2η Σειρά Ασκήσεων FoCS

Γιάννης Πολυχρονόπουλος 03121089

Άσκηση 1. (Αναδρομή – Επανάληψη – Επαγωγή)

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

(β) Δείξτε ότι ο αριθμός κινήσεων του αναδρομικού ισούται με τον αριθμό μετακινήσεων του επαναληπτικού αλγορίθμου. *Προσοχή: θα πρέπει να ορίσετε προσεκτικά την 'θετική φορά' ώστε να ισχύει αυτό.*

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

(δ) 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.

(α) Έστω πως η συνάρτηση $Recursion(n)$ μας δίνει τον αριθμό κινήσεων δίσκων που κάνει ο αναδρομικός αλγόριθμος για τους πύργους του Hanoi.

Ο αναδρομικός αλγόριθμος δουλεύει ως εξής:

- Για $n = 1$, μετακινούμε τον ένα δίσκο στον τελικό πάσσαλο άρα $Recursion(1) = 1$
- Για $n > 1$:
  - Μετακινούμε $n - 1$ δίσκους από τον αρχικό πάσσαλο στον βοηθητικό.
  - Μετακινούμε τον n-th δίσκο(δηλαδή τον μεγαλύτερο στον τελικό πάσσαλο.
  - Μετακινούμε τους $n - 1$ δίσκους από τον βοηθητικό πάσσαλο στον τελικό.

  Επομένως για $n > 1$: $Recursion(n) = Recursion(n-1) + 1 + Recursion(n-1)$

$$\Leftrightarrow Recursion(n) = 2Recursion(n-1) + 1$$

n	1	2	3	4
$Recursion(n)$	1	$2Recursion(1)+1$	$2Recursion(2)+1$	$2Recursion(3)+1$
$Recursion(n)$	1	3	7	15

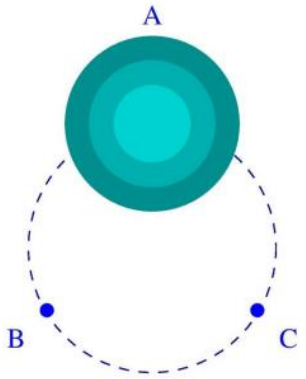
Φαίνεται πως η συνάρτηση $Recursion(n)$ ακολουθεί το μοτίβο $Recursion(n) = 2^n - 1$.

- Για $n = 1$: $Recursion(1) = 2^1 - 1 = 1$, που ισχύει
- Έστω πως ισχύει $Recursion(n) = 2^n - 1$
- Τότε, $Recursion(n+1) = 2Recursion(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$

Επομένως, με επαγωγή αποδείξαμε πως $Recursion(n) = 2^n - 1$.

(β) Έστω πως η συνάρτηση $Iteration(n)$ μας δίνει τον αριθμό κινήσεων δίσκων που κάνει ο επαναληπτικός αλγόριθμος.

Ορίζουμε την θετική φορά του αλγορίθμου ως εξής:



- Αν το n είναι περιττός τότε κινούμαστε κατά τη φορά των δεικτών του ρολογιού.
- Ενώ αν είναι άρτιος κινούμαστε αντίθετα.

Θέλουμε να δείξουμε πως $Iteration(n) = 2^n - 1$. Αυτό θα το κάνουμε με επαγωγή.

- Για n=1: $Iteration(1) = 1$ (καθώς απλώς μετακινούμε τον δίσκο κατά την θετική φορά δηλαδή ως προς το C, αφού το n είναι περιττός)
- Για n=k έστω πως $Iteration(k) = 2^k - 1$.
- Για n=k+1 τότε για να μετακινηθεί ο μεγαλύτερος δίσκος στον πάσσαλο C πρέπει πρώτα όλοι οι υπόλοιποι δίσκοι να μετακινηθούν στον πάσσαλο B δηλαδή έχουμε πάλι το πρόβλημα Hanoi Towers με κ δίσκους, το B να είναι ο τελικός πάσσαλος και το C ο βοηθητικός. Αφού μετακινηθούν οι κ δίσκοι στον πάσσαλο B μετακινούμε τον μεγαλύτερο δίσκο στο C (καθώς είναι η μοναδική επιτρεπτή κίνηση που δεν αφορά τον μικρότερο δίσκο) και έπειτα πρέπει να μετακινήσουμε τους κ δίσκους από το B στο C. Επομένως, $Iteration(k+1) = Iteration(k) + 1 + Iteration(k) = 2^{k+1} - 1$.

Συνεπώς, με επαγωγή αποδείξαμε πως $Iteration(n) = 2^n - 1 = Recursion(n)$.

(γ) Ας υποθέσουμε πως η συνάρτηση $Steps(n)$ υπολογίζει τον αριθμό των κινήσεων που απαιτούνται για να μεταφερθούν οι n δίσκου από τον αρχικό στον τελικό πάσσαλο. Προφανώς για να λυθεί το πρόβλημα αυτό πρέπει ο μεγαλύτερος δίσκος να μεταφερθεί στον τελικό πάσσαλο. Ωστόσο, για να γίνει αυτό πρέπει οι υπόλοιποι n-1 δίσκοι να μεταφερθούν στον βοηθητικό πάσσαλο. Έπειτα μετακινούμε τον μεγαλύτερο δίσκο στον τελικό πάσσαλο και στην συνέχεια μετακινούμε τους υπόλοιπους n-1 δίσκους στον τελικό. Η μετακίνηση των n-1 δίσκων ωστόσο αποτελεί ένα πρόβλημα Hanoi με n-1 δίσκους.

Επομένως, απαιτούνται τουλάχιστον $Steps(n-1) + 1 + Steps(n-1)$ βήματα για την μετακίνηση όλων των δίσκων.

$Steps(n) \geq 2Steps(n-1) + 1$, Άρα το $Steps(n)$ ελαχιστοποιείται όταν:

$$Steps(n) = 2Steps(n-1) + 1 \Leftrightarrow \boldsymbol{Steps(n) = 2^n - 1 = Recursion(n) = Iteration(n)}$$

Έχουμε λύσει ξανά την ίδια αναδρομική σχέση στο ερώτημα (α).

(δ) Έχοντας έναν επιπλέον πάσσαλο έχουμε την δυνατότητα να μειώσουμε κατά μεγάλο βαθμό τον αριθμό βημάτων και την επίλυση του προβλήματός μας χρησιμοποιώντας τον εξής αλγόριθμο:

- Έστω η συνάρτηση $Steps(n)_4$ υπολογίζει τον αριθμό των κινήσεων που απαιτούνται για να μεταφερθούν οι n δίσκου από τον αρχικό στον τελικό πάσσαλο, με δύο βοηθητικούς πασσάλους.
- Για $n = 1$: $Steps(1)_4 = 1$, καθώς μετακινούμε τον δίσκο από τον αρχικό πάσσαλο στον τελικό.
- Για $n = 2$: $Steps(2)_4 = 3$, καθώς μετακινούμε τον μικρότερο δίσκο σε ένα βοηθητικό πάσσαλο, στη συνέχεια μετακινούμε τον μεγαλύτερο δίσκο στον τελικό πάσσαλο και κατόπιν τοποθετούμε επίσης τον μικρότερο δίσκο στο τελικό πάσσαλο.
- Για $n > 2$: Μετακινούμε τους $n - 2$ μικρότερους δίσκους σε έναν βοηθητικό πάσσαλο. Έπειτα μετακινούμε τους 2 μεγαλύτερους δίσκους στον τελικό πάσσαλο με την βοήθεια του άλλου βοηθητικού πασσάλου. Τέλος μετακινούμε τους $n - 2$ μικρότερους δίσκους στον τελικό πάσσαλο.

Επομένως, η αναδρομική σχέση είναι η εξής:

$Steps(n)_4 = 2Steps(n-2)_4 + 3, \quad με\ Steps(1)_4 = 1\ και\ Steps(2)_4 = 3$

Αποτελεί αναδρομική σχέση με βήμα 2, άρα:

Για $n = 2k\ (άρτιος)$, $\quad Steps(2k)_4 = 2Steps(2k-2)_4 + 3,\ με\ Steps(2)_4 = 3$

Για $n = 2k+1\ (περιττός)$, $\quad Steps(2k+1)_4 = 2Steps(2k-1)_4 + 3, με\ Steps(1)_4 = 1$

```
def steps(n):
    if (n == 1):
        return 1
    elif (n == 2):
        return 3
    return 2*steps(n-2)+3


for n in range(1, 100):
    print(steps(2*n - 1), steps(2*n))
```

Με την χρήση του παραπάνω κώδικα παράγουμε τις τιμές της συνάρτησης $Steps(n)_4$ για άρτιους και περιττούς αριθμούς και παρατηρούμε το εξής μοτίβο:

- Για $n = 2k$ $\quad Steps(2k)_4 = 3\left(2^k - 1\right) \overset{k=\frac{n}{2}}{\Longleftrightarrow} Steps(n)_4 = 3\left(2^{\frac{n}{2}} - 1\right)$
- Για $n = 2k + 1$ $Steps(2k+1)_4 = 2^{k+2} - 3 \overset{k=\frac{n-1}{2}}{\Longleftrightarrow} Steps(n)_4 = 2^{\frac{n+1}{2}} - 3$

Θα αποδείξουμε επαγωγικά πως αυτές οι σχέσεις ισχύουν

- Για $n = 2k$:
  - Για $k = 1$: $Steps(2)_4 = 3(2-1) = 3$,άρα ισχύει
  - Έστω πως ισχύει για λ: $Steps(2\lambda)_4 = 3\left(2^{\lambda} - 1\right)$
  - Για λ+1:

    $$Steps(2\lambda + 2)_4 = 2\,Steps(2\lambda)_4 + 3 = 3\left(2^{\lambda+1} - 2\right) + 3 = 3\left(2^{\lambda+1} - 1\right)$$
- Για $n = 2k + 1$:
  - Για $k = 0$: $Steps(1)_4 = 2^2 - 3 = 1$,άρα ισχύει
  - Έστω πως ισχύει για λ: $Steps(2\lambda + 1)_4 = 2^{\lambda+2} - 3$
  - Για λ+1: $Steps(2\lambda + 3)_4 = 2Steps(2k+1)_4 + 3 = 2^{\lambda+2} - 6 + 3 = 2^{\lambda+2} - 3$

Άσκηση 2. (Επαναλαμβανόμενος Τετραγωνισμός – Κρυπτογραφία)

(α) 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:

Αν n πρώτος τότε για κάθε a τ.ώ. $1 < a < n - 1$, ισχύει

$$a^{n-1} mod\ n = 1$$

Αν λοιπόν, για δεδομένο n βρεθεί a ώστε να μην ισχύει η παραπάνω ισότητα τότε ο αριθμός n είναι οπωσδήποτε σύνθετος. Αν η ισότητα ισχύει για το συγκεκριμένο a, τότε η δοκιμή πρέπει να επαναληφθεί με νέο a, καθώς υπάρχει περίπτωση ο αριθμός να είναι σύνθετος και παρ'όλα αυτά η ισότητα να ισχύει για κάποιες τιμές του a. Μια ενδιαφέρουσα ιδιότητα λέει ότι, αν ο n είναι σύνθετος, η πιθανότητα να ισχύει η ισότητα είναι ≤ 1/2 (αυτό ισχύει για όλα τα n εκτός από κάποιες 'παθολογικές' περιπτώσεις, που λέγονται αριθμοί Carmichael, δείτε Σημ. 2 παρακάτω). Έτσι, μπορούμε να αυξήσουμε σημαντικά την πιθανότητα επιτυχίας (δηλ. της επιβεβαίωσης της συνθετότητας του αριθμού n) επαναλαμβάνοντας μερικές φορές τη δοκιμή (τυπικά 30 φορές) με διαφορετικό a. Αν όλες τις φορές βρεθεί να ισχύει η παραπάνω ισότητα τότε λέμε ότι το n "περνάει το test" και ανακηρύσσουμε το n πρώτο αριθμό· αν έστω και μία φορά αποτύχει ο έλεγχος, τότε είμαστε βέβαιοι ότι ο αριθμός είναι σύνθετος.

Το πρόγραμμά σας θα πρέπει να δουλεύει σωστά για αριθμούς χιλιάδων ψηφίων. Δοκιμάστε την με τους αριθμούς:

$$67280421310721, 170141183460469231731687303715884105721,\ 2^{2281} - 1, 2^{9941} - 1$$

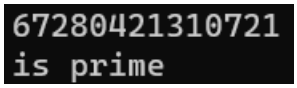
Σημείωση 1: το $a^{2^{9941}-2}$ έχει 'αστρονομικά' μεγάλο πλήθος ψηφίων (δεν χωράει να γραφτεί ούτε σε ολόκληρο το σύμπαν!), ενώ το a $a^{2^{9941}-1} mod(2^{9941} - 1)$ είναι σχετικά "μικρό" (έχει μερικές χιλιάδες δεκαδικά ψηφία μόνο :-)) οπότε είναι δυνατόν να το υπολογίσουμε (με λίγη προσοχή).
Σημείωση 2: Υπάρχουν (λίγοι) σύνθετοι που έχουν την ιδιότητα να περνούν τον έλεγχο Fermat για κάθε a που είναι σχετικά πρώτο με το n, οπότε για αυτούς το test θα αποτύχει όσες δοκιμές και αν γίνουν (εκτός αν πετύχουμε κατά τύχη a που δεν είναι σχετικά πρώτο με το n, πράγμα αρκετά απίθανο για αρκετά μεγάλο n). Αυτοί οι αριθμοί λέγονται Carmichael – δείτε και http://en.wikipedia.org/wiki/Carmichael_number. Ελέγξτε τη συνάρτησή σας με αρκετά μεγάλους αριθμούς Carmichael που θα βρείτε π.χ. στη σελίδα http://de.wikibooks.org/wiki/Pseudoprimzahlen:_Tabelle_Carmichael-Zahlen. Τι παρατηρείτε;
(β) Μελετήστε και υλοποιήστε τον έλεγχο Miller-Rabin (π.χ. από τις σημειώσεις που θα βρείτε στη σελίδα του μαθήματος στο Helios) 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. Βλέπετε κάτι περίεργο; Πώς το εξηγείτε;

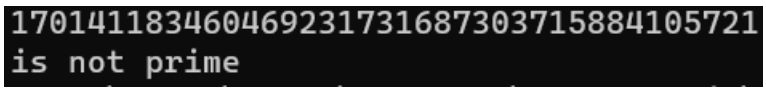
(γ) Γράψτε πρόγραμμα που να βρίσκει όλους τους πρώτους αριθμούς Mersenne, δηλαδή της μορφής $n = 2^x - 1$ με $1 < x < 200$ (σημειώστε ότι αν το x δεν είναι πρώτος, ούτε το $2^x - 1$ είναι πρώτος – μπορείτε να το αποδείξετε;). Αντιπαραβάλετε με όσα αναφέρονται στην ιστοσελίδα https://www. mersenne.org/primes

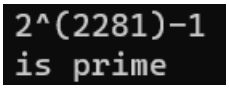
(α) Η γλώσσα προγραμματισμού που επέλεξα για την υλοποίηση του Fermat's Test είναι η python και ο κώδικας είναι ο εξής:

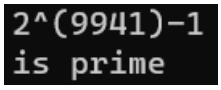
```
# Library for generating random numbers
import random

# If n is prime, then always returns true.
# If n is composite than returns false with high probability.
# As the value of k increases the probability that our result is correct
also increases

def Fermat(n, k):

    # Corner cases
    if n == 1 or n == 4:
        return False
    elif n == 2 or n == 3:
        return True

    # Try k times
    else:
        for i in range(k):

            # Pick a random number in the interval [2,n-1]
            a = random.randint(2, n - 1)

            # Fermat's test
            if pow(a, n-1, n) != 1:  # a^(n-1)%n
                return False

    return True


k = 30  # We typically initialize k at this value

test_number = int(input()) # the number that we are going to test
if (Fermat(test_number, k)):
    print(f"is prime")
else:
    print(f"is not prime")
```

Για τους αριθμούς που δίνεται στην άσκηση η έξοδος του προγράμματος είναι η εξής:



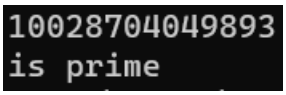






Επομένως, συμπεραίνουμε πως το πρόγραμμα μας λειτουργεί και για πολύ μεγάλους αριθμούς χρησιμοποιώντας φυσικά την μέθοδο modular exponentiation.

Ωστόσο, ο αλγόριθμος αυτός δεν είναι ορθός όταν βάζουμε για είσοδο μεγάλους αριθμούς Carmichael. Για παράδειγμα, για εισόδους 973694665856161, 10028704049893, … κ.α ο αλγόριθμός μας αποφαίνεται πως οι αριθμοί αυτοί είναι πρώτοι, ενώ είναι σύνθετοι.



(β) Πάλι η γλώσσα που χρησιμοποιήθηκε για την επίλυση του ερωτήματος είναι η python και ο αλγόριθμος είναι ο εξής:

- Βρίσκουμε $n-1=2^s \cdot d$ με $n>2$ και d περιττός
- Διαλέγουμε a: $0<a<n-1$
- Παράγουμε την ακολουθία b_n $με$ $b_0=a^d mod\ n$ και $b_i=b_{i-1}^2 mod\ n,\ 0 \le i < s$
    - Αν $b_0=1$ ή $b_0=n-1$, τότε το n είναι το πιο πιθανόν πρώτος
    - Αν $b_i=-1(mod\ n)=n-1$, τότε το n είναι το πιο πιθανόν πρώτος

```
# Library for generating random numbers
import random
def Miller_Rabin_Test(d, s, n):

    # Pick a random number from the interval [1, n - 2]
    a = random.randint(1, n - 2)

    # Compute a^d % n
    b = pow(a, d, n)

    if (b == 1 or b == n - 1):
        return True

    # Keep squaring x while one of the following doesn't happen
    # (i) d does not reach n-1
    # (ii) (b^2) % n is not 1
    # (iii) (b^2) % n is not n-1
```

```
    for i in range(s):
        b = (b * b) % n
        # If b=1 then all bi will be 1, so in conclusion n is composite
        if (b == 1):
            return False
        if (b == n - 1):
            return True

    # Return composite
    return False

# The following function returns false if n is composite and returns
# True if n is probably prime.
# k is an input parameter that determines accuracy level.
# Higher value of k indicates more accuracy.
def isPrime(n, k):

    # Corner cases
    if (n == 1 or n == 4):
        return False
    if (n == 2 or n == 3):
        return True
    # Find r such that n - 1 = 2^s * d for some d >= 1
    d = n - 1
    s = 0
    while (d % 2 == 0):
        s = s+1
        d //= 2
    # Iterate given number of 'k' times
    for i in range(k):
        if (Miller_Rabin_Test(d, s, n) == False):
            return False

    return True
k = 30
# With k=30 the probability that there is going to be an error is slim to
# none
n = int(input())
if (isPrime(n, k)):
    print("is prime")
else:
    print("is not prime")
```

Ο αλγόριθμος αυτός αποτελεί μια βελτίωση του Fermat's test. Συγκεκριμένα, όταν παίρνουμε για είσοδο μεγάλους αριθμούς Carmichael που το Fermat's test έδινε εσφαλμένο αποτέλεσμα, ο παραπάνω αλγόριθμος μπορεί να καταλήξει σε σωστά αποτελέσματα.

```
10028704049893
is not prime
```

```
973694665856161
is not prime
```

Παράλληλα, μπορούμε να πούμε πως με 30 επαναλήψεις του Miller-Rabin test η πιθανότητα εσφαλμένου αποτελέσματος είναι απειροελάχιστη με αποτέλεσμα να ισχυροποιούμε την πεποίθησή μας πως το Miller-Rabin primality test είναι πιο ισχυρό από το Fermat's primality test.

Πηγές: https://en.wikipedia.org/wiki/Miller–Rabin_primality_test

https://www.youtube.com/watch?v=8i0UnX7Snkc&t=234s

(γ) Αρχικά, θα αποδείξουμε πως αν το x δεν είναι πρώτος, ούτε το $2^x - 1$ είναι πρώτος.

Εφόσον το x δεν είναι πρώτος αριθμός μπορεί να γραφτεί ως γινόμενο 2 αριθμών: $x = a \cdot b$

Τότε έχουμε: $2^x - 1 = 2^{a \cdot b} - 1 = (2^a - 1)\left(1 + 2^a + 2^{2a} + 2^{3a} + \cdots + 2^{(b-1)a}\right) =$

$$= (2^b - 1)(1 + 2^b + 2^{2b} + 2^{3a} + \cdots + 2^{(a-1)b})$$

Επομένως, το $2^x - 1$ δεν είναι πρώτος αριθμός. Αντίστροφα μπορούμε να πούμε πως για να είναι ο αριθμός $2^x - 1$ πρώτος πρέπει και το x να είναι πρώτος.

Στο κώδικα που θα υλοποιήσουμε αρκεί να βρούμε όλους τους πρώτους αριθμούς με έναν από τους 2 τρόπους στα προηγούμενα ερωτήματα και στη συνέχεια να ελέγξουμε αν το $2^x - 1$ είναι πρώτος.

```python
k = 30
# With k = 30 the chance that the algorithm is going to give us a wrong
# answer is pretty low
for x in range(1, 200):
    if (isPrime(x, k)):  # Using the Miller-Rabin test
        n = 2**x - 1  # Calculate 2^x-1
        if (isPrime(n, k)):
            print(f"x: {x} 2^x-1: {n}")
# Print the pair x and 2^x-1, where x is prime and bound from 1 to 200
```

```
x: 2 2^x-1: 3
x: 3 2^x-1: 7
x: 5 2^x-1: 31
x: 7 2^x-1: 127
x: 13 2^x-1: 8191
x: 17 2^x-1: 131071
x: 19 2^x-1: 524287
x: 31 2^x-1: 2147483647
x: 61 2^x-1: 2305843009213693951
x: 89 2^x-1: 618970019642690137449562111
x: 107 2^x-1: 162259276829213363391578010288127
x: 127 2^x-1: 170141183460469231731687303715884105727
```

Πράγματι αυτοί είναι οι πρώτοι αριθμοί Mersenee, σύμφωνα και με την ιστοσελίδα https://www. mersenne.org/primes/

#	2^p-1	Digits	Date Discovered	Discovered By	Method / Hardware	Perfect Number
1	2^2-1	1	c. 500 BCE	Ancient Greek mathematicians		$2^1 \cdot (2^2-1)$
2	2^3-1	1	c. 500 BCE	Ancient Greek mathematicians		$2^2 \cdot (2^3-1)$
3	2^5-1	2	c. 275 BCE	Ancient Greek mathematicians		$2^4 \cdot (2^5-1)$
4	2^7-1	3	c. 275 BCE	Ancient Greek mathematicians		$2^6 \cdot (2^7-1)$
5	2^{13}-1	4	1456	Anonymous	trial division	$2^{12} \cdot (2^{13}-1)$
6	2^{17}-1	6	1588	Pietro Cataldi	trial division	$2^{16} \cdot (2^{17}-1)$
7	2^{19}-1	6	1588	Pietro Cataldi	trial division	$2^{18} \cdot (2^{19}-1)$
8	2^{31}-1	10	1772	Leonhard Euler	Enhanced trial division	$2^{30} \cdot (2^{31}-1)$
9	2^{61}-1	19	1883	Ivan Mikheevich Pervushin	Lucas sequences	$2^{60} \cdot (2^{61}-1)$
10	2^{89}-1	27	1911 Jun	R. E. Powers	Lucas sequences	$2^{88} \cdot (2^{89}-1)$
11	2^{107}-1	33	1914 Jun 11	R. E. Powers	Lucas sequences	$2^{106} \cdot (2^{107}-1)$
12	2^{127}-1	39	1876 Jan 10	Édouard Lucas	Lucas sequences	$2^{126} \cdot (2^{127}-1)$

Άσκηση 3. (Αριθμοί Fibonacci)

(α) 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;

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

(γ) 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.

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

(α)

Αρχικά, υλοποιούμε τον αναδρομικό αλγόριθμο με memoization στην γλώσσα προγραμματισμού python.

```python
# This algorithm has time complexity of O(n) and space complexity O(n)
# In particular it has an upper bound of 2n invocations
def memo_fib(n, memo):
    # If the term we are trying to compute isn't cached then store it
    # In our dictionary
    if n not in memo:
        memo[n] = memo_fib(n-1, memo) + memo_fib(n-2, memo)
    # Then we return the desired value
    return memo[n]


# Create a dictionary which stores the n-th position as it's key
and      # the value of the nth term as it's value
memo = {0: 0, 1: 1}
n = int(input())
print(f"The {n} term is {memo_fib(n, memo)}")
```

Στη συνέχεια, υλοποιούμε τον επαναληπτικό αλγόριθμο.

```python
# This iterative algorithm has a time complexity of O(n) and space
# complexity of O(1)
def iterative_fib(n):
    if (n <= 1):
       # Corner cases of n=0 or n=1
        return n
    else:
        previous = 0  # Fn-2
        current = 1  # Fn-1
        for i in range(1, n):
            next = current + previous  # Fn = Fn-1 + Fn-2
            previous = current
            current = next
        return next


n = int(input())
print(f"The {n} term is {iterative_fib(n)}")
```

Και κατόπιν, υλοποιούμε τον αλγόριθμο με τον πίνακα, καθώς γνωρίζουμε πως:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

```python
# This function multiplies two 2x2 matrices
def multiply(F, M):

    x = (F[0][0] * M[0][0] +
         F[0][1] * M[1][0])
    y = (F[0][0] * M[0][1] +
         F[0][1] * M[1][1])
    z = (F[1][0] * M[0][0] +
         F[1][1] * M[1][0])
    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
```

```python
# Auxilary function to compute a 2x2 matrix raised to the n-th power


def power(F, n):

    if (n == 0 or n == 1 or n == 2):
        return
    M = [[1, 1],
         [1, 0]]

    power(F, n // 2)  # Fast powering Gauss algorithm
    multiply(F, F)

    if (n % 2 != 0):
        multiply(F, M)

# This algorithm has a time complexity of O(logn)


def matrix_fib(n):
    # Creating the matrix we are going to raise to the (n-1) power
    F = [[1, 1]
         [1, 0]]
    if (n <= 1):
        return n
    power(F, n - 1)

    return F[0][0]  # This is Fn


n = int(input())
print(f"The {n} term is {matrix_fib(n)}")
```

Παρατηρούμε πως ο πιο αποδοτικός αλγόριθμος σε χρόνο είναι ο τελευταίος με πολυπλοκότητα $O(logn)$ και ο πιο αποδοτικός αλγόριθμος σε χώρο είναι ο επαναληπτικός με πολυπλοκότητα $O(1)$ ($O(n)$ $time\ complexity$).

(β) Γενικά ισχύει: $F_n = \frac{(\varphi^n+\psi^n)}{\sqrt{5}}$, όπου $\varphi = \frac{1+\sqrt{5}}{2}$, $\psi = \frac{1-\sqrt{5}}{2} = 1-\varphi$

Επομένως : $F_n = \frac{(\varphi^n+(1-\varphi)^n)}{\sqrt{5}}$

```
# The math library allows us to use the sqrt function
import math
# This algorithm has a time complexity of O(logn)
phi = (1+math.sqrt(5))/2  # Initializing phi
n = int(input())
# Finding the n-th term with the above formula
result = round((phi**n + (1-phi)**n)/math.sqrt(5))
print(f"The {n} term is {result}")
```

Ο παραπάνω κώδικας απαιτεί λιγότερες πράξεις και μνήμη σε σύγκριση με τον τελευταίο αλγόριθμο του προηγούμενου ερωτήματος, ενώ παράλληλα είναι πολύ πιο απλός στην υλοποίησή του.

Ωστόσο επειδή το φ είναι άρρητος για αρκετά μεγάλο n ο παραπάνω αλγόριθμος μπορεί να επιστρέψει λάθος αποτέλεσμα. Αναλυτικότερα για $n > 70$ τότε αρχίζει ο αλγόριθμός μας να αποφαίνεται εσφαλμένα συμπεράσματα.

Σχόλιο: Για n λίγο μεγαλύτερο του 70 τα σφάλματα είναι σχεδόν αμελητέα, ωστόσο όσο αυξάνεται το n τόσο μεγαλύτερο γίνεται και το σφάλμα.

(γ)

```
def last_digits(n, k):
    # Find the fibonacci number using the algorithm with the matrices
    number = fib(n)
    # Initialize an array which is going to store the last k digits
    digits = [0 for i in range(k)]
    # Store the last k digits in the digits array
    for i in range(k):
        digits[i] = number % 10
        number //= 10
    return digits


n = int(input())
k = int(input())  # Obviously k < n
print(f"The last k: {k} digits of the n: {n} fibonacci number are:")
# Print the digits array in reverse order
for i in range(k):
    print(last_digits(n, k)[k-i-1], end='')
```

(δ) Η μέθοδος Fast-Doubling βασίζεται στις αναδρομικές σχέσεις:

$$F_{2n+1} = F_{n+1}^2 + F_n^2 \qquad and \qquad F_{2n} = 2 \cdot F_{n+1} \cdot F_n - F_n^2$$

Η οποίες αναδρομικές σχέσεις προκύπτουν από τον αλγόριθμο με την χρήση πινάκων αντικαθιστώντας το $n \rightarrow 2n$

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} \overset{n \rightarrow 2n}{\Longleftrightarrow} \begin{bmatrix} F_{2n+1} \\ F_{2n} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Στην συνέχεια θα αξιοποιήσουμε το γεγονός πως ο πίνακας $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ γράφεται ως εξής:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

https://chunminchang.github.io/blog/post/matrix-difference-equation-for-fibonacci-sequence

Άρα

$$\begin{bmatrix} F_{2n+1} \\ F_{2n} \end{bmatrix} = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n+1}^2 + F_n^2 \\ F_n \cdot F_{n+1} + F_n \cdot F_{n-1} \end{bmatrix} \overset{F_{n-1} = F_{n+1} - F_n}{\Longleftrightarrow}$$

$$\begin{bmatrix} F_{2n+1} \\ F_{2n} \end{bmatrix} = \begin{bmatrix} F_{n+1}^2 + F_n^2 \\ 2F_n \cdot F_{n+1} - F_n^2 \end{bmatrix}$$

https://www.geeksforgeeks.org/fast-doubling-method-to-find-the-nth-fibonacci-number/

Εφαρμογή του αλγορίθμου σε κώδικα C++ μπορεί να βρεθεί εδώ:

https://chunminchang.github.io/blog/post/calculating-fibonacci-numbers-by-fast-doubling

Ο αλγόριθμος Fast-Doubling έχει ασυμπτωτική πολυπλοκότητα $O(logn)$, ωστόσο είναι λίγο πιο αποδοτικός σε σύγκριση με τον αλγόριθμο που χρησιμοποιεί πίνακες (συγκεκριμένα είναι πιο γρήγορος κατά μια σταθερά). Ουσιαστικά αυτό ισχύει καθώς στον αλγόριθμο Fast-Doubling παραλείπονται περιττές πράξεις που πραγματοποιούνται στον αλγόριθμο με τους πίνακες.

https://www.nayuki.io/page/fast-fibonacci-algorithms

Άσκηση 4. (Εύρεση ΜΚΔ)

Θεωρήστε τον παρακάτω αλγόριθμο για εύρεση ΜΚΔ που είναι γνωστός ως Binary GCD.

bgcd(a, b):

(* υποθέτουμε a, b > 0 *)

- Αν $a = b$ επίστρεψε a

- αν a, b άρτιοι επίστρεψε $2 \cdot bgcd(\frac{a}{2}, \frac{b}{2})$

- αν a είναι άρτιος και b περιττός επίστρεψε $bgcd(\frac{a}{2}, b)$, και αντίστοιχα αν b άρτιος και a περιττός

- αν a, b περιττοί επίστρεψε $bgcd\left(\min(a, b), \frac{|a-b|}{2}\right)$

(α) Αποδείξτε την ορθότητα του Binary GCD.

(β) Ποια είναι η πολυπλοκότητά του και γιατί;

(γ) Υλοποιήστε τον και συγκρίνετε την αποδοτικότητά του με αυτήν του Ευκλείδειου αλγόριθμου. Δοκιμάστε τους δύο αλγορίθμους με τουλάχιστον 10 ζεύγη πολύ μεγάλων αριθμών.

(α) Για να αποδείξουμε την ορθότητα του αλγορίθμου θα αποδείξουμε αρχικά πως όλα τα βήματα που πραγματοποιεί είναι ορθά.

- $\gcd(a, a) = a$, αυτό θεωρώ πως δεν χρειάζεται απόδειξη είναι αρκετά προφανές
- Αν a, b είναι άρτιοι τότε:

  $\gcd(a, b) = \gcd(2k, 2n) \xLeftrightarrow{\gcd(a\cdot m, b\cdot m) = m\cdot \gcd(a,b)} \gcd(a, b) = 2 \cdot \gcd(k, n) = \gcd\left(\frac{a}{2}, \frac{b}{2}\right)$

  Απόδειξη πως $\gcd(a \cdot m, b \cdot m) = m \cdot \gcd(a, b)$:

  Έστω, $c = \gcd(a, b)$ άρα το c διαιρεί το a και το b (Συμβολισμός: $c \mid a, b$). Επίσης, ισχύει πως μπορούμε να γράψουμε το c ως εξής $c = na + kb \; for \; some \; n, k \epsilon N$. Τώρα για c·m:

  $c \cdot m | a \cdot m, b \cdot m \;\; \& \;\; c \cdot m = n \cdot a \cdot m + k \cdot b \cdot m \Leftrightarrow c \cdot m = \gcd(a \cdot m, b \cdot m) \Leftrightarrow$
  $m \cdot \gcd(a, b) = \gcd(a \cdot m, b \cdot m)$
- Χωρίς βλάβη της γενικότητας έστω a άρτιος και b περιττός. Τότε, $\gcd(a, b) = \gcd\left(\frac{a}{2}, b\right)$.

  Απόδειξη:

  Έστω $a = 2\kappa$. Το α έχει παράγοντες το 2 και ότι παράγοντες έχει ο αριθμός κ. Το b είμαστε σίγουροι πως δεν έχει παράγοντα το 2, διότι είναι περιττός αριθμός. Επομένως, ο μέγιστος κοινός παράγοντας του α με το b είναι ο ίδιος με τον κοινό παράγοντα του k και του b. Δηλαδή: $\gcd(a, b) = \gcd(2k, b) = \gcd(k, b) = \gcd\left(\frac{a}{2}, b\right)$

- Αν a, b περιττοί αριθμοί τότε: $gcd(a, b) = \gcd\left(min(a, b), \frac{|a-b|}{2}\right)$
  Αρχικά, θα αποδείξουμε πως γενικά ισχύει: $gcd(a, b) = \gcd(min(a, b), |a - b|)$
  Χωρίς βλάβη της γενικότητας έστω c είναι κοινός παράγοντας των a, b $\mu\varepsilon \; a \geq b$.

Επομένως ισχύει $a = p \cdot c \; and \; b = q \cdot c, άρα \; a - b = p \cdot c - q \cdot c = (p - q)c$
Άρα c είναι ένας κοινός διαιρέτης των $a - b \; και \; b$.
Αντίστροφα, αν θεωρήσουμε πως c είναι κοινός παράγοντας των a-b και b τότε:
$a - b = r \cdot c, \qquad b = q \cdot c, \qquad άρα \;\; a = (a - b) + b = r \cdot c + r \cdot q = (r + q)c$
Συμπεραίνουμε πως τα ζευγάρια (a, b) και (a-b, b) έχουν κοινούς παράγοντες και συνεπώς έχουν κοινό μέγιστο κοινό διαιρέτη.
Δηλαδή: $\gcd(a, b) = \gcd(b, a - b)$
Αν τώρα δεν ξέρουμε εξαρχής την διάταξη των α,b τότε η παραπάνω ιδιότητα γράφεται ως εξής: $\gcd(a, b) = \gcd(min(a, b), |a - b|)$.
Στην περίπτωσή μας όμως γνωρίζουμε πως $min(a, b)$ είναι περιττός και $|a - b|$ είναι άρτιος (εφόσον a,b περιττοί). Από την προηγούμενη ιδιότητα που αποδείξαμε ισχύει:

$$\gcd(a, b) = \gcd(min(a, b), |a - b|) = \gcd\left(min(a, b), \frac{|a - b|}{2}\right)$$

Έχοντας αποδείξει πως κάθε βήμα του αλγόριθμού μας είναι ορθό μένει να αποδείξουμε πως για κάθε ζεύγος αριθμών a, b θα καταλήξουμε στο base case $a = b$.

Αρχικά, είναι προφανές πως ο αλγόριθμος μας στην χειρότερη περίπτωση θα καταλήξει στο $\gcd(1,1) = 1$, διότι σε κάθε βήμα του αλγορίθμου μειώνουμε τουλάχιστον μια μεταβλητή τουλάχιστον κατά παράγοντα 2. Άρα έχουμε δείξει πως ο αλγόριθμός μας σίγουρα θα καταλήξει στο base case. Επομένως, αν έχουμε το ζεύγος a, b με $\gcd(a, b) = d$ ο αλγόριθμός μας πρέπει να καταλήξει σε ορθό αποτέλεσμα καθώς αλλιώς είτε θα καταλήξει σε εσφαλμένο base case είτε θα φτάσει στη χειρότερη περίπτωση που σημαίνει τα a, b είναι μεταξύ τους πρώτοι. Και οι δύο αυτές περιπτώσεις οδηγούν σε άτοπο διότι έχουμε αποδείξει την ορθότητα κάθε βήματος του αλγορίθμου μας.

(β) Όσον αφορά την πολυπλοκότητα του αλγορίθμου έχουμε ως εξής:

Στην καλύτερη περίπτωση ο αλγόριθμός μας θα βρει τον μέγιστο κοινό διαιρέτη μέσα στο πρώτα βήματα. Στην χειρότερη περίπτωση ο αλγόριθμός μας απαιτεί $O(n)$ βήματα (όπου n ο αριθμός των bits του μεγαλύτερου αριθμού), καθώς σε κάθε βήμα μειώνουμε τουλάχιστον μια μεταβλητή κατά παράγοντα τουλάχιστον 2 (η διαίρεση με το 2). Σε κάθε βήμα λαμβάνει μέρος η ακέραια διαίρεση με 2, όπου στο δυαδικό σύστημα αντιστοιχεί στην αφαίρεση του τελευταίου ψηφίου (LSB) και στην μετατόπιση των υπόλοιπων bits κατά μία θέση δεξιά (δηλαδή οι συντελεστές των bits αντιστοιχούν σε μικρότερη δύναμη). Αυτή η διαδικασία που πραγματοποιείται σε κάθε βήμα, όταν το n είναι μικρό είναι της τάξης $O(1)$. Ωστόσο, καθώς το n αυξάνεται η διαδικασία αυτή καταλαμβάνει γραμμικό χρόνο με αποτέλεσμα η χρονική πολυπλοκότητα του Binary GCD να είναι ασυμπτωτικά $O(n * n) = O(n^2)$.

Αν θέλαμε να γράφαμε την πολυπλοκότητα συναρτήσει των παραμέτρων α, b ,των οποίων υπολογίζουμε το gcd, θα λέγαμε πως για μικρά n έχουμε πολυπλοκότητα $O(\log(\max(a, b)))$, ενώ ασυμπτωτικά θα λέγαμε πως έχουμε πολυπλοκότητα $O(\log(\max(a, b))^2)$.

Αυτό ισχύει καθώς το $\log(\max(a, b))$ δηλώνει τον αριθμό τον ψηφίων (bits) που έχει ο μεγαλύτερος από τους δύο αριθμούς.

(γ)

```
# Importing this library will allow us to keep track of the time each
algorithm takes to finish
import time
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

def bgcd(a, b):
    # Base Case
    if (a == b):
        return a
# For division by 2 and checking if a number is ever or odd we are
# Utilizing bitwise operators
    # Checking if a is even
    if ((~a & 1) == 1):

        # b is odd
        if ((b & 1) == 1):
            return bgcd(a >> 1, b)
        else:
            # both a and b are even
            return (bgcd(a >> 1, b >> 1) << 1)

    # a is odd, b is even
    if ((~b & 1) == 1):
        return bgcd(a, b >> 1)

    # reduce larger number cause both a and b are odd
    return bgcd((abs(b - a)) >> 1, min(a, b))

# Creating pairs of numbers (a,b)
a = [
    820934399349376992926967104307914103890024215543523645673671234234523475892374399349376992926967104307914103589127589373465893725,
    562916588322647614399349376992926967104307914103711870890813471854334903459097991879759738495789081347185433490,
    3253146810513958538523498244439934937699292696710430791410368197424399349376992926968157493209318908134718543349079181130173523,
    2916588322647617118708513958538523689081347185433490291658839349376823475827367153454,
    23865920563095210205175164399349376992926967104307914103578889081347185433490092856179674699846671125434534534555234,
```

```
    8606542644399349376992926967104307914103557890813471439934937699292696
7104307914103854334908676888696564911878827855234645 6,
    1096302467601101819301916956475069858640814399349376992926967104307914
103495445686786987001234093963,
    6195700272232774753404040804399349376992926967104307943993493769929269
6710430791410314103728547011604399349376992926967104307914103121,
    9910210897353984752398791827348743552346443993493769929269671043079141
0375644253246534253265 71,
    3452345248782346598304563482741238439934937699292694399349376992926967
1043079141034399349376992926967104307914103657769 1,
]
b = [
    5883226439934937699292696710430791410349376992926967104307914103439934
937699292696710430791410308193 50,
    2093439934937699292696712345723894755643586571234123456457562456534 64,
    1051395853852349824468197420481934399349376992926967123457238947557493
22314265464786787901 46,
    1234712374098127340981278234475924538657394934399349376992926967112782
34475922345723894758759381275983465 98,
    1236498176509363456734569743234523456345236934399349376992926967123457
2389475934399349376992926967123457238947545756723545651,
    7634568790013457236134000745745600534563245893439934937699292696712345
72389475739847555645673412345645 6,
    5869586309869048509568750968456781654848421254841248693439934937699292
69671234572389475,
    6623954783475908234758913275934399349376992926967123457238947509451246
7321374861261124648934399349376992926967181 83,
    2123759834759283778264528934399349376992926967123457238947593452346586
28349572957201493439934937699292696712345723894034 5,
    1000010000000000000000000000000000000000000000000000000000000000000000,
]

for i in range(10):
    binary_start = time.time()
    binary_gcd = bgcd(a[i], b[i])
    binary_end = time.time()
    euclid_start = time.time()
    euclid_gcd = gcd(a[i], b[i])
    euclid_end = time.time()
    print("Binary gcd algorithm result:", binary_gcd,
          "|time:", binary_end-binary_start, end=' || ')
    print("Euclid's gcd algorithm result:", euclid_gcd,
          "|time:", euclid_end-euclid_start)
```

```
PS C:\Users\User\Desktop\Python> python3 main.py
Binary gcd algorithm result: 25 |time: 0.0009996891021728516 || Euclid's gcd algorithm result: 25 |time: 0.0
Binary gcd algorithm result: 6 |time: 0.0 || Euclid's gcd algorithm result: 6 |time: 0.0
Binary gcd algorithm result: 1 |time: 0.0010018348693847656 || Euclid's gcd algorithm result: 1 |time: 0.0
Binary gcd algorithm result: 2 |time: 0.0 || Euclid's gcd algorithm result: 2 |time: 0.0
Binary gcd algorithm result: 1 |time: 0.0010023117065429688 || Euclid's gcd algorithm result: 1 |time: 0.0
Binary gcd algorithm result: 8 |time: 0.000997304916381836 || Euclid's gcd algorithm result: 8 |time: 0.0
Binary gcd algorithm result: 1 |time: 0.0009996891021728516 || Euclid's gcd algorithm result: 1 |time: 0.0
Binary gcd algorithm result: 1 |time: 0.002000093460083008 || Euclid's gcd algorithm result: 1 |time: 0.0
Binary gcd algorithm result: 1 |time: 0.0010023117065429688 || Euclid's gcd algorithm result: 1 |time: 0.0
Binary gcd algorithm result: 11 |time: 0.0 || Euclid's gcd algorithm result: 11 |time: 0.0
```

Παρατηρούμε, πως για αρκετά μεγάλα ζευγάρια αριθμών ο binary gcd είναι λιγότερο αποδοτικός από τον ευκλείδειο αλγόριθμο όπως και περιμέναμε, καθώς η πράξη της διαίρεσης (το shift που γίνεται) όταν παίζουμε με μεγάλους αριθμούς είναι σχετικά χρονοβόρα.

Άσκηση 5. (Σχεδόν Δέντρο)

Έστω συνεκτικό μη κατευθυνόμενο γράφημα G(V, E, w) με θετικά βάρη w στις ακμές. Υποθέτουμε ότι το G είναι σχεδόν δέντρο, με την έννοια ότι |E| = |V| + c, για κάποια σταθερά c. Να διατυπώσετε αποδοτικό αλγόριθμο (κατά προτίμηση γραμμικού χρόνου) για τον υπολογισμό ενός ελάχιστου συνδετικού δέντρου σε ένα τέτοιο γράφημα G. Να αιτιολογήσετε την ορθότητα και την υπολογιστική πολυπλοκότητα του αλγορίθμου σας.

Για την επίλυση της άσκησης θα χρησιμοποιήσουμε τον αλγόριθμο του Kruskal.

```
Tree := {};

for each vertex v in our Graph G
{
    Create_Set(v);
}
for each edge (u,v)∈ E sorted by weight in ascending order
{
    // If there is no cycle
    if(find(u) != find(v))
    {
        Tree := Tree U {(u,v)};
        union(u,v);
    }
}

return Tree;
```

Θα αποδείξουμε την ορθότητα του παραπάνω αλγορίθμου αποδεικνύοντας το παρακάτω θεώρημα.

Θεώρημα: Ένα σύνολο ακμών A που είναι υποσχόμενο (δηλαδή υποσύνολο ενός MST) παραμένει υποσχόμενο αν του προσθέσουμε την ελαφρύτερη ακμή $e = (u, v)$ που συνδέει μια συνεκτική συνιστώσα V_i του τρέχοντος υπογράφου (που ορίζεται από τους κόμβους του V και τις ακμές του A) με τον υπόλοιπο υπογράφο $V - V_i$.

Απόδειξη: Θεωρούμε T ένα MST που είναι υπερσύνολο του A (στην συγκεκριμένη άσκηση πάντα θα υπάρχει τουλάχιστον ένα MST καθώς ο γράφος είναι συνεκτικός). Έστω πως $e \notin T$. Ωστόσο, στο ελάχιστα συνδετικό δέντρο πρέπει να υπάρχει μονοπάτι από το u στο v τέτοιο ώστε να περιέχει μια ακμή e_1 που διασχίζει την τομή $(V_i, V - V_i)$. Ισχύει πως $cost(e) \leq cost(e_1)$. Επομένως, αν στο μονοπάτι αυτό ανταλλάξουμε την ακμή e_1 με την e τότε προκύπτει ένα διαφορετικό ελάχιστο συνδετικό δέντρο T'.

Συνεπώς, ο αλγόριθμος Kruskal είναι ορθός, διότι σε κάθε βήμα του for loop προσθέτουμε στο υποσχόμενο σύνολο ακμών μας την ακμή με το λιγότερο κόστος που συνδέει μια συνεκτική συνιστώσα του τρέχοντος υπογράφου με τον υπόλοιπο υπογράφο.

Για τον υπολογισμό της πολυπλοκότητας του αλγορίθμου πρέπει πρώτα να λάβουμε υπόψη τρεις παράγοντες που καθορίζουν την πολυπλοκότητα του αλγορίθμου μας.

- Η δημιουργία συνόλων
- Η ταξινόμηση των ακμών
- Οι συναρτήσεις union και find

Καταρχάς, θα χρησιμοποιήσουμε την δομή δεδομένων union-find, με Union-by-Rank + Path Compression αποσκοπώντας να βελτιστοποιήσουμε τις συναρτήσεις union και find.

Η δημιουργία συνόλων κοστίζει O(|V|), καθώς δημιουργούμε |V| σύνολα σε σταθερό χρόνο. Όμως επειδή |E| = |V| + c, μπορούμε να ξαναγράψουμε το κόστος ως O(|E|)

Όσον αφορά την ταξινόμηση των ακμών χρησιμοποιούμε κάποιον γνωστό και γρήγορο αλγόριθμο ταξινόμησης (έστω Merge Sort) ο οποίος κοστίζει O(|E|log|E|)

Χρησιμοποιώντας την δομή δεδομένων union-find, με Union-by-Rank + Path Compression οι συναρτήσεις union και find κοστίζουν O(1) και O(log|E|) αντίστοιχα. Ωστόσο, το for loop πραγματοποιεί |E| επαναλήψεις. Άρα έχει κόστος O(|E|log|E|).

Επομένως η τελική πολυπλοκότητα του αλγορίθμου είναι η εξής:

$$O(|E| + |E|\log|E| + |E|\log|E|) = O(|E|\log|E|) \xrightarrow{|E|=|V|+c} O(|V|\log|V|)$$

Άσκηση 6. (r-περιορισμένο μονοπάτι)

Έστω μη κατευθυνόμενο γράφημα G(V, E, w) με θετικά βάρη w στις ακμές, και έστω s, t ∈ V . Για κάποιο r > 0, λέμε ότι ένα s–t μονοπάτι p είναι r-περιορισμένο αν το βάρος κάθε ακμής στο p είναι μικρότερο ή ίσο του r.

1. Να διατυπώσετε αποδοτικό αλγόριθμο που για δεδομένο r, ελέγχει αν υπάρχει r-περιορισμένο s-t μονοπάτι στο G.

2. Να δείξετε ότι το G περιέχει r-περιορισμένο s-t μονοπάτι αν και μόνο αν ένα ελάχιστο Συνδετικό Δέντρο του G περιέχει r-περιορισμένο s-t μονοπάτι.

3. Να διατυπώσετε αποδοτικό αλγόριθμο που υπολογίζει την ελάχιστη τιμή του r για την οποία υπάρχει r-περιορισμένο μονοπάτι s-t στο G.

1. Ο παρακάτω αλγόριθμος αποτελεί εφαρμογή του BFS με λίγες παραλλαγές.

Αναλυτικότερα, με κόμβο εκκίνησης τον s διατρέχουμε έναν BFS με την προϋπόθεση πως στην ουρά προσθέτουμε γειτονικούς κόμβους μόνο όταν το βάρος της ακμής ανάμεσά τους είναι μικρότερο ή ίσο του r.

```
Let s be our initial vertex;
bool BFS (G, s)
{
    let Q be our queue;
    Q.enqueue(s);
    mark s as visited;
    while (!Q.empty())
    {
        v = Q.dequeue();
        for all neighbors w of v in Graph G
        {
            if (w is not visited and the weight of the edge (w,v)<=r)
            {
                if (w == t)
                return true;
                Q.enqueue(w);
            }
            mark w as visited;
        }
    }

    return false;
}
```

Ο παραπάνω αλγόριθμος έχει την ίδια πολυπλοκότητα με τον αλγόριθμο BFS δηλαδή $O(|V| + |E|)$, καθώς το μόνο που προσθέσαμε είναι ένα $if\ statement$ που είναι της τάξης $O(1)$.

2.

Έστω πως υπάρχει MST που περιέχει μονοπάτι p r-περιορισμένο μεταξύ των κόμβων s-t. Το μονοπάτι αυτό του MST αποτελεί υποσύνολο των μονοπατιών μεταξύ των κόμβων s-t στον γράφο G. Επομένως, το μονοπάτι p υπάρχει στο γράφο και είναι r-περιορισμένο.

Αντιστρόφως, έστω πως ο γράφος μας G περιέχει r-περιορισμένο μονοπάτι p μεταξύ των κόμβων s-t. Για να υπάρχει μονοπάτι που συνδέει αυτούς τους κόμβους τότε πρέπει να υπάρχει και MST που συνδέει αυτούς τους δυο κόμβους. Έστω, πως δεν υπάρχει MST στο γράφο μας που να συνδέει τους κόμβους s-t με r-περιορισμένο κομμάτι. Ωστόσο, ένα MST διασυνδέει όλους τους κόμβους του γράφου με το μικρότερο δυνατόν βάρος. Επομένως, εάν υπάρχει μονοπάτι όπου συνδέει τους κόμβους s-t με ενδιάμεσες ακμές που δεν ξεπερνούν το r, τότε θα υπάρχει MST που να το περιέχει, με αποτέλεσμα να καταλήγουμε σε άτοπο.

Συνεπώς, το G περιέχει r-περιορισμένο s-t μονοπάτι αν και μόνο αν ένα ελάχιστο Συνδετικό Δέντρο του G περιέχει r-περιορισμένο s-t μονοπάτι.

Άσκηση 7. (Μονοπάτι με ελάχιστο πλήθος ακμών)

Θεωρούμε κατευθυνόμενο γράφημα G(V, E, w) με n κορυφές, m ακμές και θετικά μήκη w στις ακμές, και μια αρχική κορυφή s του G. Να διατυπώσετε αποδοτικό αλγόριθμο που για όλες τις κορυφές u ∈ V , υπολογίζει ένα συντομότερο s − u μονοπάτι με ελάχιστο πλήθος ακμών (μεταξύ όλων των συντομότερων s−u μονοπατιών). Ποια η υπολογιστική πολυπλοκότητα του αλγορίθμου σας;

```
// Visited vertices with s being the initial vertex
S := {s};

// D(u) gives us the shortest distance we have found so far between the
// vertices s and u
D(s) := 0;

// Previous(u)
P(s) := NIL;

// E(u) gives us the least amount of edges we have found so far
// between the vertices s and u
E(s) := 0;
```

```
for each v:(s,v)∈ E do
{
    D(v) := cost(s,v);
    E(v) := 1;
    P(v) := s;
}
for each v:(s,v)∉ E do
{
    D(v) := ∞;
    E(v) := ∞;
    P(v) := NIL;
}
repeat n times
{
    select u from V\S with minimum D(u);
    S := S U {u};
    for all v in V\S: (u,v) E do
    {
        if (D(u) + cost(u,v) < D(v)) then
        {
            D(v) := D(u) + cost(u,v);
            P(v) := u;
            E(v) := E(u) + 1;
        }
        else if(E(u) + 1 < E(v) and D(u) + cost(u,v) == D(v)) then
        {
            E(v) := E(u) + 1;
            P(v) := u;
        }
    }
}
```

Επεξήγηση: Ο παραπάνω αλγόριθμος αποτελεί εφαρμογή του αλγόριθμου Dijkstra με μια μικρή προσθήκη. Συγκεκριμένα, ορίζουμε τον πίνακα E(u) με τον οποίο καταγράφουμε τον ελάχιστο πλήθος ακμών που έχει το συντομότερο μονοπάτι ανάμεσα στους κόμβους s και u.

Άσκηση 8. (Επιβεβαίωση Αποστάσεων)

Θεωρούμε ένα κατευθυνόμενο γράφημα G(V, E, ℓ) με n κορυφές, m ακμές και (ενδεχομένως αρνητικό) 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 διαδρομής από την u στην v) στο G. Δίνονται n αριθμοί $\delta_1, \ldots, \delta_n$, όπου κάθε δ_k (υποτίθεται ότι) ισούται με την απόσταση $d(v_1, v_k)$ στο G. Να διατυπώσετε αλγόριθμο που σε χρόνο $\Theta(n + m)$, δηλ. γραμμικό στο μέγεθος του γραφήματος, ελέγχει αν τα $\delta_1, \ldots, \delta_n$ πράγματι ανταποκρίνονται στις αποστάσεις των κορυφών από την v_1, δηλαδή αν για κάθε v_k ∈ 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 $\Theta(n + m)$)

Έστω πως $\delta_k \equiv d(v_1, v_k), \forall k \in \{1, \ldots, n\}$. Τότε, για κάθε ακμή του γράφου $e = (u, v)$ πρέπει να ισχύει η ανισότητα $\delta_v \leq \delta_u + cost(u, v)$. Όταν ισχύει η ισότητα τότε στο Δέντρο Συντομότερων Μονοπατιών ο κόμβος v είναι παιδί του κόμβου u.

Επομένως, εάν δεν ισχύει η ανισότητα για κάθε ακμή του γράφου μας τότε η ακολουθία δ_n που μας δόθηκε είναι ψευδής και επιστρέφουμε false.

Πρώτου ελέγξουμε αν η παραπάνω ανισότητα ισχύει για κάθε ακμή αρχικοποιούμε κάθε θέση του πίνακα Previous με το u1 όπου αποτελεί και την καλύτερη περίπτωση.

```
// Initializing the previous array
for i in range(n)
        P(i) := u1;

// Checking the conditions
for each edge(u, v) in G
    {
        if (dv = du + cost(u, v))
            P(v) := u;
        else if (dv > du + cost(u, v))
            return false;
    }
return previous;
```

Όσον αφορά την πολυπλοκότητα του παραπάνω αλγορίθμου είναι $\Theta(n + m)$, διότι αρχικοποιούμε τον πίνακα Previous το οποίο κοστίζει $\Theta(n)$ και ελέγχουμε για κάθε ακμή αν ισχύει η ανισότητα, το οποίο κοστίζει $\Theta(m)$.