως του $O(n^2)$.

Διαιρώντας το πρόβλημα

Χωρίζουμε κάθε ακέραιο σε 2 μισά

```
        aL = 19        80 = aR
        bL = 23        15 = bR
```

άρα:

```
        aL             aR
x       bL             bR
---------------------------------
        aLbR           aRbR
+aLbL aRbL
---------------------------------
aLbL   aLbR + aRbL  aRbR
```

Οπότε για τον αλγόριθμό μας, θα πρέπει να υπολογιστούν τα aL bL, aR bL, aR bR και να τα προσθέσουμε.
Ο χρόνος εκτέλεσης θα είναι:

$$T(n) \leq 4T(n/2) + O(n)$$
$$T(n) = O(n^2)$$

Ξέρουμε ότι τα aL και aR είναι τα αριστερά και δεξιά μισά του a. Αντίστοιχα, τα bL και bR είναι του b.

Algorithm mult(a,b):
if (a || b has one digit)
return a*b;
else {
Έστω ότι n είναι ο μέγιστος αριθμός ψηφίων στα {a,b};
Ξέρουμε ότι τα aL και aR είναι τα αριστερά και δεξιά μισά του a. Αντίστοιχα, τα bL και bR είναι του b;
x_1 = mult(aL, bL);
x_2 = mult(aL, bR);
x_3 = mult(aR, bL);
x_4 = mult(aR, bR);
return $x_1 * (x_2 + x_3)*10^{n/2} + x_4$;
}

Θέμα 4ο

Χρωματισμός (κόμβων) γράφων – Graph coloring
Θέλουμε να χρωματίσουμε τους κόμβους ενός γράφου έτσι ώστε δύο γειτονικοί κόμβοι να έχουν διαφορετικά χρώματα. Το κυρίως πρόβλημα είναι να βρούμε ένα χρωματισμό ώστε ο απαιτούμενος αριθμός διαφορετικών χρωμάτων να είναι ο ελάχιστος.
(α) Περιγράψτε έναν πολυωνυμικό αλγόριθμο (όσο πιό γρήγορο) πού να αποφασίζει άν ένας γράφος δέχεται έναν χρωματισμό με δύο χρώματα. Αποδείξτε την ορθότητα και την πολυπλοκότητα του.
(β) Ποιά είναι η ιδιότητα που πρέπει να έχει ο γράφος ώστε να δέχεται δι-χρωματισμό?
(γ) Περιγράψτε το Graph coloring problem ώς decision problem για k χρώματα, αποδείξτε ότι ανήκει στη κλάση NP.
(δ) Άν έπρεπε να μαντέψετε, θα λέγατε ότι “το πρόβλημα άν ένας γράφος δέχεται έναν χρωματισμό με 3 διαφορετικά χρώματα (3-coloring)” είναι NP-complete, ή όχι? Δώστε πειστικά επιχειρήματα (όχι απόδειξη).

Λύση

(α) Θα πρέπει να είναι bipartite που σημαίνει ότι δέν πρέπει να έχει odd cycle. Αυτό μπορεί να ελεγθεί με τον ακόλουθο αλγόριθμο για BFS.
Ξεκινάμε μ ένα vertex s που το χρωματίζουμε red, και τρέχουμε τον BFS προκειμένου να χρωματίσουμε όλους τους κόμβους που είναι στο ίδιο επίπεδο με το s. Είναι **απαραίτητο** όλοι οι γείτονες του s να χρωματιστούν blue, και όλοι οι γείτονες αυτών των γειτόνων να χρωματιστούν red κ.ο.κ. Άν σε κάποιο σημείο βρούμε ότι ένας vertex είναι χρωματισμένος με 2 χρώματα, τότε ο γράφος **δέν** είναι 2-colorable.

```
BFS(V)
1.    for v in V do:
2.        if Color(v) == null do
3.            initialize an empty queue Q
4.            Color(v) ← red
5.            Enqueue(Q,v)
6.            while Q is not empty do
7.                u ← Dequeue(Q)
8.                for each w in adj[u] do
9.                    if Color(w) == null do
10.                       if Color(u) = red
11.                           Color(w)← blue
12.                       else
13.                           Color(w)← red
14.                       enqueue(Q,w)
15.                   else if Color(w) == Color(u)
16.                       output NO
17.   output YES
```

Running time: Ο χρόνος που χρειάζεται το for loop(γραμμή 8) είναι O(|E|) γιατί κάθε loop αντιστοιχεί σε κάθε ακμή του G. Άρα το running time είναι O(|V|+|E|).

(β) Όμοια με το (α) απαντάμε ότι πρέπει ο γράφος να είναι bipartite. Δηλαδή να μήν έχει odd cycle.

(γ) Το k-coloring ενός γράφου G είναι η ανάθεση ενός χρώματος σε κάθε κόμβο G έτσι ώστε να μήν υπάρχουν παραπάνω από k χρώματα και να μήν υπάρχουν 2 γειτονικοί κόμβοι με το ίδιο χρώμα.
Το decision-problem που αναλογεί στο graph coloring problem είναι το εξής: “Δοθέντος ενός γράφου G και ενός ακεραίου k, υπάρχει valid coloring με το πολύ k χρώματα?”
Το παραπάνω πρόβλημα (έστω π) θα ανήκει στο NP άν υπάρχει ένα πολυωνυμικό certificate που επαληθεύει ένα στιγμιότυπο του “ΝΑΙ”
Χρονική ανάλυση:
guess → O(n) , check → O(nm), total → O(nm)
(δ) Το 3-coloring είναι NP-complete γιατί:

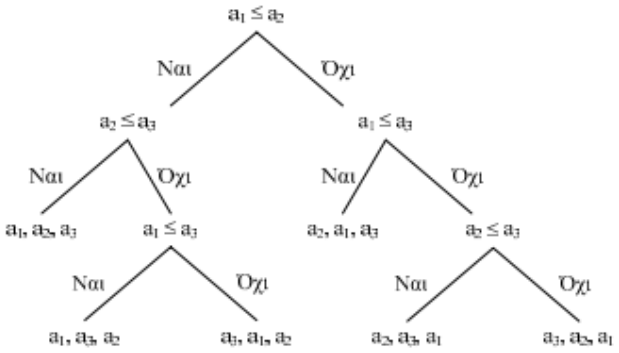
- Certificate: Για κάθε κόμβο διαλέγουμε ένα χρώμα {1,2,3}
- Certifier: Έλεγχω άν για κάθε ακμή (u,v) το χρώμα του u είναι διαφορετικό απ αυτό του v.

Θέμα 5ο

Κάτω φράγματα και πολυπλοκότητα:
(α) Περιγράψτε ένα μή-τετρημένο κάτω φράγμα (π.χ. κάτω φράγμα για ταξινόμηση με συγκρίσεις).
(β) Ποιά είναι η σημασία τών εννοιών P, NP, NP-complete, NP-hard
(γ) Περιγράψτε ένα πρόβλημα που είναι NP-hard, αλλά έχει καλό/γρήγορο 2-προσεγγιστικό αλγόριθμο. Περιγράψτε τον αλγόριθμο.

Λύση

(α) Κάθε ντερμινιστικός συγκριτικός αλγόριθμος ταξινόμησης μπορεί να αναπαρασταθεί με το δέντρο τών συγκρίσεων τις οποίες εκτελεί. Δίδεται παράδειγμα δέντρου συγκρίσεων για 3 αριθμούς (a_1, a_2, a_3).



Το δέντρο συγκρίσεων για την ταξινόμηση n αριθμών πρέπει να έχει τουλάχιστον n! φύλλα / αποτελέσματα, αφού κάθε διαφορετική αναδιάταξη τών αριθμών εισόδου αποτελεί πιθανό αποτέλεσμα του αλγορίθμου και ο αλγόριθμος πρέπει να μπορεί να καταλήξει σ αυτή. Το ύψος του δέντρου τών συγκρίσεων είσαι ίσο με τον αριθμό τών συγκρίσεων που εκτελεί ο αλγόριθμος στη χειρότερη περίπτωση.
Αφού ο αλγόριθμος ανάλογα με το αποτέλεσμα μιας σύγκρισης, μπορείμα διαλέξει το πολύ ανάμεσα σε 2 ενδεχόμενα / μονοπάτια του δέντρου, το δέντρο τών συγκρίσεων είναι ένα δυαδικό δέντρο. Γνωρίζουμε ότι κάθε δυαδικό δέντρο με ύψος h έχει 2^h φύλλα. Επομένως, άν h είναι το ύψος του δέντρου τών συγκρίσεων, πρέπει να είναι $2^k \geq n!$. Λογαριθμώντας αυτή την ανισότητα, έχουμε:

$$h \geq \log(n!) = \sum_{i=1}^{n} i \geq \sum_{i=n/2}^{n} logi \geq \sum_{i=n/2}^{n} \log(n/2) = \Omega(nlogn)$$

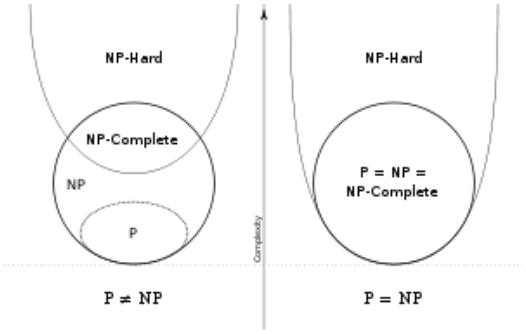
Επομένως για κάθε ντερμινιστικό αλγόριθμο, υπάρχει τουλάχιστον ένα στιγμιότυπο εισόδου για το οποίο ο αλγόριθμος χρειάζεται Ω(nlogn) συγκρίσεις. Συνεπώς, ο χρόνος εκτέλεσης χειρότερης περίπτωσης κάθε ντερμινιστικού συγκριτικού αλγορίθμου είναι Ω(nlogn).

(β)

P: Είναι το complexity class που αναπαριστά ένα set όλων των προβλημάτων απόφασης που μπορούν να λυθούν σε πολυωνυμικό χρόνο.
π.χ. Δωθέντος ενός στιγμιότυπου ενός προβλήματος, η απάντηση “NAI” ή OXI μπορεί να αποφασιστεί σε πολυωνυμικό χρόνο.
NP: Είναι το complexity class που αναπαριστά ένα set όλων των προβλημάτων απόφασης για τα οποία η απάντηση “NAI” έχει αποδείξεις, που μπορούν να επαληθευτούν σε πολυωνυμικό χρόνο.
π.χ. Άν μας δοθεί ένα στιγμιότυπο του προβλήματος και ένα certificate ότι η απάντηση είναι “NAI”, μπορούμε να ελέγξουμε ότι είναι σωστή σε πολυωνυμικό χρόνο.
NP-complete: Αυτό σημαίνει οτι το πρόβλημα μπορεί να λυθεί σε πολυωνυμικό χρόνο χρησημοποιόντας ένα Non-deterministic Turing machine.Βασικά, μια λύση θα πρέπει να μπορεί να τεστάρεται σε πολυωνυμικό χρόνο. Σε αυτή την περίπτωση εαν ένα γνωστό NP πρόβλημα μπορεί να λυθεί χρησημοποιώντας το δοθέν πρόβλημα με μια παραλλαγή της εισόδου (ένα NP πρόβλημα μπορεί να αναχθεί στο δοθέν πρόβλημα) τότε το πρόβλημα είναι NP-complete.

Το κυριότερο πράγμα για ένα NP-complete πρόβλημα είναι οτι δεν μπορεί να λυθεί σε πολυωνυμικό χρόνο με κανένα γνωστό τρόπο.
Όπως έχουν παρατηρήσει άλλοι, υπάρχουν συνήθως προσεγγιστικές λύσεις για NP-complete προβλήματα. Σε αυτή την περίπτωση η προσεγγιστική λύση συνήθως ένα προσεγγιστικο φράγμα χρησημοποιώντας special notation το οποίο μας λέει πόσο κοντά είναι η προσέγγιση.
Με λίγα λόγια είναι το complexity class που αναπαριστά ένα set όλων των προβλημάτων X στο NP για τα οποία είναι δυνατόν να αναχθεί οποιοδίποτε NP πρόβλημα Y στο X σε πολυωνυμικό χρόνο.
Διαισθητικά αυτό σημαίνει ότι μπορούμε να λύσουμε το Y άν ξέρουμε πώς να λύσουμε το X σε πολυωνυμικό χρόνο.
NP-hard: Είναι τα προβλήματα τα οποία ειναι τουλάχιστον τόσο δύσκολα όσο τα δυσκολότερα προβλήματα στο NP. Παρατήρηση τα NP-complete προβλήματα είναι επίσης NP-hard.Όμως δεν είναι όλα τα NP-hard προβλήματα NP (ή ακόμα και decision problem), και ας έχουν την πρόθεση NP.
Συνοπτικά ένα πρόβλημα X είναι NP-hard άν υπάρχει ένα NP-complete πρόβλημα Y τέτοιο ώστε το Y να μπορεί να αναχθεί στο X σε πολυωνυμικό χρόνο.

NP-hard/NP-Complete είναι ένας τρόπος για να δείξουμε οτι συγκεκριμένες κλάσεις προβλημάτων δεν μπορούν να λυθούν σε ρεαλιστικό χρόνο.



(γ) Ένα πρόβλημα το όποιο ανήκει στην κλάση των NP-hard προβλημάτων και έχει έναν καλό και γρήγορο 2-προσεγγιστικό αλγόριθμο ειναι το πρόβλημα της Κάλυψης Κόμβων (Vertex Cover).
Ο αλγόριθμος του οποίου είναι ο εξής:

```
Appr_Vertex_Cover(G)
{
        C = empty set;
        E' = G.E
        while E' != empty set
                let (u,v) be an arbitrary edge to E'
                C = C U {u,v}
                remove from E' every edge incident on either u or v.
}
```