ενώ ο αναδρομικός του τύπος είναι ο :

$$H(n) = 2*H(n-1) + 1$$

$$H(1) = 1$$

Με την επίλυση του πρώτου σκέλους του αναδρομικού τύπου έχουμε:

$H(n) = 2*H(n-1) + 1 = 2^2*H(n-2) + 2 + 1 = \dots = 2^{n-1}*H(1) + 2^{n-2} + \dots + 2 + 1 = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0 = \sum_{k=1}^{n} 2^k$ =>$H(n) = 2^n - 1$.

Β) Για να κατασκευάσουμε τον επαναληπτικό αλγόριθμο του προβλήματος των πύργων του Hanoi αρκεί να παρατηρήσουμε το μοτίβο του παραπάνω αλγορίθμου. Κατ' επανάληψη, κάθε φορά που θέλουμε να μεταφέρουμε μια στοίβα/υπο-στοίβα από μια ράβδο σε μια άλλη, μετακινούμε τον δίσκο #1 δεξιά και στη συνέχεια μετακινούμε τον δίσκο που είναι δυνατόν να μετακινηθεί (προφανώς όχι τον δίσκο #1, αλλιώς θα δημιουργούταν loop).

Έστω ότι διαθέτουμε τις 3 ράβδους του παραδείγματος **(Α)**, Α, Β και C και έστω ότι θέλουμε να μετακινήσουμε τον πύργο Hanoi από το Α στο C. Θα πρέπει αρχικά να ορίσουμε μια θετική φορά για τον επαναληπτικό αλγόριθμο, έστω A-C-B-A. Υπάρχει το ενδεχόμενο ο αλγόριθμος να μετακινήσει τη στοίβα στη ράβδο Β, κάτι που είναι προφανώς λάθος και μας καταδεικνύει ότι η σωστή θετική φορά θα ήταν A-B-C-A. Συνεπώς ο αριθμός των κινήσεων παραμένει ίδιος με το ερώτημα **(Α)**. Για να μπορούμε εξ αρχής να επιλέγουμε σωστή θετική φορά, θα πρέπει το πρόγραμμά μας να ελέγχει αν ο αριθμός των δίσκων n είναι άρτιος ή περιττός και να υιοθετεί ως θετική φορά την A-C-B-A ή A-B-C-A αντίστοιχα.

Στη συνέχεια θα αποδείξουμε ότι, και πάλι, $H(n) = 2^n - 1$. Έστω ότι για οποιοδήποτε δοθέν n έχουμε επιλέξει τη σωστή θετική φορά. Για να ισχύει ο τύπος για κάθε n ακέραιο, θα πρέπει να ισχύει και για n+1. Σε αυτή την περίπτωση, για $2^n - 1$ κινήσεις έχουμε δημιουργήσει μια στοίβα n δίσκων στο empty_rod και απαιτείται ακόμη μια κίνηση για να μετακινηθεί ο n-οστός δίσκος στο destination_rod. Έπειτα, απαιτούνται άλλες $2^n - 1$ κινήσεις, ώστε η στοίβα n δίσκων να τοποθετηθεί πάνω στον n-οστό δίσκο. Σύνολο, δηλαδή, οι κινήσεις που πραγματοποιούνται είναι:

$H(n+1) = 2^n - 1 + 1 + 2^n - 1 = 2*2^n - 1 = 2^{n+1} - 1$, κάτι που επιβεβαιώνει την αρχική μας υπόθεση.

Γ) Σε αυτό το σημείο θα αποδείξουμε ότι για την επίλυση του προβλήματος των πύργων του Hanoi απαιτούνται τουλάχιστον H(n) βήματα, όπως έχουν οριστεί παραπάνω. Είδαμε ότι για να μετακινήσουμε τον πύργο Hanoi από το origin_rod στο destination_rod, μετακινούνται οι top n-1 δίσκοι στο empty_rod, ο n-οστός δίσκος στο destination_rod και, τέλος, οι n-1 δίσκοι στο destination_rod. Επαγωγικά, αυτό σημαίνει ότι :

$$H(n) = 2*H(n-1) + 1 \text{ για κάθε ακέραιο } n$$

Γνωρίζουμε, όμως, ότι H(1) = 1 και H(0) = 0 (είναι προφανή). Άντικαθιστώντας, λοιπόν, στην παραπάνω σχέση n = 1, έχουμε

$$H(1) = 2*H(0) + 1 \text{ -> } 1 = 0 + 1, \text{ το οποίο ισχύει.}$$

Είναι, λοιπόν, σωστό να θεωρήσουμε πως $H(n) \geq 2*H(n-1) + 1$ για κάθε ακέραιο n, με την ισότητα να ισχύει για n = 1. Συνεπώς, οι ελάχιστες κινήσεις για την επίλυση του προβλήματος των πύργων του Hanoi ισούνται με $H(n) = 2^{n-1} + 1$, άρα τόσο ο αναδρομικός αλγόριθμος του ερωτήματος **(Α)**, όσο και ο επαναληπτικός αλγόριθμος του ερωτήματος **(Β)** είναι ελάχιστοι.

Δ) Έστω ότι διαθέτουμε το πρόβλημα των πύργων του Hanoi, αλλά αυτή τη φορά με 4 ράβδους έναντι 3. Μπορούμε, αντίστοιχα με το ερώτημα **(Α)** να ονομάσουμε τις ράβδους origin_rod, empty_rod1, empty_rod2 και destination_rod. Όμοια με το ερώτημα **(Α)**, στον αλγόριθμο αυτό θα μετακινούμε πρώτα τους n-2 δίσκος στο empty_rod1, τον (n-1)-οστό δίσκο στο empty_rod2, τον n-οστό δίσκο στο destination_rod. Έπειτα, ακολουθώντας την αντίστροφη διαδικασία, μετακινούμε τον (n-1)-οστό δίσκο στο destination_rod κι έπειτα τους n-2 δίσκους στο destination_rod. Ο αλγόριθμος που προκύπτει σε c++ για τον παραπάνω αλγόριθμο είναι ο εξής:

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

void TowerOfHanoi_4rods(int n, char origin_rod, char empty_rod1, char empty_rod2, char destination_rod) {
    if (n == 0) return;
    if (n == 1) {
        printf("\nΜετακινούμε τον δίσκο 1 από τη ράβδο %c στη ράβδο %c.", origin_rod, destination_rod);
        return;
    }
    TowerOfHanoi_4rods(n - 2, origin_rod, destination_rod, empty_rod2, empty_rod1);
    printf("\nΜετακινούμε τον δίσκο %d από τη ράβδο %c στη ράβδο %c.", n - 1, origin_rod, empty_rod2);
    printf("\nΜετακινούμε τον δίσκο %d από τη ράβδο %c στη ράβδο %c.", n, origin_rod, destination_rod);
    TowerOfHanoi_4rods(n - 2, empty_rod1, origin_rod, empty_rod2, destination_rod);
}

int main() {
    int A, B, C, D, n;
    cin >> n;
    TowerOfHanoi_4rods(n, A, B, C, D);
}
```

Η αναδρομική σχέση του αλγορίθμου είναι:

$$H(n) = 2*H(n-2) + 3 \text{ και, προφανώς, } H(1) = 1, H(0) = 0.$$

Επιλύοντας τον τύπο έχουμε ότι:

$$H(n) = 2*H(n-2) + 3 = 2^2*H(n-4) + 2*3 + 3 = 2^3*H(n-6) + 2^2*3 + 2^1*3 + 2^0*3 = \dots = 2^k*H(n-2k) + 3*[2^k + 2^{k-1} + \dots + 2^0] = 2^k*H(n-2k) + 3*\sum_{i=0}^{k} 2^i$$

Άσκηση 2.

Α) Θα προσπαθήσουμε να σχεδιάσουμε το ζητούμενο πρόγραμμα στη γλώσσα c++. Το πρόγραμμα που προκύπτει είναι το παρακάτω:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    if ((n == 1) || (n == 2)) {
        printf("\nΟ αριθμός %d είναι πρώτος κατά Fermat.", n);
        return 0;
    }
    for (int a = 2; a < (n-1); a++) {
        if (pow(a, n-1) != 1) {
            printf("\nΟ αριθμός %d δεν είναι πρώτος κατά Fermat.", n);
            return 0;
        }
    }
    printf("\nΟ αριθμός %d είναι πρώτος κατά Fermat.", n);
}
```

Ο αλγόριθμος αυτός, ωστόσο, είναι πολύ “αργός”. Προσπαθώντας να σχεδιάσουμε το βέλτιστο πρόγραμμα, καταλήγουμε στο εξής:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

bool Prime_Number(unsigned int n) {
    int a = n-2;
    while (a > 1) {
        if ( fmod(pow(a, n - 1), n) != 1) return false;
        --a;
    }
    return true;
}

int main() {
    int n;
    cin >> n;
    if ( (n <= 1) || (n == 4) ) {
        printf("\nO αριθμός %d δεν είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.", n);
        return 0;
    }
    else if ( (n > 1) && (n <= 3) ) {
        printf("\nO αριθμός %d είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.", n);
        return 0;
    }
    unsigned int k = n;
    if (Prime_Number(k) == true) printf("\nO αριθμός %d είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.", n);
    else printf("\nO αριθμός %d δεν είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.", n);
    return 0;
}
```

Αν και πιο γρήγορο, παρατηρούμε ότι το πρόγραμμα αυτό δεν ανταποκρίνεται στα ζητούμενα του προβλήματος, καθώς υπάρχει υπερχείλιση για αριθμούς με εκατοντάδες ψηφίων. Θα γράψουμε, λοιπόν, ένα παρόμοιο πρόγραμμα στη γλώσσα Python 3.

```python
def Prime_Numbers() :
    n = input("Παρακαλώ πληκτρολογήστε τον αριθμό που θέλετε να ελέγξετε.\n")
    n = int(n)
    if ( (n <= 1) or (n == 4) ) :
        print(f'O αριθμός {n} δεν είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.')
        return False

    if ( (n > 1) and (n <= 3) ) :
        print(f'O αριθμός {n} είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.')
        return True

    else :
        a = n - 2
        k = 30
        while (k >= 1) :
            if ( pow(a, n-1, n) != 1) :
                print(f'O αριθμός {n} δεν είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.')
                return False
            a -= 1
            k -= 1

        print(f'O αριθμός {n} είναι πρώτος σύμφωνα με τον έλεγχο του Fermat.')
        return True


Prime_Numbers()
```

Ο αλγόριθμος αυτός είναι αποδοτικός και λειτουργεί ακόμη και για αριθμούς με εκατοντάδες ψηφίων.

Εισάγοντας αριθμούς Carmichael στο πρόγραμμα, παρατηρούμε ότι θεωρούνται πρώτοι σύμφωνα με τον έλεγχο Fermat, παρόλο που κάτι τέτοιο δεν ισχύει. Αυτό είναι αναμενόμενο, γι΄ αυτό άλλωστε και οι αριθμοί Carmichael ονομάζονται και ψευδο-πρώτοι (pseudoprime).

Β)Σύμφωνα με τον αλγόριθμο Miller-Rabin, κάθε πρώτος αριθμός n (n > 2) μπορεί να γραφτεί ως εξής:

$n = 2^s*d + 1$ δηλαδή $n – 1 = 2^s*d$

Στη συνέχεια επιλέγουμε έναν ακέραιο a, τέτοιο ώστε 0 < a < n. Αν ισχύει πως

- $a^d = 1 \bmod n$ ή πως
- $a^{2^r * d} = -1 \bmod n$, για κάποιο r ∈ [0, s),

τότε το n είναι ισχυρά πρώτος αριθμός με βάση το a. Αν κανένας από τους δύο παραπάνω ισχυρισμούς δεν ισχύει, τότε το n δεν είναι πρώτος αριθμός.

Θα υλοποιήσουμε τον παραπάνω αλγόριθμο σε Python 3.

```python
from random import seed
from random import randint

def MillerRabin(n, d, s, a):
    b = pow(a, d, n)
    k = 1
    if ( (b == 1) or (b == n-1) ):
        return True
    while (k < s):
        b = (b**2)%n
        if (b == n-1):
            return True
        k += 1
    return False

def main():
    n = input("Παρακαλώ πληκτρολογήστε τον αριθμό που θέλετε να ελέγξετε.\n")
    n = int(n)
    c = 30
    if ( (n <= 1) or (n == 4) ) :
        print(f'Ο αριθμός {n} δεν είναι πρώτος σύμφωνα με τον έλεγχο του Miller Rabin.')
        return False

    if ( (n > 1) and (n <= 3) ) :
        print(f'Ο αριθμός {n} είναι πρώτος σύμφωνα με τον έλεγχο του Miller Rabin.')
        return True

    d = n - 1
    s = 0
    while (d % 2 == 0):
        d = d/2
        s += 1
    d = int(d)
    s = int(s)
    while (c >= 1):
        a = randint(1, n-1)
        a = int(a)
        if (MillerRabin(n, d, s, a) == True):
            print(f'Ο αριθμός {n} είναι πρώτος σύμφωνα με τον έλεγχο του Miller Rabin.')
            return True
        else:
            print(f'Ο αριθμός {n} δεν είναι πρώτος σύμφωνα με τον έλεγχο του Miller Rabin.')
            return False

main()
```

Το πρόγραμμα αυτό αποδίδει σωστό αποτέλεσμα για όλους τους αριθμούς που δέχεται, ακόμα και τους pseudoprime αριθμούς. Ο αλγόριθμος, ωστόσο, είναι λιγότερο αποδοτικός από τον αλγόριθμο του ερωτήματος **(Α)**.

Γ) Για να εντοπίσουμε τους αριθμούς Mersenne, θα λάβουμε πρωτίστως υπόψη μας την παρακάτω ιδιότητα:

Αν το x δεν είναι πρώτος αριθμός, τότε ούτε και το $2^x - 1$ δεν είναι πρώτος αριθμός.

Για τον σκοπό αυτό, θα αξιοποιήσουμε το πρόγραμμα του ερωτήματος **(Α)**, εφόσον από εκφώνηση 100 < x < 1000. Σε πρώτο στάδιο θα ελέγχουμε αν το x είναι πρώτος αριθμός, κι αν αυτό ισχύει τότε θα τυπώνουμε τον αριθμό $2^x - 1$. Το πρόγραμμα που προκύπτει φαίνεται παρακάτω:

```python
def Prime_Numbers(n) :
    if ( (n <= 1) or (n == 4) ) :
        return False

    if ( (n > 1) and (n <= 3) ) :
        return True

    else :
        a = n - 2
        k = 30
        while (k >= 1) :
            if ( pow(a, n-1, n) != 1) :
                return False
            a -= 1
            k -= 1

        return True

def Mersenne():
    x = 101
    while (x < 1000):
        if (Prime_Numbers(x) == True):
            print(pow(2,x) - 1)
        x += 1

Mersenne()
```

Ετσι, εκτυπώνονται όλοι οι πρώτοι αριθμοί Mersenne, 2^x-1, με το x ∈ (100, 1000).

Άσκηση 3.

Α)Ο αλγόριθμος για τον εντοπισμό του n-οστού αριθμού Fibonacci με memoization είναι ο εξής:

```python
def Fibonacci_memoization(n):
    if (n == 0):
        return False
    elif (n == 1):
        return 0
    elif (n == 2):
        return 1
    elif (n > 2):
        return Fibonacci_memoization(n-1) + Fibonacci_memoization(n-2)

def main():
    n = input("Please give me an integer\n")
    n = int(n)
    while (n <= 0):
        n = input("Please give me another integer\n")
        n = int(n)
    if (n == 1):
        print('The first Fibonacci number is 0')
    elif (n == 2):
        print('The second Fibonacci number is 1')
    elif (n == 3):
        print('The third Fibonacci number is 1')
    else:
        print(f'The {n}-th Fibonacci number is {Fibonacci_memoization(n)}')

main()
```

Για την εύρεση του n-οστού αριθμού Fibonacci με επαναληπτικό αλγόριθμο, ο κώδικας φαίνεται παρακάτω:

```python
def Fibonacci_iteration(n):
    x = 0
    y = 1
    for i in range (2, n):
        z = y
        y = x + y
        x = z
    return y

def main():
    n = input("Please give me an integer\n")
    n = int(n)
    while (n <= 0):
        n = input("Please give me another integer\n")
        n = int(n)
    if (n == 1):
        print('The first Fibonacci number is 0')
    elif (n == 2):
        print('The second Fibonacci number is 1')
    elif (n == 3):
        print('The third Fibonacci number is 1')
    else:
        print(f'The {n}-th Fibonacci number is {Fibonacci_iteration(n)}')

main()
```

ενώ ο αλγόριθμος με πίνακα είναι ο εξής:

```python
import numpy as np

def Fibonacci_matrix(n):
    result = np.zeros((n, n), dtype=int)
    result[0][0] = 0
    result[0][1] = 1
    a, b = 0, 1
    for i in range(2, n):
        c = a + b
        result[0][i] = c
        a = b
        b = c
    return result[0][n-1]

def main():
    n = input("Please give me an integer\n")
    n = int(n)
    while (n <= 0):
        n = input("Please give me another integer\n")
        n = int(n)
    if (n == 1):
        print('The first Fibonacci number is 0')
    elif (n == 2):
        print('The second Fibonacci number is 1')
    elif (n == 3):
        print('The third Fibonacci number is 1')
    else:
        print(f'The {n}-th Fibonacci number is {Fibonacci_matrix(n)}')

main()
```

Θα κάνουμε merge και τους 3 αλγορίθμους με σκοπό να τους συγκρίνουμε, εισάγοντας τον παράγοντα του χρόνου.

```python
import numpy as np
import time

def Fibonacci_matrix(n):
    result = np.zeros((n, n), dtype=int)
    result[0][0] = 0
    result[0][1] = 1
    a, b = 0, 1
    for i in range(2, n):
        c = a + b
        result[0][i] = c
        a = b
        b = c
    return result[0][n-1]

def Fibonacci_memoization(n):
    if (n == 0):
        return False
    elif (n == 1):
        return 0
    elif (n == 2):
        return 1
    elif (n > 2):
        return Fibonacci_memoization(n-1) + Fibonacci_memoization(n-2)

def Fibonacci_iteration(n):
    x = 0
    y = 1
    for i in range (2, n):
        z = y
        y = x + y
        x = z
    return y

def main():
    n = input("Please give me an integer\n")
    n = int(n)
    while (n <= 0):
        n = input("Please give me another integer\n")
        n = int(n)
    if (n == 1):
        print('The first Fibonacci number is 0')
    elif (n == 2):
        print('The second Fibonacci number is 1')
    elif (n == 3):
        print('The third Fibonacci number is 1')
    else:
        time_a = time.time()
        print(f'Using the memoization method, the {n}-th Fibonacci number is {Fibonacci_memoization(n)}')
        time_b = time.time()
        print(f'The memoization method was {time_b - time_a} seconds long')
        print(f'Using the iteration method, the {n}-th Fibonacci number is {Fibonacci_iteration(n)}')
        time_c = time.time()
        print(f'The iteration method was {time_c - time_b} seconds long')
        print(f'Using the matrix method, the {n}-th Fibonacci number is {Fibonacci_matrix(n)}')
        time_d = time.time()
        print(f'The matrix method was {time_d - time_c} seconds long')

main()
```

```
Please give me an integer
5
Using the memoization method, the 5-th Fibonacci number is 3
The memoization method was 6.3180923461914 06e-05 seconds long
Using the iteration method, the 5-th Fibonacci number is 3
The iteration method was 8.130073547363281e-05 seconds long
Using the matrix method, the 5-th Fibonacci number is 3
The matrix method was 0.00012469291687011 72 seconds long


...Program finished with exit code 0
Press ENTER to exit console.
```

Παρατηρούμε πως ο ταχύτερος αλγόριθμος είναι αυτός που χρησιμοποιεί πίνακες, ο βραδύτερος είναι ο recursive με memoization, ενώ ανάμεσά τους βρίσκεται ο επαναληπτικός αλγόριθμος.

Β) Θα προσπαθήσουμε, σε αυτό το ερώτημα, να μεταγράψουμε τους παραπάνω αλγορίθμους κάνοντας χρήση της *χρυσής τομής*. Λαμβάνουμε την ακολουθία Fibonacci για του 13 πρώτους αριθμούς:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

Διαιρούμε κάθε αριθμό με τον προηγούμενό του (ξεκινώντας, προφανώς, από τον δεύτερο άσσο). Τα αποτελέσματα που λαμβάνουμε είναι τα εξής:

1/1 = 1, 2/1 = 2, 3/2 = 1.5, 5/3 = 1.666, 8/5 = 1.6, 13/8 = 1.625, 21/13 = 1.615, 34/21 = 1.619, 55/34 = 1.6176, 89/55 = 1.6181, 144/89 = 1.6179

Σχηματίζεται, έτσι, μια νέα ακολουθία αριθμών:

1, 2, 1.5, 1.66, 1.6, 1.625, 1.615, 1.619, 1.6176, 1.6181, 1.6179 …

η οποία συγκλίνει στο 1.618, δηλαδή στη χρυσή τομή φ.

Γνωρίζουμε πως φ = $\frac{1+\sqrt{5}}{2}$, πως ψ = -$\frac{1}{\varphi}$= $\frac{1-\sqrt{5}}{2}$και πως F_n = $\frac{\varphi^n-(-\varphi)^{-n}}{\sqrt{5}}$= $\frac{\varphi^n-\psi^n}{\sqrt{5}}$. Συνεπώς ο αλγόριθμος θα είναι ο παρακάτω:

Ο αλγόριθμος αυτός προσεγγίζει με αρκετά μεγάλη ακρίβεια τα αποτελέσματα των

```python
import time

def Fibonacci_GoldenRatio(n):
    phi = ( 1 + 5**0.5 ) / 2
    psi = ( 1 - 5**0.5 ) / 2
    return (phi**(n-1) - psi**(n-1))/(5**0.5)

def main():
    n = input("Please give me an integer\n")
    n = int(n)
    while (n <= 0):
        n = input("Please give me another integer\n")
        n = int(n)
    if (n == 1):
        print('The first Fibonacci number is 0')
    elif (n == 2):
        print('The second Fibonacci number is 1')
    elif (n == 3):
        print('The third Fibonacci number is 1')
    else:
        time_a = time.time()
        print(f'Using the golden ratio method, the {n}-th Fibonacci number is {Fibonacci_GoldenRatio(n)}')
        time_b = time.time()
        print(f'The golden ratio method was {time_b - time_a} seconds long')

main()
```

αλγορίθμων του ερωτήματος **(Α)**, ενώ ο χρόνος τον οποίο χρειάζεται για να εμφανίσει το αποτέλεσμα είναι σημαντικά μικρότερος.

Γ) Το bit complexity για την πρόσθεση δύο ακεραίων x και y ισούται με το O(max{$\|x\|$, $\|y\|$}. Από τη μία, ο επαναληπτικός αλγόριθμος εκτελεί n-1 επαναλήψεις (βλ. ερώτημα **(Α)**), επομένως η πολυπλοκότητά του θα είναι O(1 + 2 + 3 +… + n-1) δηλαδή O(n^2). Από την άλλη, στον αλγόριθμο με τη χρήση πινάκων, εκτελούνται και πάλι n-1 επαναλήψεις, οπότε όμοια ο αλγόριθμος θα έχει πολυπλοκότητα O(n^2). Αν λάβουμε υπόψη τον πολλαπλασιασμό ακεραίων, το κόστος τους θα είναι O($\|F(n)\|^2$) = O(n^4), ενώ αν λάβουμε υπόψη τον πολλαπλασιασμό Gauss-Karatsuba το κόστος τους θα είναι O($\|F(n)\|^{log3}$) = O(n^{2*log3}). Παρατηρούμε ότι και οι δύο αλγόριθμοι (με επανάληψη και με πίνακες) έχουν πρακτικά την ίδια πολυπλοκότητα, η οποία μειώνεται ελάχιστα με την αξιοποίηση του πολλαπλασιασμού των Gauss-Karatsuba.

Δ)Εφόσον, από το ερώτημα **(Α)**, ο πιο αποδοτικός αλγόριθμος ήταν αυτός που κάνει χρήση πινάκων, θα τον μετατρέψουμε έτσι, ώστε να τυπώνει τα k least significant beats του n-οστού αριθμού Fibonacci. Το πρόγραμμα γίνεται ως εξής:

```python
import numpy as np

def Fibonacci_matrix(n):
    result = np.zeros((n, n), dtype=int)
    result[0][0] = 0
    result[0][1] = 1
    a, b = 0, 1
    for i in range(2, n):
        c = a + b
        result[0][i] = c
        a = b
        b = c
    return result[0][n-1]

def write_k_times(a):
    return a*'0'

def main():
    n = input("Please give me integer n\n")
    n = int(n)
    k = input("Please give me integer k\n")
    k = int(k)
    while (n <= 0):
        n = input("Please give me another integer n\n")
        n = int(n)
    if (n == 1):
        print(f'The {k} least significant beat(s) of the first Fibonacci number are {write_k_times(k)}')
    elif (n == 2):
        print(f'The {k} least significant beat(s) of the second Fibonacci number are 1{write_k_times(k-1)}')
    elif (n == 3):
        print(f'The {k} least significant beat(s) of the third Fibonacci number are 1{write_k_times(k-1)}')
    else:
        c = Fibonacci_matrix(n)
        result = c % 10**k
        if (result == 0):
            result = k*'0'
        print(f'The {k} least significant beat(s) of the {n}-th Fibonacci number are {result}')

main()
```

Άσκηση 4.

Α) Γράφουμε το εν λόγω πρόγραμμα ως πρόγραμμα σε Python 3 και κάνουμε δοκιμές για τα διάφορα a και b.

```python
def bgcd(a, b):
    if (a<=0 or b<=0):
        return False
    elif (a == b):
        return a
    elif (a%2==0 and b%2==0):
        return 2*bgcd(a/2, b/2)
    elif (a%2==0 and b%2!=0):
        return bgcd(a/2, b)
    elif (a%2!=0 and b%2==0):
        return bgcd(a, b/2)
    else:
        return bgcd(min(a, b), abs(a-b)/2)

def main():
    for i in range (1, 7):
        a = input("Please give me the first number\n")
        a = int(a)
        b = input("Please give me the second number\n")
        b = int(b)
        print(bgcd(a, b))

main()
```

Κάνουμε διάφορες δοκιμές, που περιλαμβάνουν όλες τις προαναφερθείσες περιπτώσεις, και τα αποτελέσματα που λαμβάνουμε είναι τα εξής:

```
Please give me the first number
-1
Please give me the second number
-2
False
Please give me the first number
2
Please give me the second number
2
2
Please give me the first number
4
Please give me the second number
2
2.0
Please give me the first number
3
Please give me the second number
6
3
Please give me the first number
10
Please give me the second number
5
5.0
Please give me the first number
9
Please give me the second number
3
3


...Program finished with exit code 0
Press ENTER to exit console.
```

Μπορούμε να διακρίνουμε την ορθότητα των αποτελεσμάτων πολύ εύκολα, για κάθε περίπτωση. Έτσι, ο αλγόριθμος που περιγράφεται στην εκφώνηση είναι σωστός.

B) Θα εξετάσουμε τις διαφορετικές περιπτώσεις της συνάρτησης bgcd (για μια μόνο επανάληψή του).

- Στις πρώτες δύο περιπτώσεις, γίνεται μια μόνο επανάληψη για τον καθορισμό του αποτελέσματος της συνάρτησης, επομένως η πολυπλοκότητα είναι O(1).
- Στην τρίτη περίπτωση, και οι δύο αριθμοί είναι άρτιοι. Επομένως θα εισέρχονται στο τρίτο if μέχρι ο ένας από τους δύο αριθμούς να ισούται με ένα. Συνεπώς, η πολυπλοκότητα είναι max{a, b}/2.
- Στην τέταρτη περίπτωση, το a είναι άρτιο ενώ το b περιττό. Η συνάρτηση θα καλείται αναδρομικά έως ότου a = 1. Άρα, η πολυπλοκότητα ισούται με a/2.
- Όμοια με την τέταρτη περίπτωση, στην πέμπτη περίπτωση η πολυπλοκότητα ισούται με b/2.
- Στην έκτη και τελευταία περίπτωση, κάθε φορά που καλείται αναδρομικά η συνάρτηση, το b ισούται με $\frac{|a-b|}{2}$. Η αναδρομική κλήση σταματάει όταν b = 0. Ας υποθέσουμε ότι για αυτό απαιτούνται N βήματα, δηλαδή N κλήσεις. Επειδή κάθε φορά επιλέγεται ο ελάχιστος αριθμός ως a, κάνοντας χρήση των αριθμών Fibonacci (βλ. **Άσκηση 3.**) θα ισχύει ότι$a \geqslant f_{N+2}$και $b \geqslant f_{N+1}$. Γνωρίζουμε (επίσης από την **Άσκηση 3.**) πως $F_N = \frac{\varphi^N-(-\varphi)^{-N}}{\sqrt{5}} \approx \phi^n$και $N = log_{\varphi}(F_N)$. Συνεπώς, ισχύει ότι F_{N+1} = min{a, b} και $N + 1 = log_{\varphi}\big(min(a,b)\big)$, άρα O(N) = O(N + 1) = log(min{a, b}).

Συνολικά για τη συνάρτηση bgcd(a, b), έχουμε:

O(N) = 1 + 1 + max{a, b}/2 + a/2 + b/2 + log(min{a, b}), δηλαδή O(N) = log(min{a, b}).

Γ) Γράφουμε τον ευκλείδειο αλγόριθμο μαζί με τον αλγόριθμο bgcd και τα αποτελέσματα

```python
import time

def euclidean_gcd(a, b):
    if (a == 0):
        return b
    return euclidean_gcd(b%a, a)


def bgcd(a, b):
    if (a<=0 or b<=0):
        return False
    elif (a == b):
        return a
    elif (a%2==0 and b%2==0):
        return 2*bgcd(a/2, b/2)
    elif (a%2==0 and b%2!=0):
        return bgcd(a/2, b)
    elif (a%2!=0 and b%2==0):
        return bgcd(a, b/2)
    else:
        return bgcd(min(a, b), abs(a-b)/2)

def main():
    a = input("Please give me the first number\n")
    a = int(a)
    b = input("Please give me the second number\n")
    b = int(b)
    time_a = time.time()
    print(f'The bgcd of {a} and {b} is {bgcd(a, b)} and it was found in ')
    time_b = time.time()
    print(f'{time_b - time_a} seconds.\n')
    print(f'The euclidean gcd of {a} and {b} is {euclidean_gcd(a, b)} and it was found in ')
    time_c = time.time()
    print(f'{time_c - time_b} seconds.\n')


main()
```

που λαμβάνουμε είναι τα κάτωθεν:

```
Please give me the first number
87412365214586215884561
Please give me the second number
98765654122232558789626544585
The bgcd of 87412365214586215884561 and 98765654122232558789626544585 is 3.0 and it was found in
0.00026726722717285156 seconds.

The euclidean gcd of 87412365214586215884561 and 98765654122232558789626544585 is 1 and it was found in
5.316734313964844e-05 seconds.



...Program finished with exit code 0
Press ENTER to exit console.
```

Βλέπουμε πως για πολύ μεγάλους αριθμούς, ο Ευκλείδειος αλγόριθμος δεν λειτουργεί σωστά. Επομένως, προτιμούμε τον αλγόριθμο bgcd.

Άσκηση 5.

Α)Θα θεωρήσουμε τον γράφο κατευθυνόμενο, χωρίς κύκλους, και με βάρη το κόστος των διοδίων. Για την εύρεση της βέλτιστης διαδρομής:

- Πρωτίστως θα βρούμε την τοπολογική διάταξη του γράφου (υπάρχει, εφόσον δεν περιέχει κύκλους). Οι κορυφές του γράφου αποθηκεύονται σε έναν πίνακα Α, διατηρώντας τη σειρά της τοπολογικής διάταξης.

- Έπειτα, θεωρούμε δύο κόμβους m και n, και τα κόστη των διοδίων S(m) = 0 ≠S(n) (για κάθε m≠n) για τη διαδρομή από τον κόμβο m στον κόμβο n.
- Έχουμε έναν δείκτη d στον κόμβο που προηγείται του κόμβου n στην (ως τώρα) βέλτιστη διαδρομή που έχουμε βρει.
- Στην τοπολογική ταξινόμηση, ξεκινάμε ελέγχοντας αν ο κόμβος n είναι ο πρώτος κόμβος της διάταξης. Στην περίπτωση που δεν είναι, η πολυπλοκότητα του αλγορίθμου ελαττώνεται, καθώς δεν χρειάζεται να ελέγξουμε τις διαδρομές για τους προηγούμενους κόμβους.
- Για κάθε κόμβο i ≥ n, ελέγχουμε σε ποιους κόμβους j οδηγούν οι ακμές του πρώτου. Για κάθε ζευγάρι κόμβων, ελέγχουμε αν ισχύει η εξής σχέση: S(i) + K(i, j) < S(j) (όπου K(i, j) το κόστος της διαδρομής -βάρος της ακμής του γράφου- από τον κόμβο i στον κόμβο j). Αν κάτι τέτοιο ισχύει, θεωρούμε πως S(j) = S(i) + K(i, j) και θέτουμε τον δείκτη d στον κόμβο i. Επαναλαμβάνουμε την ίδια διαδικασία για όλα τα στοιχεία του πίνακα A.
- Το αποτέλεσμα του αλγορίθμου αυτού δίνει το S(n) που είχαμε υποθέσει αρχικά, ως το μικρότερο κόστος διαδρομής από τον κόμβο m στον κόμβο n.

Β) Σε αυτή την περίπτωση, δεν μπορούμε να εκμεταλλευτούμε την τοπολογική ταξινόμηση. Μπορούμε, όμως, να εκμεταλλευτούμε τον αλγόριθμο του Dijkstra.

- Όμοια με προηγουμένως, θα τοποθετήσουμε όλους τους κόμβους σε μια ουράunvisited κόμβων. Θεωρούμε δύο κόμβους m και n, και τα κόστη των διοδίων S(m) = 0 ≠S(n) (για κάθε m≠n) για τη διαδρομή από τον κόμβο m στον κόμβο n. Έχουμε έναν δείκτη d στον κόμβο που προηγείται του κόμβου n στην (ως τώρα) βέλτιστη διαδρομή που έχουμε βρει.
- Για κάθε κόμβο i , ελέγχουμε σε ποιους unvisited κόμβους j οδηγούν οι ακμές του πρώτου. Για κάθε ζευγάρι κόμβων, ελέγχουμε αν ισχύει η εξής σχέση: S(i) + K(i, j) < S(j) (όπου K(i, j) το κόστος της διαδρομής -βάρος της ακμής του γράφου- από τον κόμβο i στον κόμβο j). Αν κάτι τέτοιο ισχύει, θεωρούμε πως S(j) = S(i) + K(i, j) και θέτουμε τον δείκτη d στον κόμβο i. Επαναλαμβάνουμε την ίδια διαδικασία έως ότου η ουρά των unvisited κόμβων να αδειάσει, δηλαδή μέχρι όλοι οι κόμβοι να γίνουν visited. Visited θεωρείται ένας κόμβος όταν όλοι οι unvisited γείτονές του έχουν ελεγχθεί.
- Στο τέλος του αλγορίθμου θα πρέπει να μην υπάρχουν unvisited κόμβοι. Για κάθε κόμβο n το συνολικό κόστος S(n) ισούται με το ελάχιστο κόστος της διαδρομής από τον αρχικό κόμβο m στον εκάστοτε τελικό κόμβο n, ενώ ο δείκτης d δείχνει πάντα τον κόμβο που προηγείται του κόμβου n στη διαδρομή από τον m στον n.

Γ) Καθώς ο αλγόριθμος μπορεί να περιέχει κύκλους ή αρνητικά βάρη (δηλ. σε μια ακμή η προσφορά να υπερβαίνει το κόστος), δεν μπορούμε να χρησιμοποιήσουμε κανέναν από τους αλγορίθμους των προηγούμενων δύο ερωτημάτων. Μπορούμε, ωστόσο, να χρησιμοποιήσουμε τον αλγόριθμο Bellman-Ford.

- Όμοια με προηγουμένως, θεωρούμε δύο κόμβους m και n, και τα κόστη των διοδίων S(m) = 0 ≠S(n) (για κάθε m≠n) για τη διαδρομή από τον κόμβο m στον κόμβο n. Έχουμε έναν δείκτη d στον κόμβο που προηγείται του κόμβου n στην (ως τώρα) βέλτιστη διαδρομή που έχουμε βρει.

- Για κάθε ζευγάρι ακμών (I, j), ελέγχουμε αν ισχύει η ανισότητα S(i) + K(i, j) < S(j) και, στην περίπτωση που ισχύει, θεωρούμε πως S(j) = S(i) + K(i, j) και θέτουμε τον δείκτη d στον κόμβο i.
- Αν ο γράφος έχειbακμές, τότε επαναλαμβάνουμε τον αλγόριθμο b-1 φορές. Έτσι, στο τέλος των επαναλήψεων, το συνολικό κόστος S(n) δηλώνει το ελάχιστο κόστος διαδρομής από τον αρχικό κόμβο m ως τον εκάστοτε τελικό κόμβο n.

Δ) Για τις πολυπλοκότητες:

Α) Είναι γνωστό πως η τοπολογική ταξινόμηση με διάσχιση DFS έχει πολυπλοκότητα O(a+b), όπου a οι κορυφές και b οι ακμές.

Β) Οι κορυφές πρέπει να τοποθετηθούν στην ουρά unvisited (O(a)), καθώς και να αφαιρεθούν από αυτή (O(a)). Κάθε φορά που υπάρχει αλλαγή στην ουρά, ανανεώνεται η προτεραιότητα των κόμβων, το πολύ b φορές. Συνολικά, κάτι τέτοιο έχει πολυπλοκότητα O(m*logn)

C) Επαναλαμβάνουμε b-1 φορές τον αλγόριθμο για a κορυφές, επομένως ο αλγόριθμος έχει πολυπλοκότητα O(a*b).