Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Θεμελιώδη Θέματα Επιστήμης Υπολογιστών, 2019-20

2η Σειρά Γραπτών Ασκήσεων

(αλγοριθμικές τεχνικές - αριθμητικοί αλγόριθμοι - αλγόριθμοι γράφων - δυναμικός προγραμματισμός)

Ονοματεπώνυμο: *Θοδωρής Αράπης*

(el18028, theodoraraps2000@gmail.com)

Άσκηση 1.

(α) Ο αναδρομικός αλγόριθμος για τους πύργους του Hanoi χρειάζεται $K_ν$ κινήσεις για να ολοκληρωθεί. Εκτελώντας τον αλγόριθμο για διάφορες τιμές του n (πλήθος δίσκων) παρατηρούμε:

$$n = 1 \longrightarrow K_1 = 1$$

$$n = 2 \longrightarrow K_2 = 3$$

$$n = 3 \longrightarrow K_3 = 7$$

$$n = 4 \longrightarrow K_4 = 15$$

$$\bullet\bullet\bullet$$

$$n = \nu \longrightarrow K_\nu = 2K_{\nu-1} + 1$$

Θα έχουμε: $K_n = 2(2K_{n-1} + 1) - 1 = 2^2(2K_{n-2} + 1) + 2 - 1\ \kappa.ο.\kappa.$

Και επομένως: $K_n = 2^{n-1}K_1 + 2^{n-2} + \cdots + 2 + 1$

Συνεπώς, ως γεωμετρική σειρά, θα ισχύει: $\boxed{K_n = 2^n - 1}$

(β) Έστω A η αρχική στήλη, B η ενδιάμεση στήλη, C η τελική στήλη. Θα χρησιμοποιήσουμε την μέθοδο της επαγωγής. Αρχικά, για έναν δίσκο χρειάζονται $K_1 = 1$ κινήσεις. Έστω ότι για n δίσκους χρειαζόμαστε $K_n = 2^n - 1$ κινήσεις. Για n+1 δίσκους θα πρέπει οι n πρώτοι δίσκοι να έχουν μετακινηθεί στην στήλη B ($2^n - 1$ κινήσεις). Έπειτα ο κατώτερος δίσκος θα μετακινηθεί στην στήλη C (1 κίνηση). Τέλος, οι n πρώτοι δίσκοι θα μετακινηθούν από την στήλη B στην στήλη C ($2^n - 1$ κινήσεις). Άρα, για n+1 δίσκους θα χρειαστούμε $K_{n+1} = 2^n - 1 + 1 + 2^n - 1 \Rightarrow K_{n+1} = 2^{n+1} - 1.$

Συνεπώς, ο αριθμός των κινήσεων του επαναληπτικού αλγόριθμου είναι ίσος με τον αριθμό κινήσεων του αναδρομικού αλγόριθμου.

(γ) Θα χρησιμοποιήσουμε πάλι την επαγωγική μέθοδο. Αρχικά, η βέλτιστη λύση για έναν δίσκο απαιτεί $K_1 = 1$ κινήσεις. Υποθέτουμε ότι έχουμε μία βέλτιστη λύση για n δίσκους. Για να μετακινήσουμε βέλτιστα n+1 δίσκους

θα πρέπει (όπως στο ερώτημα β) να μετακινήσουμε τους n πρώτους δίσκους στην στήλη B, μετά να μετακινήσουμε τον κατώτερο δίσκο στην στήλη C και τέλος να μετακινήσουμε τους n δίσκους στην στήλη C. Δηλαδή εφαρμόζουμε τον βέλτιστο αλγόριθμο για n δίσκους 2 φορές και επιπλέον μία κίνηση για τον κατώτερο δίσκο. Έχουμε λοιπόν:

Για 1 δίσκο χρειαζόμαστε $K_1 = 2^1 - 1 = 1$ κινήσεις (βέλτιστα). Έστω ότι για n δίσκους χρειαζόμαστε $K_n = 2^n - 1$ κινήσεις με τον βέλτιστο αλγόριθμο. Ο βέλτιστος αλγόριθμος για n+1 δίσκους όπως περιγράφηκε παραπάνω θα χρειαστεί $K_{n+1} = 2^{n+1} - 1$ κινήσεις. Συνεπώς, ο βέλτιστος αλγόριθμος για την επίλυση του πύργου Hanoi χρειάζεται $K_n = 2^n - 1$ κινήσεις, όσες χρειάζεται δηλαδή ο επαναληπτικός και ο αναδρομικός αλγόριθμος.

(δ) Για n δίσκους και 4 στήλες θα ακολουθήσουμε τον εξής αλγόριθμο:

Μεταφέρουμε k δίσκους σε κάποια ενδιάμεση στήλη σε $K_{k,4}$ κινήσεις. Στη συνέχεια μεταφέρουμε τους υπόλοιπους n-k δίσκους στην τελική στήλη, χρησιμοποιώντας όμως μόνο 3 στήλες (αφού η μία έχει καταληφθεί από του k μικρότερους δίσκους). Οπότε θα χρειαστούμε $K_{n-k,3}$ κινήσεις. Τέλος, μεταφέρουμε τους k πρώτους δίσκους στην τελική στήλη με $K_{k,4}$ κινήσεις.

Θα έχουμε λοιπόν: $K_{n,4} = 2K_{k,4} + K_{n-k,3}$, $\forall k \in [1, n)$

Όμως θα ισχύει: $\boxed{K_{n,4} = 2K_{k,4} + 2^{n-\kappa} - 1}$

Άσκηση 2.

Η υλοποίηση έγινε σε γλώσσα C++ η οποί δυστυχώς δεν υποστηρίζει είσοδο τόσο μεγάλων αριθμών.

(α)

```cpp
#include <iostream>
using namespace std;

int fastmodpower(int a, long long n, long long p)
{
    int res = 1;
    while (n > 0)
    {
        if (n%2 == 1)
            res = (res*a) % p;
        n = n / 2;
        a = (a*a) % p;
    }
    return res;
}

int gcd(int a, int b)
{
    if (a < b)
        return gcd(b, a);
    else if (a%b == 0)
        return b;
    else return gcd(b, a%b);
}

bool isPrime(long long n)
{
    if (n <= 1 or n == 4) return false;
    if (n <= 3) return true;
    for (int k=0; k<30; k++)
    {
        int a = 2 + rand() % (n - 4);
        if (gcd(n, a) != 1)
            return false;
        if (fastmodpower(a, n - 1, n) != 1)
            return false;
    }
    return true;
}

int main()
{
    long long N;
    cin >> N;
    if (isPrime(N)) cout << N << " is Prime/n.";
    else cout << N << " is not Prime/n.";
    return 0;
}
```

Παρατηρούμε ότι οι αριθμοί Carmichael περνούν τον έλεγχο Fermat.

(β)

```cpp
#include <iostream>
using namespace std;

int fastmodpower(int a, unsigned long long n, int p)
{
    int res = 1;
    while (n > 0)
    {
        if (n%2 == 1)
            res = (res*a) % p;
        n = n / 2;
        a = (a*a) % p;
    }
    return res;
}

bool MiillerRabin(unsigned long long d, int n)
{
    int a = 2 + rand() % (n - 4);
    int x = fastmodpower(a, d, n);
    if (x == 1 or x == n - 1)
        return true;
    while (d != n - 1)
    {
        x = (x * x) % n;
        d *= 2;
        if (x == 1) return false;
        if (x == n - 1) return true;
    }
    return false;
}

bool PrimeTest(int n)
{
    if (n <= 1 || n == 4)  return false;
    if (n <= 3) return true;
    unsigned long long d = n - 1;
    while (d % 2 == 0)
        d = d/2;
    for (int i = 0; i < 30; i++)
        if (!MiillerRabin(d, n))
            return false;
    return true;
}

int main()
{
    unsigned long long N;
    cout << "Insert a number to test if it is prime: ";
    cin >> N;
    if (PrimeTest(N))
        cout << "It is Prime\n";
    else
        cout << "It isn't Prime\n";
    return 0;
}
```

Άσκηση 3.

(α) Εφόσον το δίκτυο (γράφος) δεν έχει κύκλους και είναι κατευθυνόμενος, τότε είναι της μορφής DAG (Directed Acyclic Graph). Συνεπώς ο καλύτερος αλγόριθμος για να βρούμε τις ελάχιστες διαδρομές από έναν κόμβο προς όλους τους άλλους, για DAG, είναι:

1) Αρχικοποιούμε έναν πίνακα distance[] όπου αποθηκέυουμε το κόστος προς κάθε κόμβο, θέτοντας INF (infinity) για όλους τους κόμβους εκτός του αρχικού όπου θετουμε distance[s] = 0 (s = source).

2) Δημιουργούμε ένα stack όπου ταξινομούμε τοπολογικά τουε κόμβους.

3) Ύστερα, για κάθε τοπολογικά ταξινομημένο κόμβο u στο stack (εκτός του αρχικού) που έχει ακμή που καταλήγει σε κάποιον κόμβο v, θέτω: $distance[v] = \min_{(u, v)}\{distance[v], distance[u] + weight(u, v)\}$.

4) Επιστρέφουμε τον πίνακα distance[].

Time complexity: O(|V|+|E|).

(β) Εφόσον το δίκτυο (γράφος) μπορεί να περιέχει κυκλους, τότε δεν είναι DAG. Συνεπώς, αφου δεν έχουμε αρνητικά βάρη, μπορούμε να χρησιμοποιήσουμε είτε τον αλγόριθμο Dijstra είτε τον Bellman-Ford. Επιλέγουμε τον Dijkstra, καθώς μπορεί να είναι ελαφρώς πιο γρήγορος ανάλογα με την υλοποίηση του αλγορίθμου (min-priority queue) και τη μορφή του γράφου. Ο αλγόριθμος είναι ο εξής:

1) Αρχικοποιούμε έναν πίνακα distance[] όπου αποθηκέυουμε το κόστος προς κάθε κόμβο, θέτοντας INF (infinity) για όλους τους κόμβους εκτός του αρχικού όπου θετουμε distance[s] = 0 (s = source).

2) Κάνουμε push τον αρχικό κόμβο σε μία min-priority queue ως (distance, κόμβος), προκειμένου οι συγκρίσεις στην ουρά να γίνονται με βάση τις αποστάσεις των κόμβων.

3) Κάνουμε pop από την ουρά τον κόμβο u με την μικρότερη απόσταση (Στην αρχή θα γίνει pop ο αρχικός κόμβος).

4) Ενημερώνουμε τις αποστάσεις των κόμβων v που ενώνονται με ακμές με τον κόμβο u που μόλις κάναμε pop, στην περίπτωση που: distance[u] + weight(u, v) < d[v]. Ύστερα κάνουμε push τον κόμβο μαζί με την νέα απόσταση στην ουρά.

5) Αν ο κόμβος που έγινε pop έχει επισκεφθεί, τότε συνεχίζουμε χωρίς να τον χρησιμοποιήσουμε.

6) Εφαρμόζουμε τον αλγόριθμο μέχρι να αδειάσει η ουρά και επιστρεφουμε τον πίνακα distance[].

Time complexity: O(|V|+|E|log|V|).

(γ) Εφόσον το δίκτυο (γράφος) μπορεί να έχουει σε μερικούς κόμβους πρσοφορές που υπερβαίνουν το κόστος (ακμές με αρνητικό κόστος) και κύκλους που δενπεριέχουν όμως αρνητικές ακμές, είναι φανερό ότι πρέπει να χρησιμοποιήσουμε τον αλγοριθμο Bellman-Ford (αφου ο Dijkstra δεν λειρουργεί με αρνητικές ακμές και ο γράφος δεν είναι DAG). Ο αλγόριθμος είναι ο εξής:

1) Αρχικοποιούμε έναν πίνακα distance[] όπου αποθηκέυουμε το κόστος προς κάθε κόμβο, θέτοντας INF (infinity) για όλους τους κόμβους εκτός του αρχικού όπου θετουμε distance[s] = 0 (s = source).

2) Ένας εξωτερικός βρόγχος κάνει |V|-1 επαναλήψεις.

3) Σε κάθε επανλάψη του εξωτερικού βρόγχου εξετάζουμε όλες τις ακμές και ελέγχουμε αν ισχύει: distance[u] + weight(u, v) < d[v]. Αν ισχυει τότε θέτουμε d[v] = distance[u] + weight(u, v).

4) Μετά από |V|-1 επαναλήψεις, επιστρέφουμε τον πίνακα distance[].

Time complexity: O(|V|·|E|).

Άσκηση 4.

(α) Η λύση του προβλήματος θα είναι αν υπάρχει διαδρομή στο MST (Minimum Spanning Tree) του γράφο, όπου οι ακμές που ενώνουν το s με το t να έχουν όλες κόστος (χιλιομετρική απόσταση) μικρότερο του L. Από τους αλγορίθμους Prim, Kruskal και Boruvka, ο πιο γρήγορος είναι ο Prim (ανάλογα με τον τρόπο υλοποίησης). Θα τον υλοποιήσουμε χρησιμοποιώντας priority queue. Ο αλγόριθμος θα είναι ο εξής:

1) Δημιουργούμε μία priority queue με μέγεθος όσο το πλήθος των κόμβων και με αντικείμενα τα ζεύγη (weight, vertex). Χρησιμοποιούμε τα βάρη (χιλιομετρικές αποστάσεις) ως το αντικείμενο σύγκρισης της ουράς.

2) Δημιουργούμε έναν Boolean πίνακα mst[] προκειμένου να ελέγχουμε ποιοι κόμβοι βρίσκονται στο MST.

3) Δημιουργούμε πίνακα parent[] όπου θα αποθηκεύουμε τον κόμβο από τον οποίο μπορείς να επισκεφτείς τον εκάστοτε κόμβο, καθώς και τον πίνακα key[] ώστε να ελέγχουμε τις αποστάσεις των πόλεων, αρχικοποιούμε όλα τα “key” ως INF, εκτός του αρχικού κόμβου s όπου θέτουμε το “key” του ίσο με 0 και ξεκινάμε από αυτόν (τον τοποθετούμε σαν ζεύγος (0, 0) στην ουρά).

4) Όσο η ουρά δεν είναι κενή:

α) βγάζουμε τον κόμβο (έστω u) με το μικρότερο βάρος (από το s στο u) από την ουρά και τον τοποθετούμε στο mst[] (mst[u] = true). Κάθε φορά ελέγχουμε αν το βάρος είναι μικρότερο του L και αν όχι τότε το πρόγραμμα θα τερματιζει. Με μία boolean μεταβλητή μμπορούμε να ελέγξουμε αν ο κόμβος u είναι ο κόμβος t που θέλουμε να φτάσουμε. Αν πετύχουμε ακμή με βάρος μεγαλύτερο του L χώρις ο κόμβος t να έχει τοποθετηθεί στο mst τότε η boolean μεταβλητή θα παραμήνει “false” ενώ, όταν το t τοποθετηθεί στο mst, η μεταβλητή θα λάβει την τιμή “true”. Έτσι στο τέλος θα ξέρουμε αν υπάρχει διαδρομή από το s στο t.

β) Διατρέχουμε επαναλαμβανόμενα όλους τους συνεκτικούς, στον κόμβο u, κόμβους και εξετάζουμε το εξής:

Αν mst[v] = false και key[v] > weight(u, v) τότε:

(i) Ενημερώνουμε το key[v] ώστε key[v] = weight(u, v).

(ii) Εισάγουμε τον v στην ουρά.

(iii) parent[v] = u.

5) Μόλις τερματίσει το πρόγραμμα, αν η μεταβλητή boolean είναι “true” τότε το ζητούμενη είναι εφικτό. Αν είναι “false” τότε δεν είναι εφικτό.

Time complexity: O(|E|log|V|).

(β) Το πρόβλημα είναι παρόμοιο με το πρόβλημα του ερωτήματος (α). Θα χρειαστούμε πάλι το MST του γράφου. Για να πετύχουμε την ζητούμενη πολυπλοκότητα θα χρησιμοποιήσουμε τον αλγόριθμο kruskal, ο οποίος είναι ο εξής:

1) Ταξινομούμε όλες τις ακμές σε άυξουσα σειρά βάρους.

2) Επιλέγουμε την μικρότερη (σε βάρος) ακμή (μικρότερη του L αλλιώς το πρόγραμμα τερματίζει) και ελέγχουμε αν σχηματίζει κύκλο με το MST που έχει σχηματιστεί προς στιγμήν (με χρήση του Union-Find algorithm). Αν όχι τότε συμπεριλαμβάνουμε και αυτήν την ακμή στο MST. Με μια boolean μεταβλητή ελέγχουμε αν βρισκονται στο ίδιο σύνολο οι κόμβοι s, t. Αν συμπεριλαμβάνονται στο ίδιο σύνολο (έλεγχος με Union-Find) τότε η boolean μεταβλητή λαμβάνει την τιμή “true” αλλιώς την τιμή “false”.

3) Επαναλαμβάνουμε το βήμα 2 μέχρι να έχουμε συνολικά |V|-1 ακμές στο MST, ή να τερματίσει το προγραμμα επειδη βρήκε ακμή με βάρος μεγαλύτερο του L. Αν η μεταβλητή boolean είναι “true” τότε υπάρχει διαδρομή, αν είναι false δεν υπάρχει.

Time complexity: O(|E|log|E|).

Άσκηση 5.

Ο αλγόριθμος θα είναι ο εξής:

1) Δημιουργούμε έναν πίνακα mc[] (mimimum cost) τύπου int και μεγέθους T (όσες και οι μέρες) και αρχικοποιούμε κάθε τιμή του πίνακα να είναι μηδέν.

2) Δημιουργούμε άλλον έναν πίνακα ticket[] τύπου int και μεγέθους T όπου θα αποθηκεύουμε το ελιδος του εισητηρίου ανά ημέρα. (π.χ. ticket[15] = 7, εισητήριο p_7).

3) Διατρέχουμε τον πίνακα S και κάθε φορά που συναντάμε τιμή 1 θα ακολουθούμε την παρακάτω διαδικασία:

Αν t η μέρα τότε:

mc[t] = min({mc[t-1] + p[0], mc[max(0, t-3)] + p[1], mc[max(0, t-7)] + p[2], … mc[max(0, t-f)] + p[k]})

Όπου f οι μέρες που διαρκει το εισητήριο p_k.

Ανάλογα ποιο εισητήριο επιλέγεται για τον πίνακα mc, αποθηκεύουμε τον δεικτη i του p_i στον πίνακα ticket[] ως ticket[t] = i.

Κάθε φορά που θα συναντάμε 0:

mc[t] = mc[t-1]

4) Μόλις διατρέξουμε τον πίνακα S, τότε διατρέχουμε ανάποδα τον πίνακα ticket[] ως εξής:

```
q = T;
While (q>0)
{
  cout << “P” << ticket[q] <<endl;
  q = q - ticket[q];
}
```

Έτσι θα εκτιπώνεται (κάθετα) ο συνδυασμός εισητηρίων.

Time complexity: O(kT).

(T επαναλήψεις και k συγκρίσεις σε κάθε επανάληψη).

Άσκηση 6.

(α) *n = 100, k = 1:*

Αφού μπορούμε να σπάσουμε μόνο ένα ποτήρι θα πρέπει αναγκαστικά να ξεκινήσουμε από το 1 εκατοστό ύψος και να ανεβαίνουμε κάθε φορά 1 εκατοστό. Μόλις σπάσει το ποτήρι, έστω ύψος h, θα έχουμε το μέγιστο ύψος: h-1.

Μέγιστο πλήθος δοκιμών: 99.

n = 100, k = 2:

Θα ρίξουμε το πρώτο ποτήρι από ύψος h_1. Αν σπάσει τότε θα κάνουμε το πολύ, σύμφωνα με τα παραπάνω, h_1-1 δοκιμές με το τελευταίο ποτήρι. Αν δεν σπάσει τότε θα δοκιμάσουμε να το ξαναρίξουμε από ύψος h_2. Έτσι θα κάνουμε άλλες h_2-h_1-1 το πολύ δοκιμές κ.ο.κ. Με άλλα λόγια, μπορούμε με το πρώτο ποτήρι να μειώσουμε το έυρος των δοκιμών. Θα πρέπει να βρούμε όμως το κατάλληλο βήμα για τις δοκιμές του πρώτου ποτηριού. Έστω h αυτό το βήμα. Αν η πρώτη δοκιμή γίνει στο ύψος h και δεν σπάσει το ποτήρι τότε η επόμενη δοκιμή θα γίνει σε ύψος h+h-1 κ.ο.κ (ώστε να έχουμε πάλι το πολύ h δοκιμές). Έτσι θα έχουμε h+(h-1)+(h-2)+…+2+1 = $\frac{h(h+1)}{2}$. Θέλουμε η αριθμητική αυτή σειρά να ισούται με 100, όσο το γνωστό ύψος θραύσης. Οπότε έχουμε την εξίσωση: $\frac{h(h+1)}{2}$ = 100, δηλαδή h = 13.65. Επειδή θέουμε ακέραια τιμή βήματος θα ισχύει h = 14.

Μέγιστο πλήθος δοκιμών: 14.

Για οποιαδήποτε n και k = 2, ακολουθούμε την ίδια λογική και επιλύουμε την εξίσωση $\frac{h(h+1)}{2}$ = n, στρογγυλοποιώντας πάντα ακέραια προς τα πάνω.

(β) Αν k = 1 τότε για κάθε n, όπως στο ερώτημα (α) ρίχνουμε το ποτήρι από κάθε όροφο ξεκινώντας από τον πρώτο ωσότου σπάσει.

Time complexity: O(n).

Αν k = 2 τότε για κάθε n θα ισχύει ότι και στο ερώτημα (α).

Time complexity: O(h). (h^2+h-2n = 0)

Αν n = 0 δεν κάνουμε δοκιμές.

Αν n = 1 θα κάνουμε μια δοκιμή

Time complexity: O(1).

Αναδρομικά θα ισχύει:

Έστω trials(int n, int k) η συνάρτηση (όπου συμπεριλαμβάνει και τις παραπάνω περιπτώσεις). Ο βασικός βρόγχος θα είναι:

```
int min = 10n;
for (I = 1; i<=n; i++)
{
  result = max(trials(i-1, k-1), trials(n-i, k));
  if (result < min) min = result;
}
Return min + 1;
```