Θεμελιώδη Θέματα Επιστήμης Υπολογιστών

Χρήστος Τσουφής
03117176

2η Σειρά Ασκήσεων

Άσκηση 1

α) Για να βρούμε κλειστό τύπο, δουλεύουμε ως εξής:

Ο ψευδοκώδικας του αναδρομικού αλγορίθμου είναι:

```
procedure move_anoi (n from X to Y using Z)
  begin
    if n=1 then
       move top disk from X to Y;
    else
       move_anoi (n-1 from X to Z using Y);
       move top disk from X to Y;
       move_anoi (n-1 from Z to Y using X);
  end
```

Παρατηρούμε ότι όταν η διαδικασία move_anoi καλείται για $n>1$ τότε κάνει κατά σειρά κλήσεις της move_anoi για $n-1$ για μετακίνηση των $n-1$ δίσκων από το X σε Y, μετακίνηση του δίσκου από το X (αρχικό πάσσαλο) στο Z (τελικό πάσσαλο) και κλήση της move_anoi για $n-1$ για μετακίνηση δίσκων από το Y σε X. Αν θεωρήσουμε ότι ο αριθμός των μετακινήσεων για n δίσκους είναι $T(n)$, τότε προκύπτει το αναδρομικό σχήμα:

$$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1)+1$$

Λύνοντας το παραπάνω αναδρομικό σχήμα:

$$T(n) = 2T(n-1)+1 = 2[2T(n-2)+1]+1 + 2T(n-2) + 2+1 = \dots =$$
$$= 2^{(n-1)} T(1) + 2^{(n-2)} + \dots + 2+1$$

Όμως $T(1)=1$ και άρα η παραπάνω σχέση γίνεται:

$$T(n) = 2^{n-1} + 2^{n-2} + \dots + 2+1 = \sum_{k=0}^{n-1} 2^k = 2^n - 1$$

Άρα, τελικά $T(n) = 2^n - 1$.

β) Ο επαναληπτικός αλγόριθμος επίλυσης των πύργων Hanoi εκτελεί πάντα τα εξής δύο βήματα μέχρι να φτάσουν όλοι οι δίσκοι στον στόχο: Πρώτα, μετακινεί το μικρότερο δίσκο προς την ίδια κατεύθυνση κάθε φορά και μετά μετακινείται ο αμέσως μεγαλύτερος δίσκος για τον οποίον υπάρχει επιτρεπτή κίνηση.

Με χρήση μαθηματικής επαγωγής, θδο ο επαναληπτικός αλγόριθμος επίλυσης κάνει ακριβώς τα ίδια βήματα με τον αναδρομικό.

Για $n=1$ δίσκο, και οι δύο αλγόριθμοι κάνουν μία κίνηση (μετακίνηση του μικρότερου δίσκου).

Υποθέτουμε ότι ισχύει για n δίσκους. Θδο ισχύει και για $n+1$ δίσκους.

Ο αναδρομικός αλγόριθμος για $n+1$ θα κάνει κλήση για να μετακινήσει τους n δίσκους από το X στο Z, θα μετακινήσει τον τελευταίο δίσκο από το X στο Y και τέλος, θα μετακινήσει τους n δίσκους από το Z στο Y. Χρησιμοποιώντας την επαγωγική υπόθεση ότι για την μετακίνηση των n δίσκων οι δύο αλγόριθμοι κάνουν τα ίδια βήματα, βλέπουμε ότι έπειτα από τις μετακινήσεις των n πρώτων δίσκων από το X στο Z, η επόμενη κίνηση του επαναληπτικού αλγορίθμου θα είναι η μετακίνηση του $n+1$ δίσκου από το X στο Y ως η μόνη επιτρεπτή κίνηση και έπειτα πάλι με χρήση του επαγωγικού βήματος, ο επαναληπτικός αλγόριθμος κάνει πάλι αυτές μετακινήσεις και ο αναδρομικός για την μετακίνηση των n δίσκων από το Z στο Y.

Άρα, αν ονομάσουμε A τον αριθμό μετακινήσεων που θεωρήσαμε στο επαγωγικό βήμα, και οι δύο αλγόριθμοι χρειάζονται $2A+1$ βήματα για να λύσουν το πρόβλημα των $n+1$ δίσκων και άρα, τελικά, οι δύο αλγόριθμοι κάνουν τον ίδιο αριθμό μετακινήσεων.

δ) Με μαθηματική επαγωγή θέλουμε τώρα να αποδείξουμε πως ο ελάχιστος αριθμός κινήσεων για τους πύργους δίνεται από τον ίδιο τύπο που βρήκαμε για τον αναδρομικό αλγόριθμο, δηλαδή: $M(n) = 2^n - 1$.

Για $n=1$, αρκεί μία κίνηση για την μετακίνηση του δίσκου από τον πάσσαλο X στον Z και επίσης $M(1)=1$.

Θδο ισχύει για $n+1$.

Για να γίνει η μετακίνηση του $n+1$ δίσκου πρέπει πρώτα να έχουν μετακινηθεί οι πρώτοι n δίσκοι και να έχουν σχηματίσει πύργο στον πάσσαλο Y, δηλ. χρειάζονται $M(n)$ κινήσεις. Έπειτα, μετακινούμε τον $n+1$ δίσκο από το X στο Z, με κόστος κίνησης 1. Έπειτα, πρέπει να μετακινήσουμε τους n δίσκους από τον Y στον Z, με κόστος πάλι $M(n)$.

Άρα, συνολικά, $M(n+1) = M(n) + 1 + M(n) = 2^n - 1 + 1 + 2^n - 1 = 2 \cdot 2^n - 1 = 2^{n+1} - 1$

Συνεπώς, ισχύει για $n+1$.

Άρα, ισχύει η επαγωγική υπόθεση και ο ελάχιστος αριθμός μετακινήσεων για τους πύργους Hanoi ταυτίζεται με τον αριθμό κινήσεων που δίνει ο αναδρομικός αλγόριθμος επίλυσης του προβλήματος και ισούται με: $M(n) = 2^n - 1$

δ) Θεωρούμε $C(n,m)$ το ελάχιστο πλήθος κινήσεων με n δίσκους και m στύλους. Μας ενδιαφέρει η περίπτωση $m=4$:

1. Μεταφέρω k δίσκους σε ένα στύλο σε χρόνο $C(k,4)$.
2. Μεταφέρω τους $n-k$ δίσκους χρησιμοποιώντας ένα λιγότερο στύλο που έχουμε καταλάβει στο προηγούμενο βήμα κάνοντας $C(n-k,3)$ κινήσεις.
3. Μεταφέρω τους k δίσκους (που είναι μικρότεροι εφόσον) σε $C(k,4)$ κινήσεις.

Άρα, $C(n,4) = 2C(k,4) + C(n-k,3)$, και από το (α) έχουμε:

$C(n,4) = 2C(k,4) + 2^{n-k} - 1$ και για το χρόνο επίλυσης

$T(n) = 2T(k) + 2^{n-k} - 1$.

Βασιζόμενοι στο Θεώρημα: Στη χειρότερη περίπτωση $T(n) = O(2^n)$, ορίζουμε $trig(n) := \frac{n(n+1)}{2}$ ως συνάρτηση που δίνει το n-οστό τριγωνικό αριθμό.

Για $n := trig(n)$, $k := trig(n-1)$ έχουμε ότι:

$$T(trig(n)) = T(trig(n-1)) + 2^n - 1$$

για $n=1$: $T(2)=1$

$n=2$: $T(3) = 2T(2) + 4 - 1$

$n=3$: $T(6) = 2T(3) + 8 - 1$

$\vdots$

Επιλύοντας την αναδρομή $T(trig(n)) = (n-1)2^n + 1$. Αντιστρέφοντας έχουμε ότι αν η είναι ένας έγκυρος τριγωνικός αριθμός τότε:

$$T(n) = \frac{\sqrt{4n+1}-1}{2}\sqrt{2^{\sqrt{4n+1}-1}} + 1$$

καθώς και ότι για κάθε φυσικό:

$$T\left(\left[\frac{\sqrt{4n+1}-1}{2}\right]\right) \le T(n) \le T\left(\left[\frac{\sqrt{4n+1}-1}{2}\right]\right)$$

Άσκηση 2.

Για την εφαρμογή (τυπικά 30 φορές για κάθε αριθμό n που εξετάζεται) του ισχυρισμού ότι: Αν n πρώτος, τότε για κάθε a τ.ω. $1<a<n-1$, ισχύει $a^{n-1} \bmod n = 1$, θα πρέπει να αντιμετωπισεί το πρόβλημα εύρεσης ενός αποδοτικού τρόπου υπολογισμού της ποσότητας $a^{n-1} \bmod n$, ώστε να μην καθυστερεί τον αλγόριθμο ο άμεσος υπολογισμός της ποσότητας a^{n-1} (δεδομένου ότι για μεγάλα n να φτάσει αστρονομικά μεγάλο μέγεθος).

Ο αλγόριθμος που ακολουθεί χρησιμοποιεί τον επαναλαμβανόμενο τετραγωνισμό Gauss διαφοροποιημένο, όμως, ώστε να χρησιμοποιεί κατάλληλα την αριθμητική modulo για τον σκοπό της άσκησης. Συγκεκριμένα, χρησιμοποιεί την ιδιότητα:

$$(a \cdot b) \bmod m = [(a \bmod m)\cdot(b \bmod m)] \bmod m.$$

ως εξής:

$$b^e \% m = (b^2)^{e/2} \% m = (\underbrace{b^2 \cdot b^2 \cdot \ldots \cdot b^2}_{b^2 \text{ πολ/νται } \frac{e}{2} \text{ φορές}}) \% m$$

$$= [(b^2\% m)\cdot(b^2\% m)\cdot \ldots \cdot (b^2\% m)]\% m = (b^2\% m)^{\frac{e}{2}}\% m$$

Άρα τελικά, $b^e \% m = (b^2 \% m)^{\frac{e}{2}} \% m$

Αλγόριθμος (σε C++):

```
#include <iostream>
using namespace std;
int modpow (int b, int e, int m) {
    if (e==0) return 1;
    if (e%2==0) {
        return (modpow ((b*b)%m, e/2, m));
    }
    else {
        return ( b*modpow ((b*b)%m, e/2, m)) %m;
    }
}

void fermat (int a, int n) {
    if ( modpow (a, n-1, n) == 1) {
        cout << n << "is prime";
    }
    else
        cout << n << "is composite";
}
```

```
Int main (void) {
        int a,n;
        cin >> a >> n;
        fermat (a,n);
        return 0;
}
```

Ενδεικτικά αποτελέσματα:

4	502	5078
7	4098	7919
7 is prime	4098 is Composite	7919 is prime.

β) Αλγόριθμος σε Python:

```
from random import randit, randrange
def check (a, s,d, n):
        x=pow(a,d,n)
        if x==1
                return Treue
        for i in range (s-1):
                if x== n-1:
                        return Tree
                x = pow(x,2,n)
        return x==n-1.

def miller_robin (n,k=40):
    """ Miller-Rabin_Primality_Test"""
    if n==2:
            return True
    if not n & 1:
            return False
    s=0
    d=n-1
```

```
    while d%2 == 0:
        d>>1
        s+=1
    return all(check(randrange(2,n-2), s, d, n), range(k))
#Main
def main():
    for n in xrange(2, 10**5 +1):
        print n
        print 'Miller-Rabin_Test-Result:_' + str(miller_rabin(n,k=40))
main()
```

Άσκηση 3

Για την εύρεση της ελάχιστης τιμής προσβάσιμου κόμβου από οποιονδήποτε κόμβο ακολουθούμε τον αλγόριθμο της C++:

```
int minDepth (int u) {
    if ( ans[u]! =0) return ans[u];
    ans[u]= vari[u];
    for (int i=0; i<n; i++) {
        if ( adj[u][i]= =1) {
            ans[u] = min (ans[u], minDepth[i]);
        }
    }
    return ans[u];
}
```

α) Για άκυκλους κατευθυνόμενους γράφους καλούμε την minDepth ως εξής:

```
for (int i=0; i<n; i++) {
    if ( ans [i]==0)
        ans[i]= minDepth[i];
}
for (int i=0; i<n; i++)
    cout << ansi[i] << " ";
```

Εκ πρώτης όψεως φαίνεται να έχει πολυπλοκότητα $O(n^2)$ όμως ο πίνακας ans λειτουργεί ως πίνακας memoization. Δηλαδή όταν βρίσκουμε την ζητούμενη ελάχιστη τιμή για έναν κόμβο αυτή αποθηκεύεται στον πίνακα. Έτσι, όταν ξανακαλείται αναδρομικά η συνάρτηση για αυτόν τον κόμβο δεν επαναλαμβάνει την διαδικασία, αλλά επιστρέφει την έτοιμη τιμή. Αυτό έχει ως αποτέλεσμα η διαδικασία της minDepth να εκτελείται μόλις μία φορά για κάθε κόμβο. Η πιο πάνω τακτική βασίζεται στο γεγονός ότι η ζητούμενη ελάχιστη τιμή για κάθε κόμβο εξαρτάται μόνο από τους "απογόνους κόμβους του" (για DAG).

β) Για γενικούς κατευθυνόμενους γράφους καλούμε την ίδια συνάρτηση minDepth με τον ακόλουθο τρόπο:

```
for (int i=0; i<n; i++) {
    cout << minDepth[i] << " ";
        for (int j=0; j<n; j++)
            ans[j]=0;
}
```

Αυτή τη φορά δεν ισχύει απόλυτα το γεγονός ότι η ζητούμενη ελάχιστη τιμή για κάθε κόμβο εξαρτάται μόνο από τους 'απογόνους' κόμβους του" διότι υπάρχει το ενδεχόμενο να γίνεται κύκλος. Σε αυτήν την περίπτωση πρέπει να ελεγχθούν όλοι οι επόμενοι κόμβοι μέχρι ο κύκλος να επιστρέψει στον κόμβο. Επειδή η συνάρτηση είναι αναδρομική και κάτι τέτοιο θα ήταν περίπλοκο, μπορούμε να καλέσουμε την minDepth για κάθε κόμβο με άδειο πίνακα ans κάθε φορά.

Ο αλγόριθμος πλέον εκτελεί την n επαναλήψεων minDepth για κάθε κόμβο ανεβάζοντας την πολυπλοκότητα σε $O(n^3)$.

Άσκηση 6

α) Για $k=1$, είναι προφανές πως δεν υπάρχει περίπτωση να μην δοκιμάσουν γραμμικά όλα τα ύψη (σε εκατοστά) από $h=1$ ως $h=n-1$, άρα βέλτιστα, θα χρειαστεί $n-1$ δοκιμές.

Για $k=2$, το ένα ποτήρι θα ορίζει ένα διάστημα εκατοστών, ξεκινώντας και πάλι από $h=1$, μέσα στο οποίο, το άλλο ποτήρι θα αναζητεί (γραμμικά και πάλι) με ακρίβεια εκατοστού το ύψος που αναζητείται. Το πρώτο ποτήρι θα απορρίπτει σταδιακά το διάστημα αναζήτησης του δεύτερου ποτηριού, κάτι το οποίο είναι απαραίτητο αφού εφ' όσον μένει μόνο το δεύτερο, η γραμμική αναζήτηση είναι μονόδρομος.

Η βέλτιστη λύση θα επιτευχθεί όταν οι δοκιμές που χρειάζονται είναι σταθερές σε οποιαδήποτε περίπτωση για το ύψος που σπάνε τα ποτήρια, γιατί για παράδειγμα, αν δοκιμάζουμε με το ένα ποτήρι ανά δεκάδες, στην χειρότερη περίπτωση χρειάζονται $N=9+9=18$ δοκιμές (τα ποτήρια σπάνε για $h=n=100$). Όμως, αν τα ποτήρια σπάνε για $h\leq 10$, χρειάζονται στην χειρότερη περίπτωση (για $h=9$) $N=1+8=9$ δοκιμές, άρα επιδέχεται βελτίωση η μέθοδος. Το ζητούμενο επιτυγχάνεται όταν το ένα ποτήρι δοκιμάζεται σε διαστήματα τα οποία μειώνονται κατά 1 σε κάθε επόμενη δοκιμή, έτσι ώστε οι δοκιμές στην γραμμική αναζήτηση του δεύτερου ποτηριού + οι δοκιμές του πρώτου = σταθερό. (Πρέπει βέβαια και να ξεκινάμε με το ελάχιστο δυνατό διάστημα κατά την πρώτη δοκιμή.)

Για $n=100$ η βέλτιστη λύση είναι λοιπόν για αρχικό ύψος 14 εκατοστά. Ο αριθμός δοκιμών σε κάθε περίπτωση που χρειάζονται είναι $N=14$.

β) Αναπαριστούμε ως trials(n,k) τη λύση του προβλήματος (δοκιμές στην χειρότερη περίπτωση) για μέγιστο ύψος n και k διαθέσιμα ποτήρια. Θα συσχετίσουμε την λύση trials(n,k) με μικρότερα υποπροβλήματα (τα οποία έχουμε υπολογίσει). Αναζητούμε από ποιο εκατοστό να κάνουμε μια δοκιμή έτσι ώστε να έχουμε την ελάχιστη χειρότερη περίπτωση ως προς τις δοκιμές.

Έστω ότι δοκιμάζουμε από ύψος h λοιπόν. Αν το ποτήρι σπάσει, κάνουμε μια δοκιμή + όσες δοκιμές χρειαστούν για να βρούμε τη λύση με ένα λιγότερο ποτήρι, ως ύψος $h-1$, δηλ. $1+\text{trials}(h-1,k-1)$. Αν το ποτήρι δεν σπάσει, κάνουμε πάλι μια δοκιμή + όσες χρειαστούν για να βρούμε τη λύση για ύψος h': $h\leq h'\leq n$, δηλ. $1+\text{trials}(n-h,k)$. Επειδή χρειαζόμαστε την χειρότερη περίπτωση μεταξύ των δύο λοιπόν, είναι:

$trials(n,k) = \min(\max(1 + trials(h-1, k-1), 1 + trials(n-h, k)))$, $1 \le h \le n$

με αρχικές συνθήκες $trials(n,1) = n$, $trials(1,k) = 0$.

δ) Θα χρησιμοποιήσουμε memoization και θα γράψουμε τη συνάρτηση αναδρομικά ως εξής:

```
#include <iostream>
#include <cmath>
#include <climits>
using namespace std;
const unsigned N=100, K=2;
int main()
{
 int dp[N+1][K+1];
   for (int i=1; i<=N; i++)  dp[i][1] = i;
   for (int i=1; i<=k; i++) dp[1][i] = 0;
   for (int n=2; n<=N; n++)
    {
     for (int k=2; k<=K; k++)
      {
       int m= INT_MAX;
       for (int h=1; h<=n; h++)
       {
        int ma = max(dp[h-1][k-1], dp[n-h][k]);
          if (ma<m)  m= ma;
       }
       dp [n][k] = 1+m;
      }
    }
   cout << dp[N][K] << endl;
   return 0;
}
```

Άσκηση 4

α) Με χρήση του αλγορίθμου του Prim, ξεκινώντας από τον κόμβο s που είναι η πρωτεύουσα και επιλέγοντας κάθε φορά κόμβο με την ελάχιστη απόσταση από το μέχρι στιγμής κατασκευασμένο δέντρο. Έτσι, εξερευνώνται όλοι οι κόμβοι καταναλώνοντας τα λιγότερα χιλιόμετρα στο σύνολο. Όταν φτάνει από πόλη σε πόλη ελέγχει αν τα καύσιμα επαρκούν για να πάει στην επόμενη πόλη αλλιώς κάνει ανεφοδιασμό. Στην χειρότερη περίπτωση είναι $O(|V|^2)$. Για να είναι εφικτό πρέπει με τα καύσιμα να μπορεί να πάει σε μια τουλάχιστον πόλη.

β) Με χρήση του αλγορίθμου του Kruskal, ξεκινώντας από την ακμή με το λιγότερο κόστος, ενώνονται οι 2 αυτές πόλεις. Συνεχίζοντας την διαδικασία ακολουθώντας αύξουσα σειρά κόστους χωρίς όμως την δημιουργία κύκλων, μέχρι να ενωθούν όλες οι πόλεις-κόμβοι με τις ελάχιστες ακμές προκύπτει ποιο είναι το ελάχιστο κόστος για να ταξιδέψει κανείς από την πόλη s σε μια άλλη πόλη όπου εκεί μπορεί να ανεφοδιαστεί. Αυτή είναι η ελάχιστη αυτονομία καυσίμου που χρειάζεται. Η πολυπλοκότητα είναι $O(|E|\log|V|)$

Άσκηση 5

Για τον αποτελεσματικό αλγόριθμο για την επίλυση του προβλήματος αυτού θα γίνει χρήση 2 πινάκων T θέσεων, A και B. Στον A θα αποθηκεύονται στην i θέση η ελάχιστη τιμή για i μέρες ανάλογα με τον πίνακα S όπου συμπεριλαμβάνεται ποιες μέρες θα χρειαστούν εισιτήρια. Στον B θα αποθηκευτεί ο συνδυασμός εισιτηρίων που προέκυψε για την τιμή αυτή. Την πρώτη μέρα, γίνεται έλεγχος αν $S[i]=0$ οπότε δεν χρειάζονται εισιτήρια. Αν $S[i]=1$ θα γίνει $A[i] = \min[A-1]+p_k$ και στη συνέχεια θα γίνει έλεγχος για όλα τα c_k αν $\min > A[x-c_k]+p_k$ και τότε θα γίνει αντικατάσταση του min με την καινούρια τιμή. Ο B γεμίζει ανάλογα με $B[i] = B[i-c_k]+c_k$. Ακολουθώντας την ίδια διαδικασία μέχρι το i να ξεπεράσει το T, ο αλγόριθμος ολοκληρώνεται. Η πολυπλοκότητα είναι $O(TK)$