



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

2^η Σειρά Ασκήσεων
Θεμελιώδη Θέματα Επιστήμης Υπολογιστών

3ο ΕΞΑΜΗΝΟ

ΔΟΝΤΗ ΕΙΡΗΝΗ

A.M.:03119839

ΑΘΗΝΑ 2020-2021

Άσκηση 1:

(α) Ο αναδρομικός αλγόριθμος για τους πύργους του Hanoi είναι ο παρακάτω:

procedure move_anoi(n from X to Y using Z);

begin

if n = 1 **then** move top disk from X to Y;

else begin

 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

end

Έστω ότι ο αριθμός κινήσεων της διαδικασίας move_anoi για $n > 1$ δίσκους είναι $A(n)$.

Για $n = 1$ δίσκο εκτελείται μία μόνο κίνηση, δηλαδή $A(1) = 1$.

Για $n > 1$ δίσκους η διαδικασία move_anoi καλεί αναδρομικά την move_anoi για μετακίνηση $n - 1$ δίσκων από τον πάσσαλο X στον Z, δηλαδή εκτελούνται $A(n-1)$ κινήσεις. Έπειτα μετακινεί τον δίσκο που απομένει από τον X στον Y, δηλαδή απαιτείται 1 επιπλέον κίνηση. Τέλος, η διαδικασία move_anoi καλεί, πάλι, αναδρομικά την move_anoi για να μετακινήσει $n - 1$ δίσκους από τον Z στον Y, δηλαδή απαιτούνται επιπλέον $A(n-1)$ κινήσεις.

Οπότε, ο αναδρομικός τύπος είναι: $A(n) = 2A(n-1) + 1$, $A(1) = 1$

Για $n = n - 1$: $A(n-1) = 2A(n-2) + 1$

Για $n = n - 2$: $A(n-2) = 2A(n-3) + 1$ κ.ο.κ.

Οπότε, λύνοντας τον τύπο προκύπτει:

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

Οπότε, ο αριθμός κινήσεων του αναδρομικού αλγορίθμου των πύργων του Hanoi για n δίσκους είναι $A(n) = 2^n - 1$.

(β) Ο επαναληπτικός αλγόριθμος για τον πύργο του Hanoi είναι ο εξής:

Επαναλάμβανε συνέχεια τα παρακάτω βήματα μέχρι να μην μπορείς να εκτελέσεις το δεύτερο βήμα:

- Μετακίνησε κατά την θετική φορά τον μικρότερο δίσκο.
- Κάνε την μοναδική επιτρεπτή κίνηση που δεν αφορά τον μικρότερο δίσκο

Έστω X ο αρχικός πάσσαλος, Z ο ενδιάμεσος και Y ο τελικός. Επιλέγοντας ως θετική φορά την φορά $X \rightarrow Y \rightarrow Z \rightarrow X$ για περιττό n και για άρτιο n επιλέγουμε την φορά $X \rightarrow Z \rightarrow Y \rightarrow X$ θα επιτύχουμε την σωστή μετάβαση από τον αρχικό πάσσαλο X στον τελικό πάσσαλο Y.

Με **επαγωγή** θα αποδείξουμε ότι ο επαναληπτικός αλγόριθμος εκτελεί τον ίδιο αριθμό κινήσεων με τον αναδρομικό. Συγκεκριμένα:

Για $n = 1$ δίσκους και οι δύο αλγόριθμοι, απαιτούν 1 κίνηση για την μετακίνηση του δίσκου.

Για n δίσκους, ο επαναληπτικός αλγόριθμος χρειάζεται $E(n) = 2^n - 1$ κινήσεις, όσες χρειάζεται και ο αναδρομικός.

Για $n + 1$ δίσκους: Οι υπόλοιποι n μικρότεροι δίσκοι πρέπει να έχουν μεταφερθεί στον Z πάσσαλο. Αυτό συμβαίνει με $2^n - 1$ κινήσεις όπως είπαμε παραπάνω. Για να μεταφερθεί ο $n + 1$ δίσκος από τον X στον Y πάσσαλο χρειάζεται μία κίνηση. Τέλος, για να μεταφερθούν οι υπόλοιποι n μικρότεροι δίσκοι από τον Z στον Y χρειάζονται άλλες $2^n - 1$ κινήσεις. Οπότε, συνολικά, χρειάζονται $2^n - 1 + 1 + 2^n - 1 = 2^{n+1} - 1$ κινήσεις.

Συμπερασματικά, λοιπόν, μπορούμε να πούμε ότι ο επαναληπτικός αλγόριθμος των Πύργων του Hanoi χρειάζεται τον ίδιο αριθμό κινήσεων με τον αναδρομικό. Δηλαδή, χρειάζονται $2^n - 1$ κινήσεις για n δίσκους.

(γ) Έστω $A(n)$ ο αριθμός κινήσεων που χρειάζεται για την μετακίνηση n δίσκων από τον πάσσαλο X στον Y , με ενδιάμεσο πάσσαλο Z . Ισχύει ότι $A(1) = 1$. Προηγουμένως αποδείξαμε ότι στον επαναληπτικό αλγόριθμο, για να μετακινήσουμε τον μεγαλύτερο δίσκο n στον τελικό πάσσαλο πρέπει, αρχικά, να έχουμε μετακινήσει τους μικρότερους $n - 1$ δίσκους στον βοηθητικό πάσσαλο. Όλοι οι μικρότεροι δίσκοι πρέπει να είναι στον ίδιο πάσσαλο Z , αλλιώς δε θα μπορεί να μετακινηθεί ο n δίσκος. Μετά μετακινούμε τον n δίσκο στον Y και ξανά τον πύργο των $n - 1$ δίσκων από πάνω του. Η διαδικασία αυτή απαιτεί τουλάχιστον $2A(n - 1) + 1$ κινήσεις. Δεν γίνεται να πραγματοποιηθεί γρηγορότερη μετακίνηση, αφού τα παραπάνω βήματα πρέπει οπωσδήποτε να γίνουν με την συγκεκριμένη σειρά. Άρα, $A(n) \geq 2A(n - 1) + 1$. Επίσης, ισχύει ότι $A(1) = 1$. Οπότε, η ισότητα στην προηγούμενη σχέση μας δίνει τον αναδρομικό τύπο που αποδείξαμε ήδη στο (α) ερώτημα.

Συμπερασματικά, ο αριθμός των κινήσεων των παραπάνω αλγορίθμων είναι ο ελάχιστος μεταξύ όλων των δυνατών αλγορίθμων για το πρόβλημα αυτό και ίσο με $A(n) = 2^n - 1$.

Άσκηση 2:

(α) Εκτελούμε πρόγραμμα στη python (έκδοση 3.8.2) που να ελέγχει αν ένας αριθμός είναι πρώτος με τον έλεγχο του **Fermat**:

```
1 from random import randint
2 #random number generator
3 def fermat(n):
4     if(n == 2 or n == 3):
5         return True
6     if(n < 2):
7         return False
8     for i in range(0,30):
9         a = randint(2, n-1)
10        if(pow(a,n-1,n) != 1):
11            #pow(a,n-1,n) is equal to
12            #pow(a,n-1)mod(a)
13            return False
14    return True
```

fermat (67280421310721)
True
fermat (170141183460469231731687303715884105721)
False
fermat (pow(2,2281)-1)
True
fermat (pow(2,9941)-1)
True
fermat (pow(2,19939)-1)
False
[]

Ο αλγόριθμος δίνει σωστό αποτέλεσμα για τις δοσμένες εισόδους, όμως αργεί να τυπώσει τις δύο τελευταίες εισόδους που του δώσαμε.

Ο αριθμός, για παράδειγμα, $2^{19939} - 1 = 2^{127 \times 157} - 1$ δεν είναι πρώτος, ενώ ο αριθμός 170141183460469231731687303715884105721 είναι πρώτος αφού γράφεται ως γινόμενο πρώτων παραγόντων.

Βάζοντας ως είσοδο μεγάλους αριθμούς Carmichael (οι οποίοι είναι σύνθετοι), προκύπτει λάθος αποτέλεσμα:

```
> fermat(461574735553)
True
> fermat(197531244744661)
True
> 
```

(β) Υλοποιούμε τον έλεγχο **Miller-Rabin** που αποτελεί βελτίωση του ελέγχου του Fermat και, έπειτα, δοκιμάζουμε με διάφορους αριθμούς Carmichael:

Εκτελούμε πρόγραμμα στη python (έκδοση 3.8.2):

```
1  from random import randint
2
3  def miller_rabin_test(d,n):
4      a=randint(2,n-2);
5      x=pow(a,d,n);
6      #x=a^d % n
7      if(x == 1 or x == n-1):
8          return True;
9
10     while(d != n-1):
11         x = pow(x,2,n);
12         d *= 2;
13
14         if(x == 1):
15             return False;
16         if(x == n-1):
17             return True;
18     return False;
19
20
21  def prime_num(n):
22     if(n <= 1 or n == 4):
23         return False;
24     if(n <= 3):
25         return True;
26     d = n - 1;
27     while(d%2 == 0):
28         d //= 2;
29
30     for i in range(0,30):
31         if (miller_rabin_test(d,n) == False):
32             return False;
33
34     return True;
```

Η συνάρτηση miller_rabin_test καλείται 30 φορές, μέσα στην συνάρτηση prime_num και επιστρέφει, για τυχαίο a , false στην περίπτωση που το n είναι ίσως πρώτος ή είναι

σύνθετος. Ο d είναι πρώτος αριθμός και για εκείνον ισχύει ότι $n - 1 = 2^s d$. Αρχικά, στη συνάρτηση `miller_rabin_test`, διαλέγεται ένας τυχαίος αριθμός a που ανήκει στο σύνολο $[2, n-2]$ και υπολογίζεται το $x = a^d \bmod n$. Αν το $x=1$ ή $x=n-1$, η συνάρτηση επιστρέφει `true`. Αντίθετα, τετραγωνίζουμε τον αριθμό x όσο:

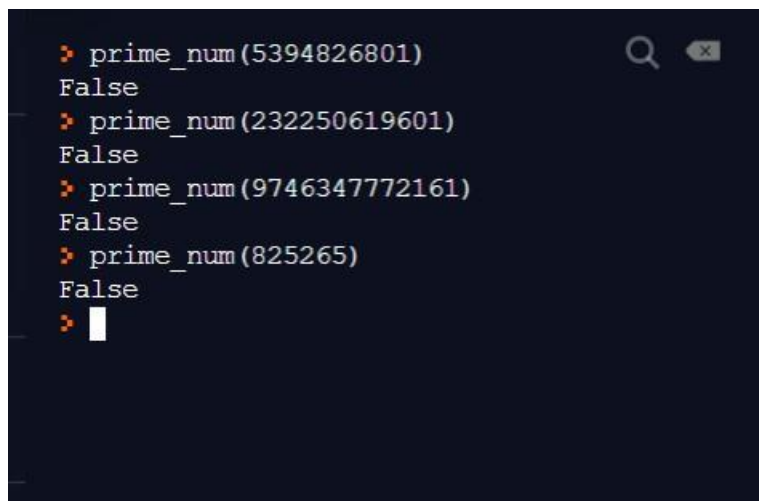
- d διάφορος του $n-1$
- $x^2 \bmod n$ διάφορος του 1 και του $n-1$

Η συνάρτηση `prime_num` επιστρέφει `true` στην περίπτωση που:

- Ο αριθμός n είναι μικρότερος ή ίσος του 3. Στην περίπτωση που ο n είναι μικρότερος ή ίσος του 1 ή ίσος με 4, επιστρέφεται `false`.
- η συνάρτηση `miller_rabin_test` είναι `true` σε όλες τις 30 διαφορετικές τιμές που δίνονται στη μεταβλητή a . Αντίθετα, επιστρέφεται `false`.

Πριν εκτελεστεί η τελευταία περίπτωση, βρίσκουμε r τέτοιο ώστε $2^d r + 1 = n$, $r \geq 1$.

Με εισόδους αριθμούς του Carmichael, παρατηρούμε ότι επιστρέφει σωστά αποτελέσματα σε σχέση με τον αλγόριθμο του Fermat:



```
> prime_num(5394826801)
False
> prime_num(232250619601)
False
> prime_num(9746347772161)
False
> prime_num(825265)
False
> 
```

Άσκηση 3:

(α) Εκτελούμε τα ζητούμενα προγράμματα στη python (έκδοση 3.8.2):

▪ **Αναδρομικός Αλγόριθμος με memoization:**

Ο συγκεκριμένος αλγόριθμος έχει πολυπλοκότητα στη χειρότερη περίπτωση $O(n \log n)$, γιατί υπολογίζει μόνο τους αριθμούς που λείπουν από τον πίνακα $a[n]$.

```

1  from array import *
2
3  def fib(n):
4      a = []
5      a = [-1 for i in range(n)];
6      a.insert(0,0);
7      a.insert(1,1);
8      if n == 0:
9          return 0;
10     if n == 1:
11         return 1;
12     if(a[n] != -1):
13         return a[n];
14     else:
15         a[n] = fib(n-1) + fib(n-2);
16         return a[n];

```

- **Επαναληπτικός Αλγόριθμος:**

Ο συγκεκριμένος αλγόριθμος έχει πολυπλοκότητα στη χειρότερη περίπτωση $O(n)$, γιατί καταχωρεί στις μεταβλητές n αριθμούς έτσι ώστε να υπολογιστεί ο n -οστός αριθμός Fibonacci.

```

1  def fib(n):
2      a=0; b=1;
3      for i in range(1,n):
4          c=b; b=a+b; a=c;
5      return b;

```

- **Αλγόριθμος με πίνακα:**

Η πολυπλοκότητα μειώνεται σε $O(n - 3) < \Theta(n)$, αφού το πλήθος των πράξεων για να υπολογιστεί ο n-οστός αριθμός Fibonacci, είναι μικρότερος από τους προηγούμενους αλγορίθμους.

```
1  from array import *
2
3  def fib(n):
4      a = []
5      a = [-1 for i in range(3)];
6
7      if n == 0:
8          return 0;
9      if n == 1:
10         return 1;
11     if n == 2:
12         return 1;
13     else:
14         a[1] = 1;
15         a[2] = 1;
16         for i in range(1,n-2):
17             a[1] = a[1]*1 + a[2]*1;
18             a[2] = a[1]*1 + a[2]*0;
19         return a[1]*1 + a[2]*1;
20
```

Τρέχοντας τους τρεις αλγορίθμους, ταυτόχρονα, με τις ίδιες τιμές, συμπεράναμε ότι ο πιο αποδοτικός αλγόριθμος, δηλαδή ο αλγόριθμος που εκτελείται και βγάζει έξοδο πιο γρήγορα, είναι ο τρίτος αλγόριθμος που είναι υλοποιημένος με πίνακα. Έπειτα, λιγότερο αποδοτικός είναι ο επαναληπτικός αλγόριθμος, ενώ ο πρώτος αλγόριθμος είναι εκείνος με την χειρότερη αποδοτικότητα. Αυτό είναι λογικό, γιατί ο αλγόριθμος με memoization έχει ως απαραίτητη προϋπόθεση τον υπολογισμό κάποιων προηγούμενων τιμών της σειράς Fibonacci. Αντίθετα, οι δύο υπόλοιποι αλγόριθμοι δεν έχουν τέτοια προϋπόθεση. Σε αυτό που διαφέρουν είναι, κυρίως, στο γεγονός ότι ο επαναληπτικός αλγόριθμος χρειάζεται να καταχωρεί, πολλές φορές, πολλές τιμές για κάθε μεταβλητή γεγονός που κοστίζει επιπλέον χρόνο από τον αλγόριθμο με πίνακα. Στον τελευταίο αλγόριθμο, το αποτέλεσμα προκύπτει πιο άμεσα από τον προηγούμενο αλγόριθμο, αφού γίνονται λιγότερες επαναλήψεις και χρειάζονται λιγότερες πράξεις για να υπολογιστεί ο n-οστός αριθμός Fibonacci.

(β*) Εκτελούμε το ζητούμενο πρόγραμμα στη python (έκδοση 3.8.2):

```
1 import math
2
3 def fib(n):
4     f = (1 + math.sqrt(5))/2;
5     fibonacci = (pow(f,n)-pow(1-f,n))/math.sqrt(5);
6     return round(fibonacci);
```

Η σχέση που συνδέει τον n-οστό αριθμό Fibonacci F_n με την χρυσή τομή φ , είναι:

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}, \text{ όπου } \varphi = \frac{1+\sqrt{5}}{2} \text{ η χρυσή τομή. Ο } F_n \text{ προκύπτει, πρακτικά από την}$$
$$\text{στρογγυλοποίηση του κλάσματος } \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}.$$

Παρατηρούμε ότι είναι ο πιο αποδοτικός αλγόριθμος που έχει ζητηθεί, αφού η πολυπλοκότητά του είναι $O(1)$ για εισόδους, σχετικά, μικρές τιμές του n .

Τρέχοντας το πρόγραμμα, παρατηρούμε ότι για μεγάλες τιμές του n , το πρόγραμμα δεν βγάζει κάποιο αποτέλεσμα, για τον λόγο ότι πρέπει να υπολογιστεί δύναμη με μεγάλο εκθετικό. Αντ' αυτού βγάζει ένα σφάλμα: 'Numerical result out of range'.

Η ιδέα του αλγορίθμου είναι πολύ αποδοτική και το σφάλμα, πιθανότατα, να οφείλεται στο γεγονός ότι η συνάρτηση `pow()` της python να μην υποστηρίζει μεγάλα εκθετικά.

(γ) Έστω $\|n\|$ ο αριθμός του αριθμού n σε bits:

Η πολυπλοκότητα ψηφιοπράξεων του **επαναληπτικού αλγορίθμου** για τον υπολογισμό αριθμών Fibonacci είναι: $O(2^{\|n\|/2} 2^{\|n\|/2}) = O(2^{\|n\|})$.

Η πολυπλοκότητα ψηφιοπράξεων του **αλγορίθμου με πίνακα** για τον υπολογισμό αριθμών Fibonacci είναι:

(α) Με πολλαπλασιασμό ακεραίων: $O((n-4)\|n\|^2) = O(n\|n\|^2)$, αφού χρειαζόμαστε $n-3$ περίπου επαναληπτικές πράξεις για να υπολογίσουμε τον απαιτούμενο πίνακα.

(β) Με πολλαπλασιασμό Gauss- Karatsuba: $O((n-3)^{\log_3\|n\|} \|n\|^{\log_3 3}) = O(3^{\|n\|} \|n\|^{\log_3 3})$, όπου το $n-3$ οι επαναλήψεις που χρειάστηκαν για τον υπολογισμό του πίνακα.

Ισχύουν τα *ακριβώς αντίστροφα* για την πολυπλοκότητα ψηφιοπράξεων, σε σχέση με την χρονική πολυπλοκότητα που εξετάσαμε στο ερώτημα (α), αφού:

$O(3^{\|n\|} \|n\|^{\log_3 3}) > O(n\|n\|^2) > O(2^{\|n\|})$ για μεγάλα n .

Άσκηση 4:

(α) Έστω ότι διαθέτουμε ένα κατευθυνόμενο γράφο με βάρη το κόστος διοδίων. Μπορούμε να χρησιμοποιήσουμε στον αλγόριθμο αυτό την **τοπολογική ταξινόμηση μέσω DFS**. Για θετικά βάρη και για γράφο χωρίς κύκλους, ο αποδοτικότερος αλγόριθμος για την εύρεση του συντομότερου μονοπατιού ελάχιστου κόστους από έναν αρχικό κόμβο s προς οποιονδήποτε άλλο κόμβο είναι ο παρακάτω:

- 1:** Αποθηκεύουμε τις ταξινομημένες κορυφές σε έναν πίνακα Π , αφού βρούμε την τοπολογική διάταξη του γράφου.
- 2:** Θέτουμε, αρχικά, $D(s) = 0$, αφού δεν πληρώνουμε την πρώτη φορά και $D(i) = \infty$ για κάθε $i \neq s$ όπου $D(i)$ είναι το κόστος της καλύτερου μονοπατιού που έχει βρεθεί μέχρι στιγμής από τον αρχικό κόμβο s έως τον κόμβο i .
- 3:** Θεωρούμε, αρχικά, $p(i) = \text{null}$ για κάθε i , όπου $p(i)$ ο προηγούμενος κόμβος του i στην καλύτερη διαδρομή που έχει βρεθεί μέχρι στιγμής.
- 4:** Αφού τελειώσουμε με την αρχικοποίηση, διατρέχουμε τους κόμβους σύμφωνα με την ταξινομημένη σειρά του πίνακα Π , αρχίζοντας από τον κόμβο s . Στην περίπτωση που ο s δεν είναι ο πρώτος κόμβος στην σειρά, τότε θεωρούμε τους προηγούμενους μη προσβάσιμους και ξεκινάμε με τον s .
- 5:** Για κάθε κόμβο u , ελέγχουμε όλους τους κόμβους στους οποίους οδηγεί ακμή που αρχίζει από τον κόμβο u . Οπότε, για κάθε τέτοιο κόμβο, αν ισχύει ότι $D(u) + w(u,v) < D(v)$, όπου $w(u,v)$ το βάρος της ακμής (u,v) . Θέτουμε $D(u) + w(u,v) = D(v)$ και $p(v) = u$. Αν το κόστος της διαδρομής προς το v μέσω του u είναι μικρότερο από το κόστος του, ως τώρα, μονοπατιού προς το v , τότε θέτουμε αυτό ως καλύτερο μονοπάτι και το u ως προηγούμενο κόμβο του v . Στην αντίθετη περίπτωση διατηρούμε τις υπάρχουσες τιμές των $D(v)$, $p(v)$.

Όταν τελειώσει ο έλεγχος για κάθε στοιχείο του πίνακα Π , η τιμή $D(i)$ αντιπροσωπεύει το ελάχιστο κόστος διαδρομής από τον αρχικό κόμβο s προς τον κόμβο i , ενώ το $p(i)$ υποδηλώνει τον προηγούμενο κόμβο του i σε αυτή τη διαδρομή και υποβοηθά στο προσδιορισμό του μονοπατιού. Οι κόμβοι με $D(i) = \infty$ ή $p(i) = \text{null}$ δεν είναι προσβάσιμοι από τον κόμβο s , γιατί συμπεριλαμβάνεται κάθε κόμβος που προηγείται του s στην τοπολογική διάταξη και στην συγκεκριμένη περίπτωση δεν είναι υποχρεωτικά μόνο εκείνοι.

Ο αλγόριθμος έχει **πολυπλοκότητα $O(n+m)$** , όπου n ο αριθμός κορυφών και m ο αριθμός ακμών του γράφου. Η τοπολογική ταξινόμηση, χρησιμοποιώντας DFS, απαιτεί επίσης χρόνο $O(n+m)$, ενώ η κάθε κορυφή ελέγχει τις γειτονικές κορυφές σε χρόνο $O(m)$.

(β) Στην περίπτωση που ο γράφος έχει κύκλους, ο παραπάνω αλγόριθμος δεν λειτουργεί ορθά. Αυτό συμβαίνει, γιατί οι γράφοι με κύκλο δεν έχουν τοπολογική διάταξη. Στην συγκεκριμένη περίπτωση, χρησιμοποιούμε τον **αλγόριθμο Dijkstra με Fibonacci Heap**, γιατί δεν χρειάζεται συγκεκριμένη σειρά για να διατρέχουμε τους κόμβους.

1: Θέτουμε όλους τους κόμβους unvisited.

2: Θέτουμε, αρχικά, $D(s) = 0$, αφού δεν πληρώνουμε την πρώτη φορά και $D(i) = \infty$ για κάθε $i \neq s$ όπου $D(i)$ είναι το κόστος της καλύτερου μονοπατιού που έχει βρεθεί μέχρι στιγμής από τον αρχικό κόμβο s έως τον κόμβο i .

3: Θεωρούμε, αρχικά, $p(i) = \text{null}$ για κάθε i , όπου $p(i)$ ο προηγούμενος κόμβος του i στην καλύτερη διαδρομή που έχει βρεθεί μέχρι στιγμής.

4: Θέτουμε τον τρέχων κόμβο ως αρχικό.

5: Για τον τρέχων κόμβο u , ελέγχουμε όλους τους unvisited γειτονικούς κόμβους στους οποίους οδηγεί ακμή που αρχίζει από τον κόμβο u . Οπότε, για κάθε γειτονικό κόμβο v αν ισχύει ότι:

$D(u) + w(u,v) < D(v)$, όπου $w(u,v)$ το βάρος της ακμής (u,v) . Θέτουμε

$D(u) + w(u,v) = D(v)$ και $p(v) = u$. Αν το κόστος της διαδρομής προς το v μέσω του u είναι μικρότερο από το κόστος του, ως τώρα, μονοπατιού προς το v , τότε θέτουμε αυτό ως καλύτερο μονοπάτι και το u ως προηγούμενο κόμβο του v . Στην αντίθετη περίπτωση διατηρούμε τις υπάρχουσες τιμές των $D(v)$, $p(v)$.

6: Μόλις ελέγξουμε όλους τους γειτονικούς κόμβους που έχουν πρόσβαση στον τρέχων κόμβο, τον θέτουμε visited.

Αν δεν υπάρχουν unvisited κόμβοι ή αν για κάθε unvisited κόμβο i ισχύει $D(i) = \infty$, τότε ο αλγόριθμος τερματίζει. Η τιμή $D(i)$ του κόμβου i , αντιπροσωπεύει το ελάχιστο κόστος διαδρομής από τον αρχικό κόμβο s προς τον κόμβο i , ενώ το $p(i)$ υποδηλώνει τον προηγούμενο κόμβο του i σε αυτή τη διαδρομή και υποβοηθά στο προσδιορισμό του μονοπατιού. Οι κόμβοι με $D(i) = \infty$ ή $p(i) = \text{null}$ δεν είναι προσβάσιμοι από τον κόμβο s .

Σε αντίθετη περίπτωση, από τους unvisited κόμβους, θέτουμε ως τρέχων τον κόμβο με την ελάχιστη $D(i)$ και επαναλαμβάνουμε τα βήματα από 2 έως το τέλος.

Η πολυπλοκότητα του αλγορίθμου ποικίλει ανάλογα με τον τρόπο υλοποίησης της ουράς προτεραιότητας των κόμβων. Η ουρά προτεραιότητας των κόμβων είναι απαραίτητη, γιατί, για να τεθεί ένας κόμβος ως τρέχων, ο κόμβος με τη μικρότερη $D(i)$ έχει προτεραιότητα εκείνη τη χρονική στιγμή. Επίσης, υποχρεωτικό είναι οι n εισαγωγές κορυφών στην ουρά, όπως και οι ελάχιστες λειτουργίες και το μέγιστο m αναβαθμίσεις στη προτεραιότητα. Οπότε, αν η ουρά υλοποιηθεί με πίνακα ή λίστα, θα χρειαστεί πολυπλοκότητα $O(n^2)$ αφού οι ελάχιστες λειτουργίες έχουν γραμμική πολυπλοκότητα $O(n)$, ενώ η αναβάθμιση $O(1)$. Αντίθετα, αν η ουρά προτεραιότητας υλοποιηθεί με binary heap, οι ελάχιστες λειτουργίες απαιτούν πολυπλοκότητα $O(1)$, ενώ οι εισαγωγές κορυφών στην ουρά και η αναβάθμιση απαιτούν πολυπλοκότητα $O(m \log n)$. Ο πιο αποδοτικός αλγόριθμος για να υλοποιηθεί η ουρά προτεραιότητας είναι ο Fibonacci heap, ο οποίος έχει **πολυπλοκότητα $O(m + n \log n)$** .

(γ) Στην συγκεκριμένη περίπτωση, έχουμε αρνητικά βάρη. Οπότε ο αλγόριθμος Dijkstra δεν είναι πάντοτε ορθός, γιατί με την μετατροπή των αρνητικών βαρών σε θετικά βάρη αυξάνει επιπλέον τα θετικά βάρη, με αποτέλεσμα να δημιουργηθεί μία ψευδαίσθηση για το ποια είναι η συντομότερη διαδρομή. Για τον λόγο αυτό, χρησιμοποιούμε τον **αλγόριθμο Bellman – Ford**.

1: Θέτουμε, αρχικά, $D(s) = 0$, αφού δεν πληρώνουμε την πρώτη φορά και $D(i) = \infty$ για κάθε $i \neq s$ όπου $D(i)$ είναι το κόστος της καλύτερου μονοπατιού που έχει βρεθεί μέχρι στιγμής από τον αρχικό κόμβο s έως τον κόμβο i .

2: Θεωρούμε, αρχικά, $p(i) = \text{null}$ για κάθε i , όπου $p(i)$ ο προηγούμενος κόμβος του i στην καλύτερη διαδρομή που έχει βρεθεί μέχρι στιγμής.

3: Εκτελούμε $n - 1$ επαναλήψεις για το βήμα 4

4: Για κάθε ακμή του γράφου (u, v) , ελέγχουμε όλους τους κόμβους στους οποίους οδηγεί ακμή που αρχίζει από τον κόμβο u . Οπότε, για κάθε τέτοιο κόμβο, αν ισχύει ότι $D(u) + w(u, v) < D(v)$, όπου $w(u, v)$ το βάρος της ακμής (u, v) . Θέτουμε $D(u) + w(u, v) = D(v)$ και $p(v) = u$. Αν το κόστος της διαδρομής προς το v μέσω του u είναι μικρότερο από το κόστος του, ως τώρα, μονοπατιού προς το v , τότε θέτουμε αυτό ως καλύτερο μονοπάτι και το u ως προηγούμενο κόμβο του v . Στην αντίθετη περίπτωση διατηρούμε τις υπάρχουσες τιμές των $D(v)$, $p(v)$.

Στο n -οστό βήμα της επανάληψης, η τιμή $D(i)$ του κόμβου i , αντιπροσωπεύει το ελάχιστο κόστος διαδρομής το πολύ n ακμών από τον αρχικό κόμβο s προς τον κόμβο i , ενώ το $p(i)$ υποδηλώνει τον προηγούμενο κόμβο του i σε αυτή τη διαδρομή και υποβοηθά στο προσδιορισμό του μονοπατιού. Οι κόμβοι με $D(i) = \infty$ ή $p(i) = \text{null}$ δεν είναι προσβάσιμοι από τον αρχικό κόμβο s με μονοπάτι n ακμών.

Η πολυπλοκότητα του αλγορίθμου είναι $O((n-1)m) = O(nm)$, αφού γίνονται $n-1$ επαναλήψεις σε m πράξεις.

Άσκηση 5:

Εφόσον υπάρχει πιθανότητα να υπάρχουν αρνητικά βάρη στο κατευθυνόμενο γράφημα $G(V, E, \ell)$ και επειδή θέλουμε αρκετά μικρή πολυπλοκότητα, θα χρησιμοποιήσουμε αλγόριθμο με DFS, διασχίζοντας όλες τις ακμές του γράφου πρώτα. Δεν μπορούμε να χρησιμοποιήσουμε τον αλγόριθμο Prim και του Kruskal, γιατί αποτυγχάνουν για κατευθυνόμενο γράφο. Ο αλγόριθμος χρησιμοποιείται για να επιβεβαιώσουμε ότι ισχύει η $\delta_k = d(u_1, u_k)$ για κάθε $k < n$ και την κατασκευή του δέντρου συντομότερων μονοπατιών.

- 1: Έστω ότι $D(i) = \infty$ για κάθε $i \neq s = 1$ όπου $D(i)$ είναι το κόστος του καλύτερου μονοπατιού που έχει βρεθεί μέχρι στιγμής από αρχικό κόμβο $s = 1$ έως τον κόμβο i .
- 2: Θεωρούμε, αρχικά, $p(i) = \text{null}$ για κάθε i , όπου $p(i)$ ο προηγούμενος κόμβος του i στην καλύτερη διαδρομή που έχει βρεθεί μέχρι στιγμής.
- 3: Εκτελούμε m επαναλήψεις για το βήμα 4
- 4: Ελέγχουμε όλες τις ακμές στις οποίες παρεμβάλλονται κορυφές που η μία κορυφή είναι ο κόμβος u_1 . Οπότε, για κάθε τέτοιο κόμβο, αν ισχύει ότι $D(u_1) + w(u_1, v) < D(v) = \delta(u_1, u_k)$ ή ισοδύναμα $w(u_1, v) < D(v) - D(u_1) = \delta(u_1, u_k) - D(u_1)$, όπου $w(u_1, v)$ το βάρος της ακμής (u_1, v) . Θέτουμε $w(u_1, v) = D(v) - D(u_1)$, και $p(v) = u_1$. Αν το κόστος της διαδρομής προς το $v = u_k$ είναι μικρότερο από το κόστος του μονοπατιού προς το $v = u_k$, τότε θέτουμε αυτό ως καλύτερο μονοπάτι και το u_1 ως προηγούμενο κόμβο του u_k . Στην αντίθετη περίπτωση διατηρούμε τις υπάρχουσες τιμές των $D(v)$, $p(v)$.

Αν ισχύει ότι $\delta(u_1, u_k) \neq D(v)$, σταματάει η επανάληψη.

Εκτελούμε DFS ώστε να διασχίσουμε τον γράφο και να δημιουργήσουμε το δέντρο συντομότερων μονοπατιών.

Η πολυπλοκότητα του αλγορίθμου δεν ξεπερνά το $\Theta(n+m)$, αφού γίνονται m επαναλήψεις στο βήμα 4, που είναι ουσιαστικά ο αριθμός των ακμών του γράφου, με τελούμενες πράξεις της τάξης του $O(1)$ την κάθε φορά, ενώ γίνονται άλλες n επαναλήψεις στο βήμα 5, ώστε να δημιουργηθεί το δέντρο συντομότερων μονοπατιών. Η DFS απαιτεί χρόνο $O(n+m)$ στην χειρότερη περίπτωση, αλλά εφόσον διασχίζονται έτσι και αλλιώς οι ακμές του γράφου πρώτα, η βοηθητική συνάρτηση μπορεί να εκτελεστεί εκεί και έπειτα να χρειαστούν άλλες n επαναλήψεις για να διασχίσουμε και τις κορυφές, όπως απαιτείται για την εκτέλεση της DFS.

Άσκηση 6:

(α) Για να υπολογίσουμε αν κάτι τέτοιο είναι εφικτό, εφαρμόσουμε μία **παραλλαγή του αλγορίθμου Prim**. Ειδικότερα, εισάγεται, με επαναλήψεις, σε ένα MST δέντρο η ακμή για την οποία ισχύει $\ell \leq L$ και δεν σχηματίζει κύκλο. Επίσης, ελέγχουμε αν υπάρχει μονοπάτι από το ορεινό θέρετρο t στη πρωτεύουσα s , το οποίο ικανοποιεί τα παραπάνω με έναν πίνακα $a[u]$.

Ο αλγόριθμος του προβλήματος αυτού φαίνεται παρακάτω:

Πρόγραμμα *Παραλλαγή Αλγορίθμου Prim*

Αρχικοποιούμε $\text{dist}(s) = 0$;

Για κάθε $v \neq s$: $\text{dist}(v) = \infty$;

Όσο όλες οι κορυφές δεν έχουν ελεγχθεί:

Πρόσθεσε $\ell \neq s$ στο MST με την μικρότερη απόσταση από αυτό;

Εάν $\text{cost}(\ell, u_i) < \text{dist}(u_i)$ και $\text{cost}(\ell, u_i) \leq L$ τότε:

$\text{dist}(u_i) = \text{cost}(\ell, u_i)$;

$a[u_i] = \ell$;

Έλεγχε την διαδρομή μεταξύ s και t , διασχίζοντας την $a[u]$;

Τέλος Προγράμματος

Η διάσχιση του πίνακα $a[u]$ απαιτεί πολυπλοκότητα $O(n)$, ενώ η πολυπλοκότητα του τροποποιημένου αλγορίθμου Prim, αν υλοποιηθεί με Fibonacci Heap, είναι $O(m+n \log n)$. Οπότε, η **συνολική πολυπλοκότητα του αλγορίθμου είναι στη χειρότερη περίπτωση $O(m + n + n \log n) = O(n \log n)$** , όπου n ο αριθμός κορυφών και m ο αριθμός ακμών του γράφου.

(β) Για να έχουμε την ζητούμενη πολυπλοκότητα, υλοποιούμε τον αλγόριθμο Kruskal στο ταξινομημένο $a[u]$ πίνακα του προηγούμενου ερωτήματος. Με τη βοήθεια του πίνακα $a[u]$, θα βρούμε το μονοπάτι ανάμεσα στους κόμβους t και s και θα επιλέξουμε την μεγαλύτερη διαδρομή ως το L .

Ο αλγόριθμος του προβλήματος αυτού φαίνεται παρακάτω:

Πρόγραμμα *Παραλλαγή Αλγορίθμου Kruskal*

Χρησιμοποίησε πίνακα $A[z]$ για αποθήκευση των ακμών, $0 \leq z \leq m$;

Υλοποίησε την μέθοδο mergesort για ταξινόμηση του πίνακα $A[z]$;

Εφάρμοσε Kruskal για τις ακμές στον ταξινομημένο πίνακα $A[z]$;

Έλεγε την διαδρομή μεταξύ s και t , διασχίζοντας την $a[u]$;

Τύπωσε το μέγιστο ℓ της διαδρομής από τον κόμβο s έως τον κόμβο t .

Τέλος Προγράμματος

Η ταξινόμηση των ακμών με την μέθοδο mergesort απαιτεί πολυπλοκότητα $O(m \log m)$, ενώ ο αλγόριθμος Kruskal $O(m)$ χάρη της ταξινόμησης των ακμών. Επίσης, η εύρεση μονοπατιού ανάμεσα στους κόμβους t και s απαιτεί πολυπλοκότητα $O(n)$ και η επιλογή της μέγιστης ακμής $O(m)$. Οπότε, για τον συνολικό αλγόριθμο έχουμε **πολυπλοκότητα $O(m \log m + m + n + m) = O(m \log m)$** .

Άσκηση 7:

Έστω κόστος $[t]$ το ελάχιστο απαιτούμενο κόστος μέχρι την ημέρα t . Ισχύει ότι κόστος $[0] = 0$, για $t = 0$. Για $t > 0$:

Αν $S[t] = 0$, δηλαδή αν δεν έρθουμε στο ΕΜΠ την t μέρα, τότε:

κόστος $[t] = \text{κόστος}[t-1]$. Αυτό συμβαίνει γιατί δεν θα πάρουμε εισιτήριο την ημέρα t , δηλαδή το κόστος εισιτηρίου είναι ο ίδιος με εκείνο της προηγούμενης μέρας.

Αν $S[t] = 1$, δηλαδή αν έρθουμε στο ΕΜΠ την t μέρα, τότε:

κόστος $[t] = \min\{\text{κόστος}[t-c_k] + p_k\}$, όπου $t > c_k$. Αυτό συμβαίνει γιατί, θα πάρουμε εισιτήριο την ημέρα t για να έρθουμε στη σχολή. Οπότε, το κόστος θα είναι ίσο με το κόστος c_k των προηγούμενων ημερών αυξημένο με το κόστος του εισιτηρίου p_k .

Παίρνουμε το ελάχιστο του αθροίσματος αυτού, γιατί χρειαζόμαστε την φθηνότερη επιλογή εισιτηρίων.

Κάθε τιμή του κόστος $[t]$, υπολογίζεται με αναδρομικό τρόπο και αποθηκεύεται για επόμενη χρήση. Για να βρούμε το συνδυασμό εισιτηρίων που δίνει το ελάχιστο

κόστος, όταν υπολογίζουμε το ελάχιστο άθροισμα, θα αποθηκεύουμε τα t , $t - c_k$ και p_k σε κάποιο πίνακα που θα τυπώνουμε στο τέλος.

Ο υπολογισμός του κόστους[t], αν έχουμε ήδη γνωστές τις τιμές κάθε στοιχείου του πίνακα πριν από την χρονική στιγμή t , έχει πολυπλοκότητα $O(k)$. Συνολικά, όμως, η **πολυπλοκότητα** θα είναι **$O(tk)$** , γιατί θα γίνουν, στη χειρότερη περίπτωση, t συγκρίσεις (άμα πάμε και τις t μέρες στη σχολή). Η πολυπλοκότητα καθορίζεται ανάλογα με τις μέρες που θα πάμε στη σχολή.