Σεπτέμβρης 2015

Θέμα 1ο

Έχουμε 2 αλγορίθμους τον Χ και τον Υ. Ο αλγόριθμος Χ διασπά ένα πρόβλημα σε τρία (3) προβλήματα μεγέθους n/2. Ο αλγόριθμος Υ διασπά το ίδιο πρόβλημα σε τέσσερα (4) προβλήματα μεγέθους n/3.
Ποιόν αλγόριθμο θα πρέπει να προτιμήσουμε και γιατί (αποδείξτε τις απαντήσεις σας), όταν:
(α) Το κόστος διάσπασης και σύνθεσης των επιμέρους λύσεων είναι n^2.
(β) Το κόστος διάσπασης και σύνθεσης των επιμέρους λύσεων είναι n.

Λύση

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

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 $\delta<1$.

(α) Για τον Χ ισχύει: $T_x(n)=3T(n/2)+n^2$
Απ τη γενική μορφή του Master Theorem: a = 3, b = 2, $f(n)=n^2$
$\log_b a=\log_2 3\simeq 1.58$ άρα, $T(n)\varepsilon\Theta(n^2)$ (case 3)

Για τον Υ ισχύει: $T_y(n)=4T(n/3)+n^2$
Απ τη γενική μορφή του Master Theorem: a = 4, b = 3, $f(n)=n^2$
$\log_b a=\log_3 4\simeq 1.26$ άρα, $T(n)\varepsilon\Theta(n^{\log_3 4})\simeq\Theta(n^{1.262})T(n)\varepsilon\Theta(n^2)$ (case 3)

Άρα συμπεραίνουμε ότι δέν παίζει ρόλο σ αυτή τη περίπτωση ποιόν αλγόριθμο θα επιλέξουμε γιατί και οι 2 αλγόριθμοι είναι ασυμπτώτικα ίσοι γιατί συγλίνουν στο O(n^2) και άρα τρέχουν στον ίδιο χρόνο.

(β) Για τον Χ ισχύει: $T_x(n)=3T(n/2)+n$
Απ τη γενική μορφή του Master Theorem: a = 3, b = 2, $f(n)=n$
$\log_b a=\log_2 3\simeq 1.58$ άρα, $T(n)\varepsilon\Theta(n^{\log_2 3})\simeq\Theta(n^{1.585})$ (case 1)

Για τον Υ ισχύει: $T_y(n)=4T(n/3)+n$
Απ τη γενική μορφή του Master Theorem: a = 4, b = 3, $f(n)=n$ (case 1)
$\log_b a=\log_3 4\simeq 1.26$ άρα, $T(n)\varepsilon\Theta(n^{\log_3 4})\simeq\Theta(n^{1.262})$

Άρα σε αυτή τη περίπτωση θα προτιμήσουμε τον 2ο αλγόριθμο γιατί όπως βλέπουμε είναι πιο γρήγορος σε σχέση με τον 1ο.

Θέμα 2ο

Maximum Spanning Trees and Longest Paths

(Α) Έστω ένας γράφος G=(V,E) με βάρη στις ακμές. Περιγράψτε έναν αλγόριθμο που βρίσκει το δένδρο **μέγιστου** κόστους (**Maximum** Spanning Trees) σε πολυωνυμικό χρονό, όσο πιο γρηγορα γίνεται. Αποδείξτε:

(α) Ότι ο αλγόριθμος δουλεύει σωστά

(β) Πόσο χρόνο χρειάζεται?

(Β) Μπορείτε να χρησιμοποίησετε μια παρόμοια τεχνική για να βρείτε μέγιστα μονοπάτια(Longest paths); Άν **ΝΑΙ**, περιγράψτε πώς και πόσο χρόνο χρειάζεται, άν **ΟΧΙ** "αποδείξτε" το.

Λύση

(Α) (α) Ταξινομούμε τις ακμές σε φθίνουσε αντί για αύξουσα σειρά **ή** τις πολλαπλασιάζουμε με -1 (**Δηλαδή ο αλγόριθμος του Prim**). Και επειδή το Minimum Spanning Tree είναι να έχεις minimum τα weights τών ακμών, όταν κάνουμε πολλαπλασιασμό με -1 τότε το πρόβλημα γίνεται max.

(β) Χρόνος = O(χρόνος για MST + |E|) γιατί θα πρέπει να περάσουμε μια φορά απ όλες τις ακμές.

(Β) Όχι γιατί μπορεί να έχουμε αρνητικούς κύκλους οπότε θα μπούμε σε infinite loop. Γενικά το πρόβλημα longest path είναι NP-complete εκτός άν ο γράφος είναι DAG (Directed Acyclic Graph). Όπου σ αυτή τη περίπτωση κάνουμε topological short και βρίσκουμε το longest path.

Θέμα 3ο

Περιγράψτε τον **καλύτερο** αλγόριθμο που γνωρίζεται (divide and conquer) για πολλαπλασιασμό δύο πινάκων A και B. Αποδείξτε ποιά είναι η πολυπλοκότητά του.

Λύση

Είναι ο αλγόριθμος του Strassen

```
Strassen(A,B) {
1.      if (n == 1) Output A x B;
2.      else {
3.              Compute A^11, B^11, …, A^22, B^22        //υπολογίζοντας το m = n/2
4.              P1 ← Strassen(A^11, B^12 – B^22)
5.              P2 ← Strassen(A^11 + A^12, B^22)
6.              P3 ← Strassen(A^21 + A^22, B^11)
7.              P4 ← Strassen(A^22, B^21 – B^11)
8.              P5 ← Strassen(A^11 + A^22, B^11 + B^22)
9.              P6 ← Strassen(A^12 – A^22, B^21 + B^22)
10.             P7 ← Strassen(A^11 – A^21, B^11 + B^12)
11.             C^11 ← P5 + P4 – P2 + P6
12.             C^12 ← P1 + P2
13.             C^21 ← P3 + P4
14.             C^22 ← P1 + P5 – P3 – P7
15.             Output C
16.     }
}
```

Οι διαδικασίες στη γραμμή 3 διαρκούν σταθερό χρόνο. Το συνολικό κόστος (γραμμές 11-14) είναι $\Theta(n^2)$. Υπάρχουν 7 αναδρομικές κλήσεις(γραμμές 4-10) Άρα έχουμε: $T(n)=7T(n/2)+\Theta(n^2)$.

Θέμα 4ο

(α) Ποιά είναι η σημασία τών εννοιών P, NP, NP-complete, NP-hard
(β) Άν είναι γνωστό ότι το πρόβλημα ΑΝΕΞΑΡΤΗΤΟ ΣΥΝΟΛΟ(INDEPENDENT SET) είναι NP-complete, να δείξετε ότι και το πρόβλημα ΚΑΛΥΨΗ ΚΟΜΒΩΝ(VERTEX COVER) είναι NP-complete.
(γ) Περιγράψτε έναν πολυωνυμικό αλγόριθμο (τον πιό γρήγορο) πού να αποφασίζει άν ένας γράφος είναι διμερής (bipartite). Ποιός είναι ο ελάχιστος αριθμός k για τον οποίο το πρόβλημα ΚΑΛΥΨΗ ΚΟΜΒΩΝ έχει λύση όταν γράφος είναι διμερής (bipartite) με n_1 κόμβους στην μιά πλευρά και n_2 κόμβους στην άλλη πλευρά; Αποδείξτε την ορθότητα και την πολυπλοκότητά του.
(δ) Υπάρχει πολυωνυμικός αλγόριθμος για το πρόβλημα ΚΑΛΥΨΗ ΚΟΜΒΩΝ όταν ο γράφος είναι δένδρο; Άν ΝΑΙ δώστε τον αλγόριθμο. Άν ΟΧΙ εξηγήστε γιατί.
(ε) Έχει το ΚΑΛΥΨΗ ΚΟΜΒΩΝ πρόβλημα ένα καλό/γρήγορο 2-προσεγγιστικό αλγόριθμο που να είναι το πολύ δύο φορές μεγαλύτερη από μια βέλτιστη; Άν ΝΑΙ δώστε τον αλγόριθμο που δίνει μια προσεγγιστική λύση. Άν ΟΧΙ εξηγήστε γιατί.

Λύση

(α) **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 “ΝΑΙ” ή ΟΧΙ μπορεί να αποφασιστεί σε πολυωνυμικό χρόνο.
NP: Είναι το complexity class που αναπαριστά ένα set όλων των προβλημάτων απόφασης για τα οποία η απάντηση “ΝΑΙ” 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 “ΝΑΙ”, μπορούμε να ελέγξουμε ότι είναι σωστή σε πολυωνυμικό χρόνο.
NP-complete: 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: 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.

(β) **Vertex Cover**: Δοθέντος ενός γράφου G και ενός αριθμού k, περιέχεται στον G κάλυψη κόμβων τουλάχιστον k?

Independent Set: Δοθέντος ενός γράφου G και ενός αριθμού k, περιέχεται στον G ένα set από τουλάχιστον k ανεξάρτητες ακμές?

Θεώρημα: Reduction and NP-completeness
if Y is NP-complete and
1. X is in NP
2. if problem Y can be reduced to problem X(this means Y is polynomial-time reducible to X, it also means that X is at least as hard as Y. Because if you can solve X you can solve Y)

Βάση της εκφώνησης, ξέρουμε ότι το INDEPENDENT SET είναι NP-complete. Άρα αρκεί να δείξουμε ότι το VERTEX COVER είναι NP.

Το Certificate είναι ότι υπάρχει μια λίστα από k κόμβους από μια συλλογή. Άρα μπορούμε να ελέγξουμε σε πολυωνυμικό χρόνο άν καλύπτονται. Άρα, εφόσον έχουμε certificate που λύνεται σε πολυωνυμικό χρόνο, τότε το VERTEX COVER είναι NP.

Τώρα με χρήση του θεωρήματος. Ξέρουμε ότι το INDEPENDENT SET είναι NP-complete και το VERTEX COVER ανήκει στο NP. Εφόσον το INDEPENDENT SET μπορεί να αναχθεί στο VERTEX COVER τότε το VERTEX COVER θα είναι NP-complete.

(γ) Δηλαδή να έχουμε 2 set από nodes στα οποία ισχύει η ιδιότητα : Οι κόμβοι του ενός set να είναι γείτονες μόνο με τους κόμβους του άλλου set και με κανένα από τα set που ανήκουν.
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|).

(δ)
```
VC(node* root) {
	if (root == NULL) return 0;
	if (root→left == NULL || root→right == NULL) return 0;
	size_with_root = 1 + VC(root→left) + VC(root→right);
	size_wo_root = 0;
	if (root→left) {
		size_wo_root += 1 + VC(root→left→left) + VC(root→left→right);
	}
	if (root→right) {
		size_wo_root += 1+ VC(root→right→left) + VC(root→right→left);
	}
	return min(size_with_root, size_wo_root);
}
```
(ε)
Ναί.
```
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.
}
```

Θέμα 5ο

Περιγράψτε 2 divide and conquer αλγορίθμους για sorting. Ποιές είναι οι κύριες διαφορές τους;

Λύση

```
Mergesort(L1, L2)
 {
	list X = empty;
	while (neither L1 not L2 empty)
	{
		remove smaller of the two from it's list
		add to end of X.
	}
	catenate remaining list to end of X.
	return X;
}
```

```
Quicksort(L)
 {
        if (length(L) < 2) return L
        else
        {
                pick some x in L
                 L1 = { y in L : y < x }
                L2 = { y in L : y > x }
                L3 = { y in L : y = x }
                quicksort(L1)
                quicksort(L2)
                return concatenation of L1, L3, and L2

        }

}
```

Η κύρια διαφορά είναι ότι η Mergesort χρησιμοποιεί ένα βοηθητικό πίνακα για να κάνει το merge, δηλαδή ενώνει τους δύο πίνακες εισόδου σε γραμμικό χρόνο. Ενώ η quicksort, κάνει partition παίρνοντας το pivot και το βάζει στη σωστή θέση. Ο πίνακας δέ θα είναι sorted αλλά, όλα τα στοιχεία αριστερά του pivot(μικρότερα ή ίσα) θα είναι sorted αλλά απ τα δεξιά όχι. Το partition μπορεί να γίνει σε γραμμικό χρόνο. Η πολυπλοκότητα της quicksort εξαρτάται απ το τρόπο με τον οποίο επιλέγεται το pivot. Worst-case άν ο πίνακας είναι sorted τοτε το pivot είναι το 1ο στοιχείο της λίστας και άρα θα πρέπει να εκτελεστεί η quicksort σ ένα sub-array μεγέθους n-1 άρα μιλάμε για πολυπλοκότητα της τάξης του $O(n^2)$.