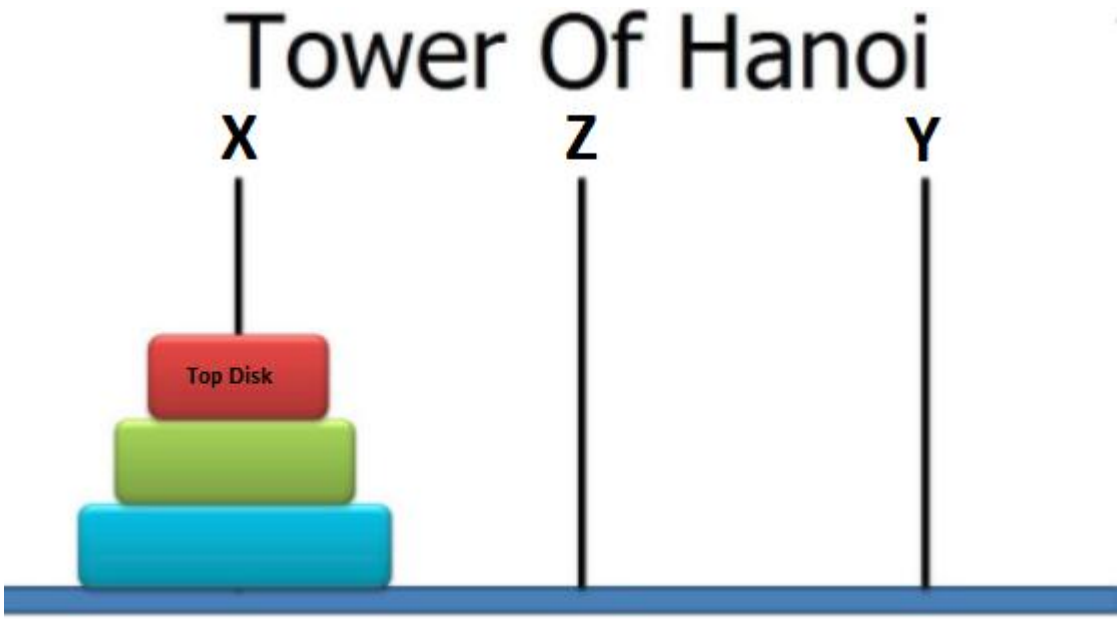
2η Σειρά Ασκήσεων

Ονοματεπώνυμο:	**Αγγλογάλλος Αναστάσιος**
Αριθμός Μητρώου:	**El18641**

Άσκηση 1 (Hanoi)

Α)

Η διάταξη του προβλήματος στην αρχική του κατάσταση:



Ο αναδρομικός αλγόριθμος που κάναμε για την άσκηση αυτή είναι:

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

Έστω λοιπόν ότι χρειαζόμαστε A(n) κινήσεις , όπου n ο αριθμός των δίσκων που έχουμε αρχικά στοιβαγμένους στον x.

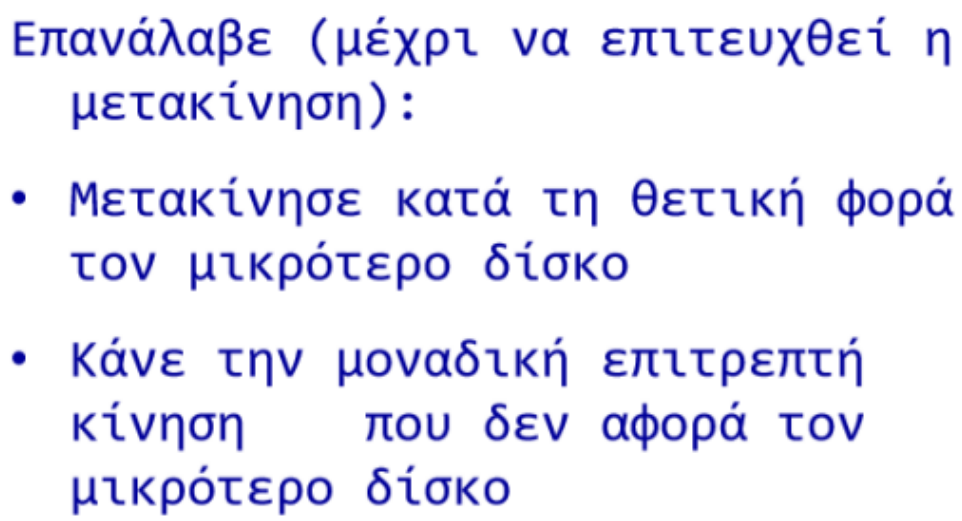
Στην περίπτωση που n=1 τότε χρειαζόμαστε μια μόνο κίνηση top disk X->Y άρα A(n=1)=1.
Για περισσότερους δίσκους η στρατηγική μας είναι η εξής:
Μετακινούμε τους n-1 δίσκους στο Z το οποίο μέσω της αναδρομής χρειάζεται A(n-1) κινήσεις.
Έπειτα μετακινούμε τον μεγαλύτερο δίσκο που έχει μείνει στο X->Y, 1 κίνηση.
Τέλος μετακινούμε τους n-1 δίσκους από το Z στο Y το οποίο αντίστοιχα χρειάζεται A(n-1) κινήσεις.

Άρα:

A(n)= 2A(n-1)+1 και A(1)=1

A(n)=2*(2A(n-2)+1)+1 =>
A(n) = 2^{n-1}A (1)+2^{n-2}+2^{n-3}+…+2^{1}+2^{0}=>
A(n)= 2^{n}-1

Β) Ο επαναληπτικός αλγόριθμος που κάναμε στο μάθημα είναι ο εξής:



Επιλέγουμε θετική φορά την X->Y->Z->X->… ανάλογα με το πλήθος των δίσκων (άρτιο ή περιττό) η τελική στήλη μπορεί να είναι Y, για περιττό n, και Z για άρτιο n.
Και στις δύο περιπτώσεις όμως το πλήθος των κινήσεων παραμένει ίδιο. Επίσης εάν θέλουμε να καταλήγει στην Y τελική στήλη για άρτια n τότε θέτουμε την φορά X->Z->Y->X->…

Για n=1 χρειαζόμαστε μόνο μία κίνηση.
Έστω τώρα ότι για n δίσκους ο αλγόριθμος χρειάζεται να κάνει 2^n-1 κινήσεις.
Τώρα αρκεί να αποδείξουμε ότι για n+1 δίσκους χρειαζόμαστε 2^{n+1}-1 κινήσεις.
Έτσι από την υπόθεση που θεωρήσαμε χρειαζόμαστε 2^n-1 κινήσεις για την μεταφορά των n δίσκων στην βοηθητική στήλη.
Έπειτα ο n+1 δίσκος θα πάει στην ελεύθερη στήλη με 1 κίνηση.
Τέλος με 2^n-1 κινήσεις θα μεταφερθούν οι n δίσκοι στην τελική τους θέση πάνω από τον n+1 δίσκο.
Άρα χρειαζόμαστε $2*(2^n-1)+1$ κινήσεις δηλαδή 2^{n+1}-1 κινήσεις. Άρα οι δύο παραπάνω αλγόριθμοι χρειάζονται τον ίδιο αριθμό κινήσεων για να λύσουν το πρόβλημα.

Γ)Έστω ότι οι ελάχιστες κινήσεις που χρειάζονται για την επίλυση του προβλήματος είναι M(n).
Τότε M(n=1)=1 όσο καλός και να είναι ο αλγόριθμος.

Εάν n>1 τότε πρέπει με κάποιο τρόπο να μετακινήσουμε τον μεγαλύτερο δίσκο στην τελική στήλη ,άρα αναγκαστικά πρέπει να τοποθετήσουμε τους n-1 δίσκους σε μια ενδιάμεση στήλη κάνοντας τουλάχιστον M(n-1) κινήσεις , έπειτα μετακινούμε τον μεγαλύτερο δίσκο στην τελική του θέση με μία κίνηση και τέλος πρέπει να τοποθετήσουμε τους n- 1 δίσκους στην τελική τους θέση κάνοντας τουλάχιστον M(n-1) κινήσεις.

Άρα χρειαζόμαστε $M(n) \geqslant 2M(n-1)+1$ κινήσεις και επειδή M(1)=1 Παρατηρούμε ότι η ισότητα αυτής της ανίσωσης είναι η σχέση των κινήσεων των δύο παραπάνω αλγορίθμων συνεπώς είναι και οι βέλτιστοι.

Δ)Μια εύκολη και προφανής λύση του προβλήματος είναι να χρησιμοποιήσουμε τον ήδη έτοιμο αναδρομικό αλγόριθμο move_anoi για να λύσουμε το πρόβλημα.

Συγκεκριμένα να μετακινήσουμε τους n-2 δίσκους σε μία ενδιάμεση στήλη , το n-1 δίσκο σε μια ενδιάμεση στήλη και έπειτα τον μεγαλύτερο δίσκο n στην τελική του θέση και έπειτα τον n-1 δίσκο από επάνω του. Τέλος μετακινούμε τους n-2 δίσκους στην τελική τους θέση.
Με την χρήση του πρώτου αναδρομικού αλγόριθμου χρειαζόμαστε τις εξής κινήσεις:

A(n) = 2*A(n-2) + 2*1 + 1 και ισχύει A(1)=1 και A(0)=0 οπότε

A(n) = 2*(2*(A(n-4) + 3)) + 3 = 2^2A(n-4)+2*3 +1*3 = 2^2(A(n-6)+3) + 2^1*3 + 2^0*3= …
Εάν το n είναι άρτιος θα καταλήξουμε στην μορφή:

A(n άρτιος) = 2^xA(n-2x) + 3*2^{x-1}
για n άρτιος καταλήγουμε σε A(0)=0 δηλαδή n-2x=0 άρα x=n/2
A(n άρτιος) = $2^{n/2}$A(0) + 3*$2^{n/2-1}$= 3*$2^{(n-2)/2}$ κινήσεις

Εάν το n είναι περιττό τότε:
A(n περιττός) = 2^xA(n-2x) + 3*2^{x-1}
και θα καταλήξουμε σε A(1)=1 δηλαδή n-2x=1 άρα x=(n-1)/2
A(n περιττός) = $2^{(n-1)/2}$A(1) + 3*$2^{(n-1)/2-1}$
A(n περιττός) = $2^{(n+1)/2}$ -3 κινήσεις

Όμως υπάρχει και πιο γρήγορος αλγόριθμος.
…
Το πρόβλημα αυτό ονομάζεται Reve's Puzzle και το 2014 αποδείχτηκε η καλύτερη λύση του για n<=30.
Ο αλγόριθμος που χρησιμοποιεί είναι ο εξής:

Διαλέγουμε έναν αριθμό k , ο οποίος είναι μεταξύ του ένα και του n (αριθμός των δίσκων)
Για n=1 απλά μετακινούμε τον δίσκο στην στήλη 4 αλλιώς:
- Μετακίνησε τους n-k «μικρότερους» δίσκους από την πρώτη(αρχική) στήλη στην δεύτερη στήλη χρησιμοποιώντας και τις τέσσερις στήλες
-Μετακίνησε τους υπόλοιπους δίσκους στην στήλη 4 ,χρησιμοποιώντας το move_anoi , 2^k
-Τέλος μετακινούμε τους n-k «μικρότερους» δίσκους από την δεύτερη στήλη στην τέταρτη στήλη(τελική θέση) χρησιμοποιώντας και τις τέσσερις στήλες
Το k που πρέπει να χρησιμοποιήσουμε είναι το στρογγυλοποιημένο n- (2n+1)^(1/2) +1

Άσκηση 2 (Πρώτοι αριθμοί)

Α) Για να λειτουργήσει το παρακάτω πρόγραμμα (σε Python) σε αποδοτικό χρόνο βασιζόμαστε στην παρακάτω σχέση-ιδιότητα :
a*b mod m = [(a mod m) *(b mod m)] mod m
b^e mod m =
$(b^2)^{e/2}$ mod m =

$(b^2*b^2*...b^2)$ mod m =
$[(b^2 \bmod m)*(b^2 \bmod m)*...*(b^2 \bmod m)] \bmod m =$
$(b^2 \bmod m)^{e/2} \bmod m$

Άρα $b^e \bmod m = (b^2 \bmod m)^{e/2} \bmod m$.

Διαλέγουμε επίσης 30 τυχαίους αριθμούς μεταξύ του 2 και το n-2 στους οποίους εφαρμόζουμε το τεστ του Fermat και αν και οι 30 αριθμοί περάσουν
το τεστ του Fermat ο n ανακυρήσεται πρώτος.
Ο κώδικας:

```
1   import random
2   def fastmod(b, e, m):
3     if(e==0):
4       return 1
5     if (e%2==0):
6       return fastmod((b*b)%m, e/2, m)
7     else :
8       return (b*fastmod((b*b)%m, (e-1)/2, m))%m
9
10  def Fermat(n):
11    flag = True
12    for i in range(30):
13      if(fastmod(random.randint(2,n-2), n-1, n) != 1):
14        flag = False
15        break
16    if(flag):
17      print(n, 'is Prime')
18    else:
19      print(n, 'is not Prime')
20  n = int(input(' n :'))
21  Fermat(n)
```

Δοκιμή αλγορίθμου με αριθμούς :

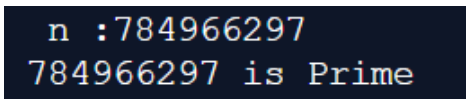
```
 n :67280421310721
67280421310721 is Prime
 n :170141183460469231731687303715884105721
170141183460469231731687303715884105721 is not Prime
```

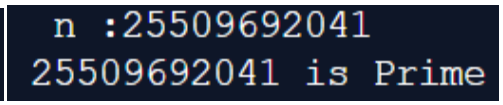
Για τους υπόλοιπους αριθμούς οι οποίοι είναι αστρονομικά μεγάλοι χρησιμοποιούμε την εξής ιδιότητα.
Αν το x δεν είναι πρώτος, τότε ούτε το 2^x-1 είναι πρώτος.

```
 n :2281
2281 is Prime
 n :19939
19939 is not Prime
```

```
 n :9941
9941 is Prime
```

Ελέγχοντας το πρόγραμμα με αρκετούς μεγάλους αριθμούς Carmichael παρατηρώ όπως είναι αναμενόμενο ότι αρκετοί περνούν το τεστ του Fermat χωρίς να είναι πρώτοι.

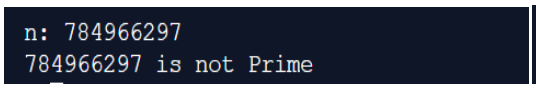


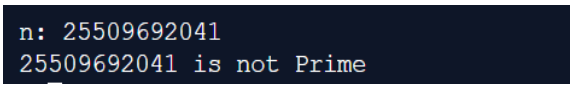


B)

Δοκιμάζοντας τον ελέγχο Miller-Rabin στους προηγούμενους αριθμούς του ερωτήματος Α που δεν ήταν αριθμοί Carmichael παίρνουμε τα ίδια αποτελέσματα.

Όταν δοκιμάζουμε τώρα τους αριθμούς Carmichael, περνάνε το τεστ.





Ο κώδικας:

```python
import random

def Prime(n):

    if n==0 or n==1 or n==4 or n==6 or n==8 or n==9:
      return False

    if n==2 or n==3 or n==5 or n==7:
      return True
    s = 0
    d = n-1
    while(d%2 == 0):
      d>>=1
      s+=1

    def trial_composite(a):
      if(pow(a, d, n) == 1):
        return False
      for i in range(s):
        if(pow(a, 2**i * d, n)== n-1):
          return False
      return True


    for i in range(8):
      a = random.randrange(2, n)
      if trial_composite(a):           return False

    return True

n = int(input('n: '))

if(Prime(n)):
  print(n, 'is Prime')
else:
  print(n, 'is not Prime')

```

γ)
Για όλους τους περιττούς αριθμούς από το 100 μέχρι το 3000 εφαρμόζουμε τον έλεγχο Miller-Rabin του ερωτήματος β. Για όσους ακεραίους x ο έλεγχος είναι θετικός τότε εφαρμόζουμε τον ίδιο έλεγχο για τον 2^x-1 και αν ο έλεγχος βγει και πάλι θετικός τότε αυτός είναι αριθμός Mersenne

Ο κώδικας :

```python
import random

def Prime(n):

    if n==0 or n==1 or n==4 or n==6 or n==8 or n==9:
      return False

    if n==2 or n==3 or n==5 or n==7:
      return True
    s = 0
    d = n-1
    while(d%2 == 0):
      d>>=1
      s+=1

    def trial_composite(a):
      if(pow(a, d, n) == 1):
        return False
      for i in range(s):
        if(pow(a, 2**i * d, n)== n-1):
          return False
      return True


    for i in range(8):
      a = random.randrange(2, n)
      if trial_composite(a):           return False

    return True

for i in range(101,3000,2):
  if (Prime(i)):
    if(Prime(2**i-1)):
      print(i)
```

```
107
127
521
607
1279
2203
2281
```

Άσκηση 3 (Fibonacci)

A)

```
4  def fib3(n): //fibonacci me pinakes
5    F=((1,1), (1,0))
6    if (n==0):
7        return 0
8    power(F ,n-1)
9    print(F(0)(0))
10
11 def power(F, n): //ypswsh se dynamh
12     A =( (1,1), (1,0))
13     for i in range (2,n):
14     a = (F(0)(0) * A(0)(0) + F(0)(1) * A(1)(0)
15     b = (F(0)(0) * A(0)(1) + F(0)(1) * A(1)(1))
16     C = (F(1)(0) * A(0)(0) + F(1)(1) * A(1)(0))
17     d = (F(1)(0) * A(0)(1) + F(1)(1) * A(1)(1))
18
19         F(0)(0) = a
20         F(0)(1) = b
21         F(1)(0) = c
22         F(1)(1) = Z
```

Με τη χρήση memorization και επανάληψης, η πολυπλοκότητα του αλγορίθμου είναι O(n), ενώ . με πίνακα είναι O(logn).

B) Γνωρίζουμε ότι ισχύει :

$$x_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}.$$

Παρατηρούμε πως με την χρήση δηλαδή της χρυσής τομής, μπορούμε να υπολογίσουμε τον ν-οστό αριθμό Fibonacci χωρίς να χρειαστεί να υπολογίσουμε τους 2 προηγούμενους , δηλαδή ακόμη πιο γρήγορα!

Όμως, με αυτήν την μέθοδο χάνεται η ακρίβεια, καθώς το φ είναι άρρητος και άρα το αποτέλεσμα x_n δεν θα είναι ακέραιος. Αν και για μικρά n αυτό δεν αποτελεί πρόβλημα και η στρογγυλοποίηση σε ακέραιο γίνεται εύκολα, για μεγάλα n με στρογγυλοποίηση το αποτέλεσμα θα αποκλίνει πολύ από το πραγματικό καθιστώντας αυτή την μέθοδο ακατάλληλη

Γ) Γνωρίζουμε ότι, για να λάβουμε τα k λιγότερο σημαντικά ψηφία ενός αριθμού, αρκεί να πάρουμε το modulo του αριθμού με το 10^k. Τροποποιούμε λίγο έναν από τους αλγόριθμους του ερωτήματος (α):

```
1   def fib4(n):
2     A=[0]*n
3     A[1]=1
4     A[2]=1
5     for i in range(3,n):
6       A[i]=A[i-1]+A[i-2]
7     return A[n-1]%pow(10,k)
```

Άσκηση 4 (Διόδια)

Α)
Ανάγουμε το πρόβλημα μας στο πρόβλημα εύρεσης ελάχιστου μονοπατιού σε κατευθυνόμενο άκυκλο γράφο (DAG) με βάρη. Για την αναγωγή κάνουμε στον γράφο μας τον εξής μετασχηματισμό, θέτουμε ως βάρος κάθε κατευθυνόμενης ακμής του γράφου την τιμή των διοδίων του κόμβου στον οποίο καταλήγει η εκάστοτε ακμή.

Αλγόριθμος:

- *Χρησιμοποιούμε τον αλγόριθμο της τοπολογικής ταξινόμησης για να πάρουμε μία τοπολογική απεικόνιση του γράφου.*
- *Έπειτα, εφαρμόζουμε τον αλγόριθμο SP-DAG ,ο οποίος έχει ως εξής:*
  - *Θέτουμε τις αποστάσεις προς όλους τους κόμβους στο άπειρο, εκτός από τον αρχικό μας κόμβο του οποίου την απόσταση θέτουμε 0.*

- Για κάθε κόμβο, έστω u, στη σειρά της τοπολογικής ταξινόμησης, ελέγχουμε κάθε γείτονα του, έστω v, και εάν η απόσταση για τον v είναι μικρότερη από το άθροισμα της απόστασης προς τον u και του βάρους της ακμής που ενώνει τους u και v, αντικαθιστούμε την απόσταση αύτη με το άθροισμα αυτό.

Ο μετασχηματισμός των ακμών χρειάζεται γραμμικό χρόνο, όπως και η εύρεση του συντομότερου μονοπατιού άρα συνολικά το πρόβλημα λύνεται σε γραμμικό χρόνο.

πολυπλοκότητα : O(V+E).

β)
Όπως και στο προηγούμενο ερώτημα πρέπει να κάνουμε αναγωγή του προβλήματος μας για να μπορέσουμε να το λύσουμε. Αυτή τη φορά το ανάγουμε γενικότερα στο πρόβλημα εύρεσης του ελάχιστου μονοπατιού σε κατευθυνόμενο γράφο με βάρη γενικότερα, δηλαδή ο γράφος μπορεί να έχει κύκλους. Ξεκινάμε με το μετασχηματισμό, θέτοντας ως βάρος κάθε κατευθυνόμενης ακμής του γράφου την τιμή των διοδίων του κόμβου στον οποίο καταλήγει η εκάστοτε ακμή.

Ο γενικότερος αλγόριθμος για την εύρεση του ελάχιστου μονοπατιού χωρίς αρνητικά βάρη σε κατευθυνόμενο γράφο είναι ο αλγόριθμος του Dijkstra ο οποίος δίνεται παρακάτω:

Θέτουμε την απόσταση από τον αρχικό κόμβο προς τον αρχικό κόμβο 0
Θέτουμε την απόσταση για όλους τους άλλους κόμβους από τον αρχικό κόμβο ως +∞

Όσο (while) { οι κόμβοι παραμένουν unvisited
. Επισκέψου τον κόμβο με την μικρότερη απόσταση από τον αρχικό κόμβο.
. Για κάθε (for) { unvisited κόμβο-γείτονα από τον κόμβο που βρισκόμαστε :
. Υπολογίζουμε την απόσταση από τον αρχικό κόμβο
. Εάν (if) { η απόσταση αυτή είναι μικρότερη από την απόσταση που έχουμε καταγεγραμένη τότε
. την αντικαθιστούμε και ανανεώνουμε τον πίνακα. }
. Επισκεπτόμαστε τον επόμενο unvisited κόμβο και προσθέτουμε τον προηγούμενο στην λίστα .
. με τους visited κόμβους. } }

Μέσω αυτού του αλγορίθμου μπορούμε να βρούμε την φτηνότερη διαδρομή.
Ο αλγόριθμος του Dijkstra έχει πολυπλοκότητα O(E+V log(V)).
Πολυπλοκότητα : O(E+V log(V)).

γ)
Σε περίπτωση που οι προσφορές είναι μεγαλύτερες από το κόστος διέλευσης έχουμε κέρδος όταν περνάμε. Άρα ανάγουμε το πρόβλημα μας στο πρόβλημα εύρεσης του ελάχιστου μονοπατιού σε κατευθυνόμενο γράφο με θετικά και αρνητικά βάρη. Κάνουμε τον κατάλληλο μετασχηματισμό, θέτοντας ως βάρος κάθε κατευθυνόμενης ακμής του γράφου την τιμή των διοδίων του κόμβου στον οποίο καταλήγει η εκάστοτε ακμή.

Για το πρόβλημα εύρεσης του ελάχιστου μονοπατιού σε κατευθυνόμενο γράφο με θετικά και αρνητικά βάρη χρησιμοποιούμε τον αλγόριθμο Bellman-Ford:

- *Θέτουμε την τιμή όλων των κόμβων στο άπειρο και την απόσταση του αρχικού(τρέχον κόμβος) στο 0.*

- *Ορίζουμε μία συγκεκριμένη σειρά με την οποία θα επισκεφτούμε όλους τους κόμβους.*

- *Ελέγχουμε τους γείτονες του τρέχοντος κόμβου. Αν η τιμή του τρέχοντος, έστω u, συν το βάρος της ακμής E(u,v) είναι μικρότερη από την τιμή του γείτονα v τότε την αντικαθιστούμε.*
  Βασική

- *Θέτουμε τρέχοντα κόμβο τον επόμενο της σειράς που ορίσαμε και επαναλαμβάνουμε το προηγούμενο βήμα*

- *Μόλις ολοκληρώσουμε το 3ο βήμα για όλους τους κόμβους ξεκινάμε ξανά από την αρχή το 3ο βήμα. Εάν για μια επανάληψη όλων των κόμβων της σειράς δεν γίνει ανανέωση της ελάχιστης απόστασης για κανένα κόμβο αλγόριθμος τερματίζει.*

Ξεκινώντας από τον τελικό κόμβο κρατάμε τον «πρόγονο» από τον οποίο οδηγηθήκαμε σε αυτόν μέσω της ελάχιστης διαδρομής και επαναλαμβάνουμε μέχρι να φτάσουμε στον αρχικό ώστε να βρούμε το

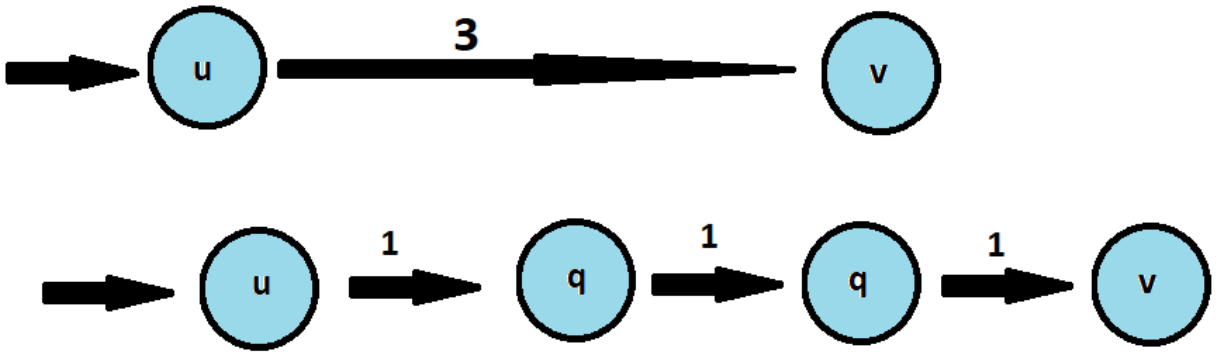
ελάχιστο μονοπάτι. Έτσι παίρνουμε το ελάχιστο κόστος και την διαδρομή σε κόμβους που πρέπει να ακολουθήσουμε από τον αρχικό στον τελικό για το κόστος αυτό.
Μέσω του αλγορίθμου αυτού βρίσκουμε την φτηνότερη διαδρομή και σε περίπτωση που υπάρχουν προσφορές.
Η πολυπλοκότητα του αλγορίθμου ταυτίζεται με αυτήν του αλγορίθμου Bellman-Ford, είναι δηλαδή O(|V||E|)

Άσκηση 5 (Συντομότερο Μονοπάτι)

Θεωρούμε το κατευθυνόμενο γράφημα G(V,E,l), με d(v,u) την απόσταση 2 κορυφών v, u. Θέλουμε να κατασκευάσουμε αλγόριθμο πολυπλοκότητας Θ(n+m), ο οποίος φτάνει από κάθε κορυφή-κόμβο σε όλες τις άλλες με ελάχιστο τρόπο και ελέγχει αν οι αποστάσεις των κορυφών μεταξύ τους ανταποκρίνονται στα δ_i.

Το πρόβλημα αυτό θα μπορούσε να επιλυθεί με τον αλγόριθμο Dijkstra, ο οποίος όμως, στην καλύτερη περίπτωση υλοποίησής του (υλοποίηση ουράς με σειρά Fibonacci) είναι πολυπλοκότητας O(E+VlogV).

Εφόσον αναζητούμε αλγόριθμο πολυπλοκότητας Θ(n+m), στρεφόμαστε στον αλγόριθμο BFS, ο οποίος χρησιμοποιείται για τη διάσχιση δένδρων. Η BFS όντως απαιτεί χρόνο Θ(n+m), όμως είναι δυνατό να εφαρμοστεί μόνο σε ακμές χωρίς βάρη (ή σε ακμές με ίσα βάρη, αφού και πάλι κάθε ακμή θα εξισώνεται με την άλλη ως προς το βάρος). Για να λύσουμε αυτό το πρόβλημα, κατασκευάζουμε πλασματικούς κόμβους, τους οποίους συνδέουμε με πλασματικές ακμές, ως εξής: για κάθε ακμή με βάρος k, κατασκευάζουμε k-1 ψευδο-κόμβους οι οποίοι συνδέονται μεταξύ τους με k ψευδο-ακμές μήκους 1, όπως φαίνεται παρακάτω για k=3:



Έχοντας «σπάσει» τις ακμές σε ακμές μοναδιαίου μήκους, ο αλγόριθμος BFS θα εντοπίσει το συντομότερο μονοπάτι από κάθε κόμβο σε όλες τις άλλες, και θα «κλειδώσει» τον κόμβο, ώστε να τον οριστικοποιήσει (ο κόμβος δεν θα εξερευνηθεί ξανά).

Κατασκευάζουμε έναν πίνακα d[] προκειμένου να αποθηκεύσουμε τις αποστάσεις που δίνει ο αλγόριθμος για κάθε κόμβο (τις οποίες μπορούμε να εξισώσουμε με το πόσοι κόμβοι και ψευδο-κόμβοι παρεμβάλλονται ανάμεσα στον αρχικό και στον τελικό κόμβο). Επίσης, κατασκευάζουμε έναν πίνακα p[] ώστε να αποθηκεύεται ο προηγούμενος κόμβος του κόμβου που κλειδώνει.

Κάθε φορά που κάποιος κόμβος «κλειδώνει», εξετάζουμε αν η απόσταση d[n] που αποθηκεύσαμε είναι ίση με τη δ_n που δίνεται. Σε περίπτωση που για κάθε n οι αποστάσεις d[n] βρεθούν ίσες με τις δ_n, επιστρέφουμε τα στοιχεία d[i] και δ_i για κάθε κόμβο i, και έτσι κατασκευάζεται το δέντρο συντομότερων μονοπατιών, χωρίς να ξεπεραστεί το όριο Θ(n+m).

Άσκηση 6 (Διακοπές)

Α)
Αρχικά ο μόνος τρόπος για να μην είναι εφικτή μια διαδρομή είναι εάν ισχύει w(e) > L .Άρα το πρόγραμμα :

- *θα διαγράφει όλες τις ακμές όπου ισχύει η προηγούμενη σχέση.*
- *Έπειτα μέσω DFS θα αναζητούμε μονοπάτι που να αρχίζει από το s (πρωτεύουσα-αφετηρία) και να τελειώνει στο t (θέρετρο-στόχος). Εάν βρίσκει το t τότε θα επιστρέφει true, αλλιώς false.*

πολυπλοκότητα : O(max{2m+n})= O(m)

Β)

Αφού γνωρίζουμε ότι το οδικό δίκτυο μπορεί να αναπαρασταθεί με την μορφή ενός μη κατευθυνόμενου συνεκτικού γράφου , για να βρούμε την πιό cost-efficient διαδρομή αρκεί να χρησιμοποιήσουμε τον αλγόριθμο του Kruskal.

Γνωρίζουμε ότι ο αλγόριθμος του Kruskal έχει την δυνατότητα να βρίσκει το minimum spanning tree και έχει πολυπλοκότητα O(E*log(E)) αν εφαρμόσουμε merge sort για την ταξινόμηση των ακμών.
(στην περίπτωση μας O(m*log(m))).

Έπειτα θα εφαρμόσουμε DFS για να βρούμε τον κόμβο t και έπειτα ακολουθώντας τον πίνακα που φτιάχτηκε με το DFS, θα συγκρίνουμε κάθε ακμή από τον t προς τα πίσω κρατώντας την μεγαλύτερη, μέχρι να βρεθούμε στην ρίζα. Έτσι βρίσκουμε την μέγιστη ακμή ,που είναι και η ελάχιστη αυτονομία καυσίμου.

Αλγόριθμος:

- *Εφαρμόζουμε Kruskal στο μη κατευθυνόμενο γράφημα χρησιμοποιώντας merge sort.*
- *Έπειτα εφαρμόζουμε DFS μέχρι να βρούμε τον κόμβο t και θα αποθηκεύονται σε μια στοίβα οι γονείς.*

*/*αναφέρομαι στην στοίβα που χρησιμοποιούμε ούτως ή αλλως στο DFS*/*

- *Μόλις βρεθεί ο t ακολουθώντας την στοίβα θα συγκρίνουμε τα w(e) και αν το προηγούμενο στοιχείο στην στοίβα έχει μεγαλύτερο w(e) τότε θα την ορίζουμε σε μία μεταβλητή (π.χ. max_w) μέχρι να φτάσουμε στην ρίζα.*

- *Τέλος θα επιστρέφει την μεταβλητή αυτή.*

Άσκηση 7 (Εισιτήρια για την Σχολή)

Θα κατασκευάσουμε ένα πίνακα S της παρακάτω μορφής:

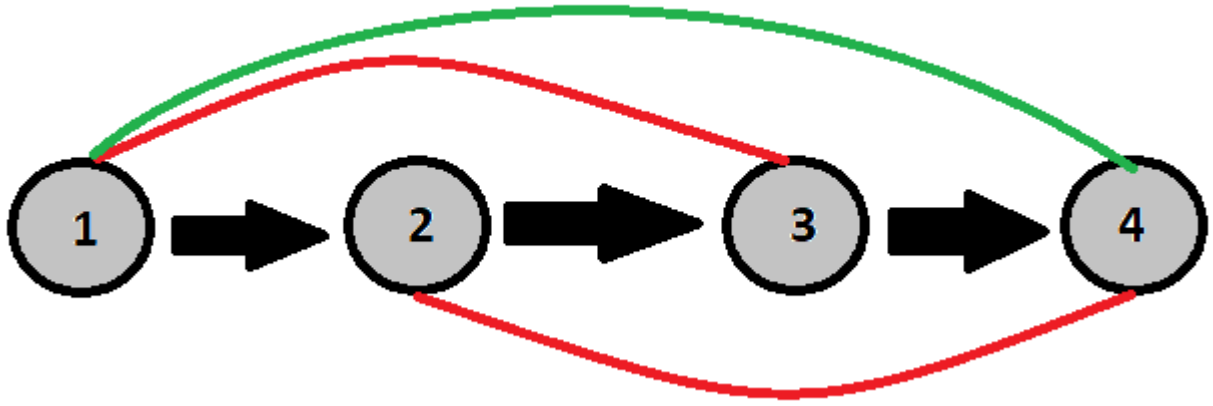
t=1	t=2	t=…	t=T
0	1	0/1	0

Θεωρούμε λοιπόν πίνακα S[] με T θέσεις, από μία για κάθε ημέρα t.

Για τις ημέρες που θα πάμε στο ΕΜΠ, S[t]=1, ενώ για τις υπόλοιπες S[t]=0.
Το κάθε εισιτήριο για να μεταβούμε στο ΕΜΠ έχει κόστος p_i και συμβολίζεται ως c_i, όπου i οι ημέρες που καλύπτει το εισιτήριο.
Γνωρίζουμε ότι η τιμή αυξάνεται με τη διάρκεια του εισιτηρίου αλλά και ότι, όσο μεγαλύτερη είναι η διάρκεια του, η τιμή ανά ημέρα μειώνεται. Συνεπώς μας συμφέρει να αγοράζουμε πολυήμερα εισιτήρια για πολυήμερες παρακολουθήσεις τις σχολής.
Ζητείται το ελάχιστο κόστος για την μετάβασή μας στο ΕΜΠ στο σύνολο των T ημερών, δηλαδή το κόστος του καλύτερου συνδυασμού εισιτηρίων που μπορούμε να πάρουμε. Συμβολίζουμε το ζητούμενο κόστος με cost.

Θα χρησιμοποιήσουμε δυναμικό προγραμματισμό και θα «σπάσουμε» το πρόβλημα σε μικρότερα υποπροβλήματα, η λύση των οποίων θα πρέπει να μας οδηγήσει στην συνολική λύση.

Θεωρούμε, αρχικά, ότι πάμε στο ΕΜΠ για τις ημέρες 0-i, με συνολικό κόστος cost[i]. Για να κατασκευάσουμε το υποπρόβλημα, κατασκευάζουμε το ακόλουθο DAG, οι κόμβοι του οποίου συμβολίζουν τις μέρες και οι ακμές το κόστος των εισιτηρίων, στο οποίο εξετάζουμε την ενδεικτική περίπτωση i=4:



όπου οι μπλε ακμές έχουν κόστος p1, οι κόκκινες p2 και οι πράσινες p3.

Παρατηρούμε πως, για την ημέρα 4, ισχύει ότι cost[4]=min{cost[4-k]+p[k]}, k=1,2,3. Δηλαδή, το κόστος για την τελική ημέρα είναι το ελάχιστο του κόστους όλων των πιθανών συνδυασμών p1-p1-p1, p1-p2, p3. Σημειώνουμε ότι, αν την i-οστή μέρα δεν έχουμε προγραμματίσει διαδρομή, υπολογίζουμε το cost[i-1], δηλαδή εφόσον δεν θα πάρουμε εισιτήριο μία μέρα, το κόστος είναι ίδιο με την προηγούμενη.

Με αναγωγή, καταλήγουμε ότι για κάθε ημέρα i ισχύει ο τύπος cost[i]=min{cost[i-k]+p[k]}, k<i, ο οποίος και αποτελεί τη λύση του προβλήματος.

Προκειμένου να γνωρίζουμε, μόλις ολοκληρωθεί το τρέξιμο του αλγορίθμου, τόσο το κόστος όσο και τον τύπο εισιτηρίων που χρησιμοποιήσαμε, χρησιμοποιούμε μια δομή αποθήκευσης δεδομένων για να αποθηκεύσουμε τους συνδυασμούς cost[i], c_i για την i-οστή μέρα. Επίσης, αποθηκεύουμε κάθε υποπρόβλημα i ημερών που λύνεται, ώστε να μην χρειαστεί να το επαναϋπολογίσουμε.

Όσον αφορά την πολυπλοκότητα του αλγορίθμου, χρειάζεται να διατρέξουμε τον πίνακα T ημερών αλλά και να εξετάσουμε τις τιμές k εισιτηρίων. Άρα, είναι της τάξης του O(T*k).

Άσκηση 8

Εάν την πρώτη ημέρα η πρόβλεψη d_i , i=1 δεν δείχνει την πώληση υπολογιστών τότε δεν θα παραγγείλουμε. Πιο συγκεκριμένα θα παραγγείλουμε την πρώτη ημέρα που θα έχουμε ζήτηση.

Επειδή έχουμε πρόσβαση στις προβλέψεις για όλες τις n ημέρες θα οργανώσουμε το εξής σχέδιο:

Ο αλγόριθμός μας πρέπει να εξετάζει εάν το κόστος αποθήκευσης επιπλέον ύλης για τις επόμενες μέρες είναι πιο κερδοφόρος από την εκ νέου προμήθειά της.

Οπότε θα εξετάζουμε το σενάριο αυτό και μετά θα το αποθηκεύουμε σε ένα πίνακα ώστε να γλιτώσουμε χρόνο.

Ο τρόπος σκέψης είναι ο εξής:

Έστω n = 1 :

Ημέρες (i)	*1η*
Πωλήσεις d_i	*α*

Ημέρες	*1η*
Πωλήσεις	*0*

Έστω την 1η ημέρα παραγγέλνουμε υπολογιστές . Ο αριθμός που θα παραγγείλουμε θα είναι ακριβώς ίσος με το αριθμό που προβλέπεται να πουλήσουμε.
Min.Cost(1) = K ή Min.Cost(1) = 0 εάν δεν έχουμε πωλήσεις

Έστω n = 2 :

Ημέρες (i)	*1η*	*2η*
Πωλήσεις d_i	*α*	*β*

Σε αυτήν την περίπτωση έχουμε τις εξής επιλογές :

-Πρώτη ημέρα δεν παραγγέλνουμε (δεν υπάρχουν πωλήσεις) τίποτα και δεύτερη ημέρα επίσης (δεν υπάρχουν πωλήσεις).
|-Πρώτη ημέρα παραγγέλνουμε όσο προβλέπει η d_1 :
-είτε δεν υπάρχουν πωλήσεις την δεύτερη ημέρα
-είτε το κόστος αποθήκευσης των υπολογιστών υπερβαίνει το κόστος μεταφοράς
-είτε τους έχουμε παραγγείλει μια από τις προηγούμενες ημέρες
|-Πρώτη ημέρα δεν παραγγέλνουμε και παραγγέλνουμε την δεύτερη d_2 υπολογιστές
Άρα θα ψάξουμε το ελάχιστο μεταξύ των επιλογών

*Min.Cost = min (2K , K + β*C)*

Έστω n = 3 :
Σε αυτήν την περίπτωση έχουμε:

|-Όλες τις επιλογές που έγινα για n=2:
-είτε δεν υπάρχουν πωλήσεις την Τρίτη ημέρα
-είτε το κόστος αποθήκευσης των υπολογιστών από προηγούμενες ημέρες υπερβαίνει το κόστος των
. -μεταφορικών
-είτε τους έχουμε παραγγείλει μια προηγούμενη ημέρα.

Άρα ας βγάλουμε ένα γενικό αναδρομικό τύπο:

Min.Cost(3) = Min (Min.Cost(2) + K , Min.Cost(2) + d_3C)
στο περίπου

Έτσι λοιπόν αρχίζουμε να σκεφτόμαστε πιο γενικά:

ϕ=0;

Ημέρες (i)	1η	2η	3η	4η	$5^η$	…	n-2	n-1	n
Πωλήσεις(d_i)	α	β	γ	δ	E	…	d_{n-2}	d_{n-1}	d_n

int Min.cost (i)
ϕ++
Min.cost (0) = 0;
Min.Cost(i) = Min (Min.Cost(i-1) + K , Min.Cost(i-1) + ϕ*d_i*C

Έτσι λοιπόν αναδρομικά λύνουμε το πρόβλημά μας και για να μην καθυστερεί ο αλγόριθμός μας μπορούμε να αποθηκεύουμε την κάθε λύση της Min.Cost(i) για κάθε ημέρα που βρίσκουμε.

Έτσι λύσαμε το πρόβλημα μας , αλλά έχουμε ένα bug υπάρχει περίπτωση κάποια ημέρα να μην πουλήσουμε υπολογιστή εκεί ο αλγόριθμος αυτός σε αυτήν την μορφή του θα κάνει λάθος.