ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΘΕΜΕΛΙΩΔΗ ΘΕΜΑΤΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

(FOCS)

(2020-2021)

2η Σειρά Ασκήσεων

Ονοματεπώνυμο:

- Χρήστος Τσούφης

Αριθμός Μητρώου:

- 03117176

Στοιχεία Επικοινωνίας:

- el17176@mail.ntua.gr

1η Άσκηση (Αναδρομή – Επανάληψη – Επαγωγή)

a) *Εκφράστε τον αριθμό κινήσεων δίσκων που κάνει ο αναδρομικός αλγόριθμος για τους πύργους του Hanoi, σαν συνάρτηση του αριθμού των δίσκων n.*

Η εύρεση κλειστού τύπου προκύπτει με τον ψευδοκώδικα του αναδρομικού αλγόριθμου:

```
procedure move_hanoi(n from X to Y using Z)

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

Αυτό που παρατηρείται είναι ότι όταν η διαδικασία move_hanoi καλείται για n > 1 τότε κάνει κατά σειρά κλήση της move_hanoi για n – 1 για μετακίνηση των n – 1 δίσκων από το X στο Y, μετακίνηση του δίσκου από το X (αρχικό πύργο) στο Z (τελικό πύργο) και κλήση της move_hanoi για n – 1 για μετακίνηση n – 1 δίσκων από το Y στο X. Θεωρώντας ότι ο αριθμός των μετακινήσεων για n δίσκους είναι T(n), τότε προκύπτει η αναδρομική σχέση:
$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1.$
Λύνοντας την παραπάνω σχέση, προκύπτει:
$T(n) = 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1 = 2T(n-2) + 2 + 1 = \ldots = 2^{n-1}T(1) + 2^{n-2} + 2 + 1$
Όμως T(1) = 1. Άρα, η σχέση γίνεται: $T(n) = 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = \sum_{k=0}^{n-1} 2^k = 2^n - 1$
Συνεπώς, $T(n) = 2^n - 1$

b) *Δείξτε ότι ο αριθμός κινήσεων του αναδρομικού ισούται με τον αριθμό μετακινήσεων του επαναληπτικού αλγορίθμου.*

Ο επαναληπτικός αλγόριθμος επίλυσης των πύργων Hanoi εκτελεί πάντας τα εξής δύο βήματα μέχρι να φτάσουν όλοι οι άλλοι δίσκοι στον στόχο. Πρώτα, μετακινεί το μικρότερο δίσκο προς την ίδια κατεύθυνση κάθε φορά και ύστερα μετακινείται ο αμέσως μεγαλύτερος δίσκος για τον οποίον υπάρχει επιτρεπτή κίνηση.
Χρησιμοποιώντας την Μαθηματική Επαγωγή, θα δειχθεί ότι ο επαναληπτικός αλγόριθμος επίλυσης κάνει ακριβώς τα ίδια βήματα με τον αναδρομικό.

- Για n=1 δίσκο, και οι δύο αλγόριθμοι μετακινούν τον μικρότερο δίσκο.
- Υποθέτοντας ότι ισχύει για n δίσκους, θα δειχθεί ότι ισχύει και για n+1 δίσκους.
- Ο αναδρομικός αλγόριθμος για n+1 δίσκους 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, φαίνεται 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 όσες μετακινήσεις και ο αναδρομικός για την μετακίνηση των n δίσκων από το Z στο Y.

Επομένως, αν ο αριθμός των μετακινήσεων που θεωρήθηκαν στο επαγωγικό βήμα είναι έστω Α, τότε και οι δύο αλγόριθμοι χρειάζονται 2Α+1 βήματα για να λύσουν το πρόβλημα των n+1 δίσκων και άρα, τελικά, οι δύο αλγόριθμοι κάνουν τον ίδιο αριθμό μετακινήσεων.

c) *Δείξτε ότι ο αριθμός των κινήσεων των παραπάνω αλγορίθμων είναι ο ελάχιστος μεταξύ όλων των δυνατών αλγορίθμων για το πρόβλημα αυτό*.

Με Μαθηματική Επαγωγή αποδεικνύεται πως ο ελάχιστος αριθμός μετακινήσεων για τους πύργους δίνεται από τον ίδιο τύπο με τον αναδρομικό αλγόριθμο, δηλαδή: $M(n) = 2^n - 1$

- Για n=1, αρκεί μια κίνηση για την μετακίνηση του δίσκου από το X στο Z και είναι M(1) = 1.
- Υποθέτοντας ότι ισχύει για n, θα δειχθεί ότι ισχύει και 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, δηλαδή χρειάζονται M(n) κινήσεις. Έπειτα, μετακινείται ο n+1 δίσκος από το X στο Z, με κόστος κίνησης 1. Τέλος, πρέπει να μετακινηθούν οι n δίσκοι από το Y στο Z, με κόστος πάλι M(n). Συνολικά θα είναι:
  $M(n+1) = M(n) + 1 + M(n) = 2^n - 1 + 1 + 2^n - 1 = 2*2^n - 1 = 2^n+1 - 1$
  Συνεπώς ισχύει και 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: $M(n) = 2^n - 1$.

d) *Θεωρήστε το πρόβλημα των πύργων του Hanoi με 4 αντί για 3 πασσάλους. Σχεδιάστε αλγόριθμο μετακίνησης n δίσκων από τον πάσσαλο 1 στον πάσσαλο 4 ώστε το πλήθος των βημάτων να είναι σημαντικά μικρότερο από το πλήθος των βημάτων που απαιτούνται όταν υπάρχουν μόνο 3 πάσσαλοι. Εκφράστε τον αριθμό των απαιτούμενων βημάτων σαν συνάρτηση του n*.

Θεωρώντας C(n, m) ως το ελάχιστο πλήθος κινήσεων με n δίσκους και m πασσάλους, προκύπτει για την περίπτωση m=4 ότι:

i. Μετάφερε k δίσκους σε ένα στύλο σε χρόνο C(k, 4).
ii. Μετάφερε τους n-k δίσκους χρησιμοποιώντας ένα λιγότερο στύλο από το προηγούμενο βήμα κάνοντας C(n-k, 3) κλήσεις.
iii. Μετάφερε τους k δίσκους (που είναι μικρότερου μεγέθους) σε C(k, 4) κινήσεις.

Άρα, $C(n, 4) = 2C(k, 4) + C(n-k, 3)$ και από το (a) προκύπτει: $C(n, 4) = 2C(k, 4) + 2^{n-k} - 1$.
Επιπλέον, για τον χρόνο εκτέλεσης ισχύει $T(n) = 2T(k) + 2^{n-k} - 1$.

Με βάση το Θεώρημα:
Στην χειρότερη περίπτωση T(n) = O(2n), ορίζεται η συνάρτηση $\text{trig}(n) := \frac{n(n+1)}{2}$ ως συνάρτηση που δίνει το n-οστό τριγωνικό αριθμό.
Για n := trig(n), k := trig(n-1), προκύπτει ότι:
$T(\text{trig}(n)) = T(\text{trig}(n-1)) + 2^n - 1$
Όπου, για n=1: T(2)=1
για n=2: T(3) = 2T(2) + 4 – 1
για n=3: T(6) = 2(3) + 8 – 1
…

Επιλύοντας την αναδρομή προκύπτει: $T(\text{trig}(n)) = (n-1)2^n + 1$.
Αντιστρέφοντας τη σχέση, προκύπτει ότι αν n είναι ένας έγκυρος τριγωνικός αριθμός, τότε:
$T(n) = \frac{\sqrt{4n+1}-1}{2}\sqrt{2^{\sqrt{4n+1}-1}}+1$ και $T(\left\lfloor\frac{\sqrt{4n+1}-1}{2}\right\rfloor) \leq T(n) \leq T(\lfloor\frac{\sqrt{4n+1}-1}{2}\rfloor)$

2η Άσκηση (Επαναλαμβανόμενος Τετραγωνισμός - Κρυπτογραφία)

a) *Γράψτε πρόγραμμα σε γλώσσα της επιλογής σας (θα πρέπει να υποστηρίζει πράξεις με αριθμούς 100δων ψηφίων) που να ελέγχει αν ένας αριθμός είναι πρώτος με τον έλεγχο (test) του Fermat…*

Για την εφαρμογή (τυπικά 30 φορές για τον κάθε αριθμό n που εξετάζεται) του ισχυρισμού ότι: Αν n πρώτος, τότε για κάθε α τέτοιο ώστε $1<\alpha<n-1$, ισχύει $\alpha^{n-1} \bmod n = 1$, θα πρέπει να αντιμετωπιστεί το πρόβλημα εύρεσης ενός αποδοτικού τρόπου υπολογισμού της ποσότητας $\alpha^{n-1} \bmod n$, ώστε να μην καθυστερεί τον αλγόριθμο ο άμεσος υπολογισμός της ποσότητας α^{n-1} (λαμβάνοντας υπόψιν ότι για μεγάλα n μπορεί να φτάσει αστρονομικά μεγάλο μέγεθος).
Ο αλγόριθμος που ακολουθεί, χρησιμοποιεί τον επαναλαμβανόμενο τετραγωνισμό Gauss ο οποίος έχει διαμορφωθεί έτσι ώστε να χρησιμοποιεί κατάλληλα την αριθμητική modulo για τον σκοπό της άσκησης. Πιο συγκεκριμένα, χρησιμοποιεί την ιδιότητα:
$(a*b) \bmod m = [(a \bmod m)*(b \bmod m)] \bmod m$ ως εξής:
$b^e \% m = (b^2)^{e/2} \% m = (b^2 * b^2 * \dots * b^2) \% m = [(b^2 \% m) * (b^2 \% m) * \dots * (b^2 \% m)] \% m \Longrightarrow$
$b^e \% m = (b^2 \% m)^{e/2} \% m$
Ο πηγαίος κώδικας του αλγορίθμου 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 I/O: 4 7 ⟹ 7 is prime

Και σε διαφορετική υλοποίηση σε **Python**:

```python
from random import randrange

def modexp( a, b, p ):
  if b == 0:
    return 1
  y = 1
  x = a
  n = b
  while n > 1:
    bit = n  % 2
    if bit == 0:
      x = ( x * x ) % p
      n = n / 2
    else:
      y = ( x * y ) % p
      x = ( x * x ) % p
      n = ( n - 1 ) / 2
  return ( x * y ) % p

def fermat( p ):
  if( p == 1 ):
    return False
  if( p == 2 ):
    return True
  a = randrange( 2, p - 2 )
  if( modexp( a, p - 1, p ) == 1 ):
    return True
  return False

n = input();
prime = True
for i in range(100):
  if( fermat( n ) == 0 ):
    prime = False
    break

print (prime)
```

b) *Μελετήστε και υλοποιήστε τον έλεγχο Miller-Rabin που αποτελεί βελτίωση του ελέγχου του Fermat και δίνει σωστή απάντηση με πιθανότητα τουλάχιστον 1/2 για κάθε φυσικό αριθμό (οπότε με 30 επαναλήψεις έχουμε αμελητέα πιθανότητα λάθους για κάθε αριθμό εισόδου). Δοκιμάστε τον με διάφορους αριθμούς Carmichael. Τι παρατηρείτε;*

Ο πηγαίος κώδικας σε **Python**:

```python
from random import randint, randrange

# Miller - Rabin
def check(a, s, d, n):
    x = pow(a, d, n)
    if x==1:
        return True
    for i in range(s-1):
        if x == n-1:
            return True
        x = pow(x, 2, n)
    return x == n-1

def miller_rabin(n, k=40):
    if n==2:
        return True
    if not n&1:
        return False
    s = 0
    d = n-1

    while d%2 == 0:
        d >>=1
        s = s+1
    return all(check(randrange(2, n-2), s, d, n), range(k))

# Main
def main():
    for n in range(2, 10**5 + 1):
        print (n)
        print ("Miller-Rabin Test Result: ", str(miller_rabin(n, k=40)))

main()
```

c) *Γράψτε πρόγραμμα που να βρίσκει όλους τους πρώτους αριθμούς Mersenne, δηλαδή της μορφής $n = 2x − 1$ με $100 < x < 1000$ (σημειώστε ότι αν το x δεν είναι πρώτος, ούτε το $2x − 1$ είναι πρώτος – μπορείτε να το αποδείξετε;). Αντιπαραβάλετε με όσα αναφέρονται στην ιστοσελίδα https://www.mersenne.org/primes/.*

Για την επίλυση, χρησιμοποιήθηκε η ιδιότητα ότι αν ένας αριθμός x δεν είναι πρώτος, τότε ούτε ο $2^x – 1$ είναι πρώτος. Αυτό ισχύει, διότι, $2^{ab} – 1 = (2^a – 1)(2^{a(b-1)} + 2^{a(b-2)} + … + 2^a + 1)$. Ο κώδικας σε **Python** (με τροποποιημένη την συνάρτηση Miller-Rabin):

```
from random import randint

def millerrabintest(n,d,s,a):
    x=pow(a,d,n)
    if(x==1 or x==n-1):
        return True
    for i in range(0,s-1):
        x=(x*x)%n
        if(x==n-1):
            return True
    return False

def millerrabin(n, k=30):
    if(n<2 or n==4):
        return False
    if(n==2 or n==3):
        return True

    d=n-1
    s=0
    while(d%2==0):
        d=d/2
        s=s+1

    for i in range(0,k):
        a=randint(2,n-1)
        if(millerrabintest(n,d,s,a)==False):
            return False
    return True


lower_bound = 100
upper_bound = 3000

primes = [True for i in range(upper_bound + 1)]
primes[0]=False
primes[1]=False
p=2
while(p*p<upper_bound+1):
    if(primes[p]):
```

```
        for i in range(p*p, upper_bound+1,p):
            primes[i]=False
    p=p+1

for p in range (lower_bound, upper_bound+1):
    if(primes[p]):
        m=pow(2,p)-1
        if(millerrabin(m)):
            print (p)
```

Το πρόγραμμα βρίσκει πρώτα τους πρώτους αριθμούς μέχρι το 3.000 με τη μέθοδο του Κόσκινου του Ερατοσθένη και στην συνέχεια για κάθε πρώτο p στο[100, 3000] καλεί την Miller-Rabin για να ελέγξει αν ο Mersenne m = 2^p – 1 είναι πρώτος και εκτυπώνει κάθε p για το οποίο βρίσκει από την σχέση ότι είναι πρώτος.

3η Άσκηση (Αριθμοί Fibonacci)

a) *Υλοποιήστε και συγκρίνετε τους εξής αλγορίθμους για υπολογισμό του n-οστού αριθμού Fibonacci: αναδρομικό με memoization, επαναληπτικό, και με πίνακα. Υλοποιήστε τους αλγορίθμους σε γλώσσα που να υποστηρίζει πολύ μεγάλους ακεραίους (100δων ψηφίων), π.χ. σε Python. Χρησιμοποιήστε τον πολλαπλασιασμό ακεραίων που παρέχει η γλώσσα. Τι συμπεραίνετε;*

Υπολογισμός με memoization σε recursive algorithm

Το παρακάτω πρόγραμμα αφορά τον αναδρομικό αλγόριθμο για την εύρεση του n-οστού Fibonacci. Μόνο μια παράμετρος αλλάζει και για αυτό το λόγο, αυτή η μέθοδος είναι γνωστή ως 1-D memoization. Ο πηγαίος κώδικας σε **C++**:

```
// C++ program to find the Nth term of Fibonacci series
#include <bits/stdc++.h>
using namespace std;

// Fibonacci Series using Recursion
int fib(int n)
{
    // Base case
    if (n <= 1)
        return n;
    // recursive calls
    return fib(n - 1) + fib(n - 2);
}

// Driver Code
int main()
{
    int n = 6;
    printf("%d", fib(n));
    return 0;
} // output is 8
```

Παρατηρώντας τον αλγόριθμο προκύπτει ότι αυτή η υλοποίηση κάνει αρκετά συχνούς και επαναλαμβανόμενους υπολογισμούς το οποίο καταναλώνει πολύ χρόνο (μπορεί να φτάσει σε πολυπλοκότητα $O(2^n)$). Αναλυτικότερα, στο link:
https://www.geeksforgeeks.org/time-complexity-recursive-fibonacci-program/

Μια καλύτερη επίλυση είναι η Top – Down προσέγγιση. Μερικές από τις τροποποιήσεις που θα φανούν παρακάτω μειώνουν την πολυπλοκότητα του προγράμματος και δίνουν το επιθυμητό αποτέλεσμα. Συγκεκριμένα, αν το fib(x) δεν έχει υπολογιστεί προηγουμένως, τότε αυτό αποθηκεύεται σε ένα πίνακα με index x και return term[x]. Με το memoization του αποτελέσματος, μειώνεται ο αριθμός των αναδρομικών κλήσεων κι έτσι, ανατρέχει κάθε φορά σε εκείνο τον πίνακα και αποφεύγει περιττές επαναλήψεις. Ο πηγαίος κώδικας σε **C++**:

```cpp
// CPP program to find the Nth term of Fibonacci series
#include <bits/stdc++.h>
using namespace std;
int term[1000];
// Fibonacci Series using memoized Recursion
int fib(int n)
{
    // base case
    if (n <= 1)
        return n;

    // if fib(n) has already been computed we do not do further recursive
    // calls and hence reduce the number of repeated work
    if (term[n] != 0)
        return term[n];

    else {

        // store the computed value of fib(n) in an array term at index n to
        // so that it does not needs to be precomputed again
        term[n] = fib(n - 1) + fib(n - 2);
        return term[n];
    }
}

// Driver Code
int main()
{
    int n = 6;
    printf("%d", fib(n));
    return 0;
} // output is 8
```

Ο αλγόριθμος θα έχει πολυπλοκότητα ίση με τον αριθμό των αριθμών που παράχθηκαν, δηλαδή O(n). Αναλυτικότερα, στο link:
https://www.geeksforgeeks.org/memoization-1d-2d-and-3d/

Υπολογισμός με iterative algorithm

Με την χρήση Dynamic Programming αποφεύγονται επίσης οι περιττές επαναλήψεις. Παρακάτω φαίνεται μια Optimized Method καθώς αποθηκεύονται οι προηγούμενοι δύο αριθμοί διότι μόνο εκείνοι χρειάζονται για τον υπολογισμό του n-οστού. Ο αλγόριθμος σε **C++**:

```
// Fibonacci Series using Space Optimized Method
#include<bits/stdc++.h>
using namespace std;

int fib(int n)
{
    int a = 0, b = 1, c, i;
    if( n == 0)
        return a;
    for(i = 2; i <= n; i++)
    {
    c = a + b;
    a = b;
    b = c;
    }
    return b;
}

// Driver code
int main()
{
    int n = 9;

    cout << fib(n);
    return 0;
}
```

Η χρονική πολυπλοκότητα και σε αυτήν την περίπτωση είναι O(n).

Υπολογισμός με memoization σε algorithm με array

Και αυτή είναι μια Top – Down προσέγγιση ώστε να αποφευχθούν οι περιττοί υπολογισμοί. Τα στοιχεία αποθηκεύονται σε πίνακα. Ο αλγόριθμος σε **C++**:

```cpp
#include <bits/stdc++.h>
using namespace std;
int dp[10];
int fib(int n)
{
    if (n <= 1)
        return n;
    // temporary variables to store values of fib(n-1) & fib(n-2)
    int first, second;

    if (dp[n - 1] != -1)
        first = dp[n - 1];
    else
        first = fib(n - 1);

    if (dp[n - 2] != -1)
        second = dp[n - 2];
    else
        second = fib(n - 2);

    // memoization
    return dp[n] = first + second;
}

// Driver Code
int main()
{
    int n = 9;

    memset(dp, -1, sizeof(dp));
    cout << fib(n);
    getchar();
    return 0;
}
```

Και εδώ, η πολυπλοκότητα είναι O(n). Αναλυτικότερα, στο link:
https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

Επιπλέον πληροφορίες υπάρχουν και στα link:

- https://stackoverflow.com/questions/13440020/big-o-for-various-fibonacci-implementations/49722403
- https://medium.com/@edwinyung/using-fibonacci-to-exemplify-recursion-big-o-and-memoization-9b1b47316c5e

Υπολογισμός με algorithm που χρησιμοποιεί δυνάμεις του πίνακα $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Ο αλγόριθμος σε **C++**:

```
#include<bits/stdc++.h>
using namespace std;

// Helper function that multiplies 2 matrices F and M of size 2*2, and
// puts the multiplication result back to F[][]
void multiply(int F[2][2], int M[2][2]);

// Helper function that calculates F[][] raise to the power n and puts the
// result in F[][] . Note that this function is designed
// only for fib() and won't work as general power function
void power(int F[2][2], int n);

int fib(int n)
{
    int F[2][2] = { { 1, 1 }, { 1, 0 } };
    if (n == 0)
        return 0;

    power(F, n - 1);

    return F[0][0];
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0] * M[0][0] +
            F[0][1] * M[1][0];
    int y = F[0][0] * M[0][1] +
            F[0][1] * M[1][1];
    int z = F[1][0] * M[0][0] +
            F[1][1] * M[1][0];
    int w = F[1][0] * M[0][1] +
            F[1][1] * M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = { { 1, 1 }, { 1, 0 } };
```

```
    // n - 1 times multiply the
    // matrix to {{1,0},{0,1}}
    for(i = 2; i <= n; i++)
        multiply(F, M);
}

// Driver code
int main()
{
    int n = 9;

    cout << " " << fib(n);

    return 0;
}
```

Σε αυτή την περίπτωση η πολυπλοκότητα μπορεί να γίνει O(logn). Αναλυτικότερα, στο link: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

b) *Δοκιμάστε να λύσετε το παραπάνω πρόβλημα με ύψωση σε δύναμη, χρησιμοποιώντας τη σχέση του Fn με το ϕ (χρυσή τομή). Τι παρατηρείτε;*

Η χρυσή τομή $\varphi = \frac{1+\sqrt{5}}{2} = 1.61803398\ldots$
Η προσέγγιση με την χρυσή τομή μπορεί και να δώσει λανθασμένο αποτέλεσμα. Το σωστό αποτέλεσμα θα προκύψει αν στρογγυλοποιείται το αποτέλεσμα σε κάθε σημείο. Ο αλγόριθμος σε **C++**:

```
// C++ Program to find n'th fibonacci Number
#include<iostream>
#include<cmath>

int fib(int n) {
double phi = (1 + sqrt(5)) / 2;
return round(pow(phi, n) / sqrt(5));
}

// Driver Code
int main ()
{
int n = 9;
std::cout << fib(n) << std::endl;
return 0;
}
```

Σε αυτή την περίπτωση η πολυπλοκότητα γίνεται O(1). Αναλυτικότερα στα link:

- https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
- https://www.geeksforgeeks.org/find-nth-fibonacci-number-using-golden-ratio/?ref=rp

c) *Υπολογίστε και συγκρίνετε την πολυπλοκότητα ψηφιοπράξεων (bit complexity) του επαναληπτικού αλγορίθμου για υπολογισμό αριθμών Fibonacci και του αλγορίθμου που χρησιμοποιεί πίνακα. Για τον αλγόριθμο με πίνακα θεωρήστε (α) απλό πολλαπλασιασμό ακεραίων και (β) πολλαπλασιασμό Gauss-Karatsuba. Τι παρατηρείτε σε σχέση και με το ερώτημα (α);*

Όσον αφορά τον επαναληπτικό αλγόριθμο, στην ουσία επαναλαμβάνει μια πράξη πρόσθεσης δύο αποθηκευμένων μεταβλητών και μετά τις μεταθέτει κατάλληλα ώστε να πραγματοποιήσει την επόμενη πρόσθεση. Για το bit complexity, δεν έχουν σημασία οι μεταθέσεις αλλά οι πράξεις, οπότε θεωρώντας ότι η πρόσθεση δύο αριθμών γίνεται σε O(n), τότε ο αλγόριθμος θα εκτελέσει n – 1 προσθέσεις άρα το bit complexity είναι $O(n(n-1)||a||) = O(||a||n^2)$.

Από την άλλη, όσον αφορά την μέθοδο των πινάκων, χρειάζονται για τον υπολογισμό του Fn, περίπου logn πολλαπλασιασμοί πινάκων. Όμως, κάθε πολλαπλασιασμός πινάκων 2x2 χρειάζεται 8 πολλαπλασιασμούς και άλλες 4 προσθέσεις, οπότε συνολικά θα έχει bit complexity ίση με $O(\log n(8||a||^2n^2 + 4n||a||)) = O(n^2\log n||a||^2)$.

Χρησιμοποιώντας τον πολλαπλασιασμό Gauss-Karatsuba, το bit complexity ελαττώνεται σε $O(n^{\log 3}\log n||a||^{\log 3})$. Συνεπώς, ο αλγόριθμος με την μέθοδο των πινάκων είναι πιο γρήγορος, όπως φανερώνει και το ερώτημα (a).

d) *Υλοποιήστε συνάρτηση που να δέχεται σαν είσοδο δύο θετικούς ακεραίους n, k και να υπολογίζει τα k λιγότερο σημαντικά ψηφία του n-οστού αριθμού Fibonacci.*

Ο κώδικας σε **C++**:

```cpp
// Optimized Program to find last digit of nth Fibonacci number
#include<bits/stdc++.h>
using namespace std;

typedef long long int ll;

// Finds nth fibonacci number
ll fib(ll f[], ll n)
{
    // 0th and 1st number of the series are 0 and 1
    f[0] = 0;
    f[1] = 1;

    // Add the previous 2 numbers in the series and store last digit of result
    for (ll i = 2; i <= n; i++)
        f[i] = (f[i - 1] + f[i - 2]) % 10;

    return f[n];
}

// Returns last digit of n'th Fibonacci Number
int findLastDigit(int n)
{
```

```
    ll f[60] = {0};

    // Precomputing units digit of first 60 Fibonacci numbers
    fib(f, 60);

    return f[n % 60];
}

// Driver code
int main ()
{
    ll n = 1;
    cout << findLastDigit(n) << endl;
    n = 61;
    cout << findLastDigit(n) << endl;
    n = 7;
    cout << findLastDigit(n) << endl;
    n = 67;
    cout << findLastDigit(n) << endl;
    return 0;
}
```

Η πολυπλοκότητα εδώ είναι O(1). Αναλυτικότερα, στα link:

- https://stackoverflow.com/questions/54480322/last-digit-of-a-large-fibonacci-number-fast-algorithm
- https://www.geeksforgeeks.org/program-find-last-digit-nth-fibonnaci-number/
- https://www.tutorialspoint.com/program-to-find-last-two-digits-of-nth-fibonacci-number-in-cplusplus

e) *Αναζητήστε και εξετάστε τη μέθοδο Fast Doubling σε σχέση με τα παραπάνω ερωτήματα. Συγκρίνετέ την με τη μέθοδο του πίνακα.*

Η μέθοδος του Fast Doubling φαίνεται στον παρακάτω κώδικα σε **C++**:

```
// C++ program to find the Nth Fibonacci
// number using Fast Doubling Method
#include <bits/stdc++.h>
using namespace std;

int a, b, c, d;
#define MOD 1000000007

// Function calculate the N-th fibanacci number using fast doubling method
void FastDoubling(int n, int res[])
{
    // Base Condition
    if (n == 0) {
        res[0] = 0;
```

```
        res[1] = 1;
        return;
    }
    FastDoubling((n / 2), res);

    // Here a = F(n)
    a = res[0];

    // Here b = F(n+1)
    b = res[1];

    c = 2 * b - a;

    if (c < 0)
        c += MOD;

    // As F(2n) = F(n)[2F(n+1) - F(n)]
    // Here c = F(2n)
    c = (a * c) % MOD;

    // As F(2n + 1) = F(n)^2 + F(n+1)^2
    // Here d = F(2n + 1)
    d = (a * a + b * b) % MOD;

    // Check if N is odd or even
    if (n % 2 == 0) {
        res[0] = c;
        res[1] = d;
    }
    else {
        res[0] = d;
        res[1] = c + d;
    }
}
// Driver code
int main()
{
    int N = 6;
    int res[2] = { 0 };

    FastDoubling(N, res);

    cout << res[0] << "\n";
    return 0;
}
```

Έχει πολυπλοκότητα O(logn) για την εύρεση του n-οστού αριθμού. Αναλυτικότερα στο link: https://www.geeksforgeeks.org/fast-doubling-method-to-find-the-nth-fibonacci-number/?ref=rp

4η Άσκηση (Λιγότερα Διόδια)

Θεωρήστε το εξής πρόβλημα σε οδικά δίκτυα: κάθε κόμβος έχει διόδια (μια θετική ακέραια τιμή) και θέλουμε να βρεθούν οι διαδρομές με το ελάχιστο κόστος έναν αρχικό κόμβο s προς κάθε άλλο κόμβο. Θεωρήστε ότι στον αρχικό κόμβο δεν πληρώνουμε διόδια (ενώ στον τελικό πληρώνουμε, όπως και σε κάθε ενδιάμεσο) και ότι το δίκτυο παριστάνεται σαν κατευθυνόμενος γράφος. Περιγράψτε όσο το δυνατόν πιο αποδοτικούς αλγόριθμους για το πρόβλημα αυτό στις εξής περιπτώσεις:

Για την εκτέλεση της ελάχιστης τιμής προσβάσιμου κόμβου από οποιοδήποτε κόμβο ισχύει ο αλγόριθμος σε **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];
}
```

a) *Το δίκτυο δεν έχει κύκλους (κατευθυνόμενους).*

1ος τρόπος επίλυσης:
Ο πηγαίος κώδικας:

```
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 υπολογίζεται η ζητούμενη ελάχιστη τιμή για έναν κόμβο αυτή αποθηκεύεται στον πίνακα. Έτσι, όταν ξανακαλείται αναδρομικά η συνάρτηση για αυτόν τον κόμβο δεν επαναλαμβάνεται η διαδικασία και απλά επιστρέφει την έτοιμη τιμή. Αυτό έχει ως αποτέλεσμα η διαδικασία minDepth να εκτελείται μόλις μια φορά για κάθε κόμβο. Η πιο πάνω τακτική βασίζεται στο γεγονός ότι η ζητούμενη ελάχιστη τιμή για κάθε κόμβο εξαρτάται μόνο από τους «απόγονους κόμβους του» (για DAG).

2ος τρόπος επίλυσης:

Μια τροποποιημένη επίλυση είναι η ακόλουθη περιγραφή:

- Στην αρχή, υπολογίζεται η τοπολογική διάταξη του γράφου και αποθηκεύονται οι ταξινομημένες κορυφές σε έναν πίνακα A. Έπειτα, τίθεται ως $d(i)$ το κόστος της καλύτερης διαδρομής που έχει βρεθεί μέχρι αυτή τη στιγμή από τον αρχικό κόμβο s προς τον κόμβο i και οι τιμές είναι $d(s)=0$ και $d(i)=\infty\ \forall i \neq s$. Ύστερα, τίθεται ως $prev(i)$ ο προηγούμενος κόμβος του i στην καλύτερη διαδρομή που έχει βρεθεί μέχρι αυτή τη στιγμή από τον s στον i και ισχύει ότι $p(i)=null\ \forall i$.
- Μετά, διατρέχονται οι κόμβοι σύμφωνα με την ταξινομημένη σειρά του πίνακα A, ξεκινώντας από τον s. Στην περίπτωση που ο s δεν είναι ο πρώτος κόμβος στη διάταξη, τότε είναι γνωστό εξαρχής ότι όλοι οι προηγούμενοι του είναι μη προσβάσιμοι ξεκινώντας από αυτόν.
- Για κάθε κόμβο u, ελέγχονται όλοι οι κόμβοι στους οποίους οδηγεί ακμή που ξεκινάει από τον u. Για κάθε τέτοιο κόμβο v, αν ισχύει $d(u)+c(u,v)<d(v)$ τότε τίθεται $d(v)=d(u)+c(u,v)$ (όπου $c(u,v)$ το βάρος της ακμής (u,v)). Στην περίπτωση αυτή τίθεται επίσης $prev(v)=u$. Δηλαδή, αν το κόστος της διαδρομής προς το v μέσω του u είναι μικρότερο από το κόστος της (ως τώρα) διαδρομής προς το v, τότε τίθεται αυτή ως καλύτερη διαδρομή και το u ως προηγούμενο του v. Στην αντίθετη περίπτωση διατηρούνται οι υπάρχουσες τιμές των $d(v)$, $prev(v)$.

Μόλις τελειώσει ο αλγόριθμος, η τιμή $d(i)$ δηλώνει το ελάχιστο κόστος διαδρομής από τον αρχικό κόμβο s προς τον κόμβο i, ενώ το $prev(i)$ δηλώνει τον προηγούμενο κόμβο του i στη διαδρομή αυτή και χρησιμοποιείται για τον προσδιορισμό της διαδρομής. Οι κόμβοι με $d(i)=\infty$ ή $prev(i)=null$ δεν είναι προσβάσιμοι με κανέναν τρόπο από τον s. Σε αυτούς συμπεριλαμβάνεται κάθε κόμβος που προηγείται του s στην τοπολογική διάταξη (αλλά ανάλογα με τον τρόπο που έγινε η ταξινόμηση δεν είναι υποχρεωτικά μόνο αυτοί). Η πολυπλοκότητα του αλγορίθμου είναι $O(n+m)$, όπου n ο αριθμός κόμβων και m ο αριθμός ακμών του γράφου. Η τοπολογική ταξινόμηση, χρησιμοποιώντας τροποποιημένη DFS, απαιτεί χρόνο $O(n+m)$. Ύστερα για κάθε κορυφή ελέγχει τις γειτονικές κορυφές ($O(m)$).

b) *Το δίκτυο μπορεί να έχει κύκλους.*

1ος τρόπος επίλυσης:
Για γενικούς κατευθυνόμενους γράφους καλείται η ίδια συνάρτηση 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 την 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 εκείνη 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, γίνεται να καλείται η 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 $O(n^2)$.

2ος τρόπος επίλυσης:

Μια τροποποιημένη επίλυση με Dijkstra, είναι η ακόλουθη:

- Αρχικά, τίθενται όλοι οι κόμβοι ως unvisited. Έπειτα, τίθεται ως d(i) το κόστος της καλύτερης διαδρομής που έχει βρεθεί μέχρι αυτή τη στιγμή από τον αρχικό κόμβο s προς τον κόμβο i και ισχύει ότι d(s)=0 και d(i)=∞ $\forall i \neq s$. Ύστερα, τίθεται ως prev(i) ο προηγούμενος κόμβος από τον i στην καλύτερη διαδρομή που έχει βρεθεί ως εκείνη τη στιγμή από τον s στον i και ισχύει ότι prev(i)=$null$ $\forall i$. Μετά, τίθεται ως τρέχων κόμβος ο αρχικός.
- Για τον τρέχων κόμβο u, ελέγχονται όλοι οι unvisited γειτονικοί κόμβοι στους οποίους οδηγεί ακμή από τον u: για τον γείτονα v, αν d(u)+$c(u,v)$<d(v) τότε τίθεται d(v)=d(u)+$c(u,v)$ και prev(v)=u. Στην αντίθετη περίπτωση διατηρούνται οι υπάρχουσες τιμές των d(v), prev(v).
- Όταν ελεγχθούν όλοι οι προσβάσιμοι γείτονες του τρέχοντος κόμβου, τότε τίθεται ως visited.
- Αν δεν υπάρχουν unvisited κόμβοι ή αν για κάθε unvisited κόμβο i ισχύει d(i)=∞ τότε ο αλγόριθμος τελείωσε. Για κάθε κόμβο i, η d(i) δηλώνει το ελάχιστο κόστος διαδρομής προς αυτόν από τον αρχικό κόμβο s, ενώ η prev(i) δηλώνει τον προηγούμενο κόμβο στη διαδρομή αυτή και χρησιμοποιείται για τον προσδιορισμό της διαδρομής. Οι κόμβοι με d(i)=∞ ή prev(i)=$null$ δεν είναι προσβάσιμοι με κανέναν τρόπο από τον s.
- Διαφορετικά, από τους unvisited κόμβους τίθεται ως τρέχων ο κόμβος με τη μικρότερη d(i) και συνεχίζει πάλι από το βήμα 2.

Η πολυπλοκότητα του αλγορίθμου εξαρτάται από τον τρόπο υλοποίησης της ουράς προτεραιότητας των κόμβων. Χρησιμοποιείται ουρά προτεραιότητας, διότι στο βήμα 5 για να τεθεί ένας κόμβος ως τρέχων, προτεραιότητα έχει ο κόμβος με τη μικρότερη d(i) εκείνη τη στιγμή. Απαιτούνται n inserts κορυφών στην ουρά, n λειτουργίες extract minimum και το πολύ m updates στην προτεραιότητα. Αν η ουρά υλοποιηθεί με απλό πίνακα ή λίστα, η πολυπλοκότητα είναι $O(n^2)$, καθώς η λειτουργία extract minimum είναι μια γραμμική αναζήτηση $O(n)$, ενώ η update $O(1)$. Αν η ουρά υλοποιηθεί με binary heap, η extract minimum απαιτεί $O(1)$ ενώ η insert και η update απαιτούν $O(logn)$, άρα ο αλγόριθμος έχει πολυπλοκότητα $O(mlogn)$. Ο πιο αποδοτικός τρόπος να υλοποιηθεί η ουρά είναι με Fibonacci heap, οπότε η πολυπλοκότητα του αλγορίθμου γίνεται $O(m+nlogn)$.

c) *Σε κάποιους κόμβους δίνονται προσφορές που μπορεί να είναι μεγαλύτερες από το κόστος διέλευσης (αλλά δεν υπάρχει κύκλος όπου συνολικά οι προσφορές να υπερβαίνουν το κόστος).*

Ο σωστός αλγόριθμος για αυτή την περίπτωση είναι ο Bellman-Ford. Αυτό ισχύει διότι στο σενάριο όπου οι ακμές έχουν αρνητικά βάρη, τότε ο αλγόριθμος του Dijkstra δεν δίνει πάντα σωστό αποτέλεσμα. Αυτό το ενδεχόμενο υφίσταται διότι αν θεωρηθούν δύο πόλεις A, B με A → B και έστω ότι η B έχει διόδια C και προσφορά G, τότε το βάρος της ακμής των A, B θα είναι: cost(A, B) = C_B - G_B . Ο ψευδοκώδικας είναι ο ακόλουθος:

```
proc Bellman_Ford
begin (/*initializations*/)
d(s) := 0; d(i) := oo /*infinity*/; prev(i) = null;
for each v!=0 do d := oo /*infinity*/
repeat n-1 times
    for each edge e = (u, v) do
        if d(u) + c(u, v) < d(v) then
            d(v) := d(u) + c(u,v); prev(V) := u;
end
```

Το κόστος της καλύτερης διαδρομής που έχει βρεθεί μέχρι εκείνη την στιγμή από τον αρχικό κόμβο s προς τον κόμβο i ορίζεται ως d(i). Αρχικοποιείται στις τιμές d(s) = 0, d(i) = ∞, ∀i ≠ s.
Ο προηγούμενος κόμβος από τον i στην καλύτερη διαδρομή που έχει βρεθεί ως εκείνη την στιγμή από τον s στον i ορίζεται ως prev(i). Αρχικοποιείται στην τιμή prev(i) = null, ∀i.
Σε έναν επαναληπτικό βρόχο, εκτελούνται τα εξής:
Για κάθε ακμή του γράφου (u, v) ελέγχεται αν d(u) + c(u, v) < d(v). Αν ισχύει, τότε τίθεται d(v) = d(u) + c(u, v) και p(v) = u, αλλιώς διατηρούνται οι υπάρχουσες τιμές των d(v), p(v).

Στο k-οστό βήμα της επανάληψης, το d(i) δηλώνει το ελάχιστο κόστος διαδρομής το πολύ ακμών από τον αρχικό κόμβο s προς τον κόμβο I, ενώ οι κόμβοι με d(i) = ∞ ή prev(i) = null δεν είναι προσβάσιμοι από τον s με διαδρομή k ακμών. Αφού κάθε απλό μονοπάτι έχει το πολύ n – 1 ακμές, στο τέλος του αλγορίθμου το d(i) θα εκφράζει το κόστος της καλύτερης διαδρομής από τον s στον i. Το prev(i) δηλώνει τον προηγούμενο κόμβο του i στην διαδρομή αυτή και χρησιμοποιείται για τον προσδιορισμό της διαδρομής. Οι κόμβοι με d(i) = ∞ ή prev(i) = null δεν είναι προσβάσιμοι με κανένα τρόπο από τον s. Η πολυπλοκότητα του αλγορίθμου είναι O(n*m) αφού γίνονται n – 1 επαναλήψεις m ελέγχων.

5η Άσκηση (Επιβεβαίωση Αποστάσεων – Δέντρο Συντομότερων Μονοπατιών)

Θεωρούμε ένα κατευθυνόμενο γράφημα G(V,E, ℓ) με n κορυφές, m ακμές και (ενδεχομένως αρνητικά) μήκος ℓ(e) σε κάθε ακμή του e ∈ E. Συμβολίζουμε με d(u, v) την απόσταση των κορυφών u και v (δηλ. το d(u, v) είναι ίσο με το μήκος της συντομότερης u − v διαδρομής) στο G. Δίνονται n αριθμοί $\delta_1,\dots,\delta_n$, όπου κάθε δ_k (υποτίθεται ότι) ισούται με την απόσταση $d(v_1, v_k)$ στο G. Να διατυπώσετε αλγόριθμο που σε χρόνο Θ(n + m), δηλ. γραμμικό στο μέγεθος του γραφήματος, ελέγχει αν τα $\delta_1,\dots,\delta_n$ πράγματι ανταποκρίνονται στις αποστάσεις των κορυφών από την v1, δηλαδή αν για κάθε $v_k \in V$, ισχύει ότι $\delta_k = d(v_1, v_k)$. Αν αυτό αληθεύει, ο αλγόριθμός σας πρέπει να υπολογίζει και να επιστρέφει ένα Δέντρο Συντομότερων Μονοπατιών με ρίζα τη v_1 (χωρίς ο χρόνος εκτέλεσης να ξεπεράσει το Θ(n + m)).

Η απάντηση προκύπτει από τα link:

- https://en.wikipedia.org/wiki/Shortest_path_problem
- http://mycourses.ntua.gr/courses/ECE1138/document/%C4%E9%E1%EB%DD%EE%E5%E9%F2_2019-20/theoritiki_pliroforiki_slides/focs_2019_20_5_graphs.pdf
- http://mycourses.ntua.gr/courses/ECE1138/document/%C4%E9%E1%EB%DD%EE%E5%E9%F2_2019-20/theoritiki_pliroforiki_slides/focs_2019_20_6_DynamicProgramming.pdf
- https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

6η Άσκηση (Προγραμματισμός Διακοπών)

Μπορούμε να αναπαραστήσουμε το οδικό δίκτυο μιας χώρας ως ένα συνεκτικό μη κατευθυνόμενο γράφημα G(V,E, ℓ) με n κορυφές και m ακμές. Κάθε πόλη αντιστοιχεί σε μια κορυφή του γραφήματος και κάθε οδική αρτηρία σε μία ακμή. Κάθε οδική αρτηρία $e \in E$ συνδέει δύο πόλεις και έχει μήκος $\ell(e)$ χιλιόμετρα. Η ιδιαιτερότητα της συγκεκριμένης χώρας είναι ότι έχουμε σταθμούς ανεφοδιασμού σε καύσιμα μόνο στις πόλεις / κορυφές του γραφήματος, όχι στις οδικές αρτηρίες / ακμές. Θέλουμε να ταξιδέψουμε από την πρωτεύουσα s σε ένα ορεινό θέρετρο t για διακοπές, για να αλλάξουμε παραστάσεις, μετά την άρση του lockdown. Θα χρησιμοποιήσουμε το αυτοκίνητό μας που διαθέτει αυτονομία καυσίμου για L χιλιόμετρα.

a) *Να διατυπώσετε έναν αλγόριθμο, με όσο το δυνατόν μικρότερη χρονική πολυπλοκότητα, που υπολογίζει αν κάτι τέτοιο είναι εφικτό. Ποια είναι η χρονική πολυπλοκότητα του αλγορίθμου σας στη χειρότερη περίπτωση;*

Ο υπολογισμός γίνεται με τον αλγόριθμο 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 το καύσιμο να φτάσει σε μια τουλάχιστον πόλη. Αναλυτικότερα, ισχύει:

- Αρχικά, ορίζεται ένα άδειο δέντρο T. Μετά, τίθεται ως $d(i)$ το ελάχιστο βάρος ακμής που συνδέει την πόλη i στο δέντρο και ισχύει ότι $d(i)=\infty\ \forall i \neq s$ και $d(s)=0$. Έπειτα, τίθεται ως $E(i)$ η ακμή ελάχιστου βάρους που συνδέει την πόλη i στο δέντρο και ισχύει ότι $E(i)=null\ \forall i$.
- Από τους κόμβους που δεν είναι στο T, επιλέγεται ο κόμβος u με το μικρότερο $d(u)$ και εισάγεται στο T. Αν όλοι οι κόμβοι είναι στο T, τότε ο αλγόριθμος τελείωσε.
- Τέλος, ενημερώνονται οι αποστάσεις των γειτονικών κόμβων v του u: αν $c(u,v)<d(v)$ τίθεται $d(v)=c(u,v)$ και $E(v)=(u,v)$. Μετά επιστρέφει στο βήμα 2.

Ουσιαστικά προστίθενται ένας-ένας κόμβοι στο δέντρο επιλέγοντας κάθε φορά εκείνον με τη μικρότερη απόσταση από αυτό. Όπως στον αλγόριθμο Dijkstra, χρησιμοποιείται ουρά προτεραιότητας, αφού στο βήμα 2 προτεραιότητα έχει ο κόμβος με το μικρότερο $d(i)$. Στο i-οστό βήμα της επανάληψης έχουν προστεθεί i κόμβοι στο δέντρο, άρα συνολικά γίνονται n επαναλήψεις. Συνεπώς απαιτούνται n διαδικασίες extract minimum και το πολύ m updates. Η πολυπλοκότητα του αλγορίθμου είναι επομένως $O(n^2)$ αν η ουρά προτεραιότητας υλοποιηθεί με απλό πίνακα ή λίστα, $O(mlogn)$ αν υλοποιηθεί με binary heap και $O(m+nlogn)$ αν υλοποιηθεί με Fibonacci heap.

Εφόσον λοιπόν, προσδιορίστηκε το ελάχιστο συνεκτικό δέντρο, μέσω του $E(t)$ υπολογίζεται το μονοπάτι ακμών μικρότερου μήκους που συνδέει την πόλη t με την πρωτεύουσα s. Συγκεκριμένα, το μονοπάτι είναι το σύνολο των ακμών $E(t)=(u_i,t), E(u_i)=(u_i-1,u_i),\dots, E(u_1)=(s,u_1)$ με $0 \leq i \leq n-1$. Το μονοπάτι αυτό τελικά πάντα οδηγεί στην πρωτεύουσα αφού ήταν η αρχική πόλη του δέντρου, αποτελείται από τις ακμές μικρότερου δυνατού μήκους και είναι μοναδικό. Αν στο μονοπάτι αυτό υπάρχει ακμή με απόσταση μεγαλύτερη από L, τότε το ταξίδι δεν είναι εφικτό.

b) *Να διατυπώσετε αλγόριθμο με χρονική πολυπλοκότητα (mlogm) που υπολογίζει την ελάχιστη αυτονομία καυσίμου (σε χιλιόμετρα) που απαιτείται για το ταξίδι από την πόλη s στην πόλη t.*

Ο υπολογισμός γίνεται με τον αλγόριθμο του Kruskal, ξεκινώντας από την ακμή με το λιγότερο κόστος, ενώνονται οι δύο 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|). Αναλυτικότερα, ισχύει:

- Αρχικά, ταξινομούνται οι ακμές κατά βάρος. Έπειτα, δημιουργείται δάσος F.
- Στην συνέχεια, επαναλαμβάνεται το επόμενο βήμα μέχρι να συνδεθούν όλοι οι κόμβοι.
- Τέλος, επιλέγεται η μικρότερη ακμή και ελέγχεται αν συνδέει δύο διαφορετικά δέντρα του F, δηλαδή δε δημιουργεί κύκλο. Στην περίπτωση αυτή εισάγεται η ακμή στο δάσος, αλλιώς αγνοείται.

Στο τέλος του αλγορίθμου έχουν χρησιμοποιηθεί οι μικρότερες δυνατές ακμές για να συνδεθούν όλοι οι κόμβοι χωρίς κύκλους, άρα δημιουργήθηκε το ελάχιστο συνεκτικό δέντρο. Η ταξινόμηση των ακμών απαιτεί $O(mlogm)$. Έπειτα γίνονται n επαναλήψεις, άρα n έλεγχοι για κύκλο όπου θα γίνει χρήση δομής union-find. Η union by rank και η find γίνονται σε $O(logn)$. Συνεπώς η πολυπλοκότητα του αλγορίθμου είναι $O(mlogm)$=$O(mlogn)$ καθώς $m \leq n^2$ και $O(mlogn^2)$=$O(2mlogn)$=$O(mlogn)$. Εν τέλει, χρησιμοποιείται DFS ή BFS (πολυπλοκότητα $O(n+m)$) για την εύρεση του μονοπατιού από την πόλη s στην πόλη t, και η μεγαλύτερη ακμή στο μονοπάτι αυτό είναι η ελάχιστη αυτονομία καυσίμου που χρειάζεται για το ταξίδι.

c) *Να διατυπώσετε αλγόριθμο με γραμμική χρονική πολυπλοκότητα που υπολογίζει την ελάχιστη αυτονομία καυσίμου για το ταξίδι από την πόλη s στην πόλη t. Υπόδειξη: Εδώ μπορεί να σας φανεί χρήσιμο ότι (με λογική αντίστοιχη με αυτή της Quicksort) μπορούμε να υπολογίσουμε τον median ενός μη ταξινομημένου πίνακα σε γραμμικό χρόνο.*

Αναλυτικότερα, στα link:

- https://brilliant.org/wiki/median-finding-algorithm/
- https://www.geeksforgeeks.org/program-for-mean-and-median-of-an-unsorted-array/

7η Άσκηση (Αγορά Εισιτηρίων)

Εκτός από τις διακοπές σας, έχετε προγραμματίσει προσεκτικά και την παρουσία σας στη Σχολή, μετά την άρση του lockdown. Συγκεκριμένα, έχετε σημειώσει ποιες από τις Τ ημέρες, που θα ακολουθήσουν την άρση του lockdown, θα έρθετε στο ΕΜΠ για να παρακολουθήσετε μαθήματα και εργαστήρια και να δώσετε εξετάσεις. Το πρόγραμμά σας έχει τη μορφή ενός πίνακα S με Τ θέσεις, όπου για κάθε ημέρα t = 1, ... , T, S[t] = 1, αν θα έρθετε στο ΕΜΠ, και S[t] = 0, διαφορετικά. Το πρόγραμμά σας δεν έχει κανονικότητα και επηρεάζεται από διάφορες υποχρεώσεις και γεγονότα. Για κάθε μέρα που θα έρθετε στο ΕΜΠ, πρέπει να αγοράσετε εισιτήριο που να επιτρέπει τη μετακίνησή σας με τις αστικές συγκοινωνίες. Υπάρχουν συνολικά k διαφορετικοί τύποι εισιτηρίων (π.χ., ημερήσιο, τριών ημερών, εβδομαδιαίο, μηναίο, εξαμηνιαίο, ετήσιο). Ο τύπος εισιτηρίου i σας επιτρέπει να μετακινηθείτε για c_i διαδοχικές ημέρες (μπορεί βέβαια κάποιες από αυτές να μην χρειάζεται να έρθετε στο ΕΜΠ) και κοστίζει p_i ευρώ. Για τους τύπους των εισιτηρίων, ισχύει ότι $1 = c_1 < c_2 < ... < c_k \leq T$, $p_1 < p_2 < ... < p_k$, και $p_1/c_1 > p_2/c_2 > ... > p_k/c_k$ (δηλαδή, η τιμή αυξάνεται με τη διάρκεια του εισιτηρίου, αλλά η τιμή ανά ημέρα μειώνεται). Να διατυπώσετε αλγόριθμο που υπολογίζει τον συνδυασμό τύπων εισιτηρίων με ελάχιστο συνολικό κόστος που καλύπτουν όλες τις ημέρες που θα έρθετε στο ΕΜΠ. Ποια είναι η χρονική πολυπλοκότητα του αλγορίθμου σας στη χειρότερη περίπτωση;

Για τον απαιτούμενο αλγόριθμο θα γίνει χρήση δύο πινάκων Τ θέσεων, Α και Β αντίστοιχα. Στον Α θα αποθηκευτούν στην i θέση η ελάχιστη τιμή για i μέρες ανάλογα με τον πίνακα S όπου ενημερώνει ποιες μέρες θα χρειαστούν εισιτήρια. Στον Β θα αποθηκεύεται 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] + pk και στην συνέχεια θα γίνει έλεγχος για όλα τα ck αν min > A[x – ck] + pk και τότε θα γίνει αντικατάσταση του min με την καινούρια τιμή. Ο Β γεμίζει ανάλογα με B[i] = B[i – ck] + ck. Ακολουθώντας την ίδια διαδικασία μέχρι το i να ξεπεράσει το Τ, ο αλγόριθμος ολοκληρώνεται. Η πολυπλοκότητα είναι O(T*K).

Με άλλα λόγια, έστω $a[t]$ το ελάχιστο απαιτούμενο κόστος από την αρχή μέχρι την t ημέρα και επίσης $a[0]=0$. Για $t>0$:

- Αν $S[t]=0$ τότε $a[t]=a[t-1]$.
- Αν $S[t]=1$ τότε $a[t]=\min\{a[t-c_k]+p_k$

Δηλαδή, αν την t ημέρα δεν βρεθεί κάποιος στο ΕΜΠ, δε θα πάρει σίγουρα νέο εισιτήριο εκείνη τη μέρα, άρα το κόστος είναι το ίδιο με την προηγούμενη μέρα. Αν την t ημέρα βρεθεί κάποιος στο ΕΜΠ, τότε το κόστος θα ισούται με το κόστος c_i μέρες πριν αν προστεθεί το κόστος εισιτηρίου p_i. Είναι θεμιτή η φτηνότερη επιλογή άρα επιλέγεται το ελάχιστο από όλους τους τύπους εισιτηρίων.

Για κάθε t το πρόβλημα τελικά ανάγεται στην base case $a[0]=0$ και λύνεται bottom up. Κάθε υποπρόβλημα $a[t]$ που λύνεται, αποθηκεύεται προς μελλοντική χρήση. Για να βρεθεί ο συνδυασμός εισιτηρίων που δίνει το ελάχιστο κόστος, όταν υπολογίζεται το min θα εκτυπώνονται τα $t,t-c_i,p_i$ (ή θα αποθηκεύονται σε κάποια δομή δεδομένων που θα δίνεται ως έξοδος).

Ο υπολογισμός του $a[t]$, αν είναι ήδη γνωστές οι τιμές κάθε $a[i],i<t$, απαιτεί $O(k)$, ενώ τα $a[i],i<t$ πάντα θα είναι ήδη γνωστά λόγω της bottom up επίλυσης (εκτός από όταν η αναδρομή φτάσει στην base case όπου θα είναι $O(1)$). Στη χειρότερη περίπτωση οι συγκρίσεις θα γίνουν t φορές άρα πολυπλοκότητα $O(t*k)$.

8η Άσκηση (Αντιπροσωπεία Φορητών Υπολογιστών)

Επιθυμούμε να βελτιστοποιήσουμε την λειτουργία μιας νέας αντιπροσωπείας φορητών υπολογιστών για τις επόμενες n ημέρες. Για κάθε ημέρα i, $1 \leq i \leq n$, υπάρχει μια (απόλυτα ασφαλής) πρόβλεψη d_i του αριθμού των πωλήσεων. Όλες οι πωλήσεις λαμβάνουν χώρα το μεσημέρι, και όσοι φορητοί υπολογιστές δεν πωληθούν, αποθηκεύονται. Υπάρχει δυνατότητα αποθήκευσης μέχρι S υπολογιστών, και το κόστος είναι C για κάθε υπολογιστή που αποθηκεύεται και για κάθε ημέρα αποθήκευσης. Το κόστος μεταφοράς για την προμήθεια νέων υπολογιστών είναι K ευρώ, ανεξάρτητα από το πλήθος των υπολογιστών που προμηθευόμαστε, και οι νέοι υπολογιστές φθάνουν λίγο πριν το μεσημέρι (άρα αν πωληθούν αυθημερόν, δεν χρειάζονται αποθήκευση). Αρχικά δεν υπάρχουν καθόλου υπολογιστές στην αντιπροσωπεία.

Το ζητούμενο είναι να προσδιορισθούν οι παραγγελίες (δηλ. πόσους υπολογιστές θα παραγγείλουμε και πότε) ώστε να ικανοποιηθούν οι προβλεπόμενες πωλήσεις με το ελάχιστο δυνατό συνολικό κόστος (αποθήκευσης και μεταφοράς). Να διατυπώσετε αλγόριθμο δυναμικού προγραμματισμού με χρόνο εκτέλεσης πολυωνυμικό στο nS για τη βελτιστοποίηση της λειτουργίας της αντιπροσωπείας.

Μια προσέγγιση αποτελεί ο *Simplex Algorithm*.

Πιο συγκεκριμένα, θεωρώντας:

ft = min {cost (t, j) + f(t+j+1), 0 ≤ j ≤ n-t, storage (t, j) <= S}

όπου cost(t, j) = jd(t+j) + (j-1)d(t+j-1) + ... + d(t+1)

και storage(t,j) = d(t+j) + d(t+j-1) + ... + d(t+1)

και το t τρέχει από n έως 1

Αναλυτικότερα, στα link:

- https://en.wikipedia.org/wiki/Simplex_algorithm
- https://www.geeksforgeeks.org/simplex-algorithm-tabular-method/