Θέματα προετοιμασίας β’ (ΚΕΦ. 11-15) στο μάθημα «ΑΛΓΟΡΙΘΜΟΙ ΚΑΙ ΠΟΛΥΠΛΟΚΟΤΗΤΑ»

Τμήμα Πληροφορικής και Τηλεπικοινωνιών, Πανεπιστήμιο Ιωαννίνων, Άρτα

τελευταία ενημέρωση: 20/1/2024

Κεφάλαιο 11 (Διαίρει και Βασίλευε)

1. Απαντήστε με ΣΩΣΤΟ/ΛΑΘΟΣ
   a. Τα προβλήματα που επιλύονται με την τεχνική διαίρει και βασίλευε τυπικά επιλύονται με τη χρήση αναδρομής.
   b. Στην τεχνική διαίρει και βασίλευε το αρχικό πρόβλημα χωρίζεται σε μικρότερα επικαλυπτόμενα υποπροβλήματα.
   c. Η χρονική πολυπλοκότητα του αλγορίθμου merge-sort είναι $O(n^2)$.
   d. Ο ταχύτερος αλγόριθμος πολλαπλασιασμού δύο n-bit ακεραίων έχει πολυπλοκότητα $O(n^2)$.
   e. Το master θεώρημα εφαρμόζεται σε αναδρομικές εξισώσεις.

Απάντηση: a (ΣΩΣΤΟ), b (ΛΑΘΟΣ), c (ΛΑΘΟΣ), d (ΛΑΘΟΣ), e (ΣΩΣΤΟ)

2. Με βάση το master θεώρημα:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

υπολογίστε την πολυπλοκότητα των ακόλουθων αναδρομικών συναρτήσεων.

   a. T(n) = 2T(n/2) + n
   b. T(n) = 7T(n/3) + n
   c. T(n) = 8T(n/2) + n^2
   d. T(n) = T(n/2) + nlogn
   e. T(n) = T(n/2) + n^2

Απάντηση:

a. a = 2, b = 2, $f(n) = n$
   Σύγκριση του $n^{log_b a} = n^{log_2 2} = n$ με το $f(n) = n$ ➔ περίπτωση 2, δλδ. το $f(n)$ είναι $\Theta(n^{log_b a} \log^0 n) = \Theta(n^{log_b a})$
   Πολυπλοκότητα: $\boldsymbol{\Theta(nlogn)}$

b. a = 7, b = 3, $f(n) = n$
   Σύγκριση του $n^{log_b a} = n^{log_3 7} = n^{1,77}$ με το $f(n) = n$ ➔ περίπτωση 1, δλδ. το $f(n)$ είναι $O(n^{log_b a - \varepsilon})$ $\boldsymbol{\Theta(n^{1,77})}$

c. a = 8, b = 2, $f(n) = n^2$
   Σύγκριση του $n^{log_b a} = n^{log_2 8} = n^3$ με το $f(n) = n^2$ ➔ περίπτωση 1, δλδ. το $f(n)$ είναι $O(n^{log_b a - \varepsilon})$
   Πολυπλοκότητα: $\boldsymbol{\Theta(n^3)}$

d. a = 1, b = 2, $f(n) = nlogn$
   Σύγκριση του $n^{log_b a} = n^{log_2 1} = n^0 = 1$ με το $f(n) = nlogn$ ➔ περίπτωση 3, δλδ. το $f(n)$ είναι $\Omega(n^{log_b a + \varepsilon})$

Επίσης, πρέπει να γίνει έλεγχος ότι υπάρχει δ<1 τέτοιο ώστε a f(n/b) <= δ f(n), δηλαδή n/2 log(n/2) <= δ n log(n).
Πολυπλοκότητα: $\boldsymbol{\Theta(nlogn)}$

e. a = 1, b = 2, $f(n) = n^2$
Σύγκριση του $n^{log_b a} = n^{log_2 1} = n$ με το $f(n) = n^2$ ➔ περίπτωση 3, δλδ. το $f(n)$ είναι $\Omega(n^{log_b a})$
Επίσης, πρέπει να γίνει έλεγχος ότι υπάρχει δ<1 τέτοιο ώστε a f(n/b) <= δ f(n), δηλαδή (n/2)^2 <= δ n^2.
Πολυπλοκότητα: $\boldsymbol{\Theta(n^2)}$

Επαλήθευση αποτελεσμάτων στο: https://www.nayuki.io/page/master-theorem-solver-javascript

3. Γράψτε μια αναδρομική συνάρτηση που να περιγράφει το χρόνο εκτέλεσης της merge sort.
Απάντηση:
T(n) = c αν n <2
T(n)=2T(n/2) + bn αν n>=2

4. Συμπληρώστε τον ακόλουθο κώδικα συγχώνευσης δύο ταξινομημένων ακολουθιών έτσι ώστε η συγχώνευση να εκτελείται σε γραμμικό χρόνο.

```
def merge(list1, list2):
    ...

list1 = [1, 3, 5, 7, 9]
list2 = [0, 2, 4, 6, 8]

merged_list = merge(list1, list2)

print ("Συγχώνευση λιστών : " + str(merged_list))
```

Απάντηση:

```
def merge(list1, list2):
    size_1 = len(list1)
    size_2 = len(list2)

    res = []
    i, j = 0, 0
    while i < size_1 and j < size_2:
        if list1[i] < list2[j]:
            res.append(list1[i])
            i += 1
        else:
            res.append(list2[j])
            j += 1

    res = res + list1[i:] + list2[j:]
    return res

list1 = [1, 3, 5, 7, 9]
list2 = [0, 2, 4, 6, 8]

merged_list = merge(list1, list2)
```

```
print ("Συγχώνευση λιστών : " + str(merged_list))
```

5. Σε ένα πρόβλημα μεγιστοποίησης δύο συναρτήσεων f1 και f2 προκύπτουν οι ακόλουθες λύσεις με κάθε ζεύγος τιμών να αναφέρει πρώτα την τιμή του f1 και στη συνέχεια την τιμή του f2: (5,6), (3,9), (4,5), (10,1), (7,2), (9,0), (0,9). Ποιες είναι οι μη κυριαρχούμενες λύσεις;
   Απάντηση: Οι μη κυριαρχούμενες λύσεις είναι: (5,6), (3,9), (10,1), (7,2)

6. Τι είναι το πρόβλημα μέγιστου συνόλου (maximaset); Τι πολυπλοκότητα έχει η επίλυσή του με διαίρει και βασίλευε;

   Απάντηση: Το πρόβλημα αφορά τον εντοπισμό των σημείων σε ένα σύνολο σημείων που ανήκουν στο λεγόμενο σύνολο μεγίστων. Ένα σημείο p ανήκει στο σύνολο μεγίστων αν δεν υπάρχει άλλο σημείο στο σύνολο με αντίστοιχες συντεταγμένες μεγαλύτερες ή ίσες των συντεταγμένων του σημείου p.
   Η πολυπλοκότητα του αλγορίθμου διαίρει και βασίλευε για την επίλυση του προβλήματος μέγιστου συνόλου είναι O(nlogn).

Κεφάλαιο 12 (Δυναμικός Προγραμματισμός)

1. Απαντήστε με ΣΩΣΤΟ/ΛΑΘΟΣ
   a. Η αποθήκευση αποτελεσμάτων υποπροβλημάτων είναι βασικό χαρακτηριστικό του δυναμικού προγραμματισμού.
   b. Ο δυναμικός προγραμματισμός δίνει λύσεις σε προβλήματα για τα οποία αρχικά φαίνεται ότι η επίλυσή τους απαιτεί εκθετικό χρόνο, σε πολύ καλύτερο χρόνο.
   c. Η ταξινόμηση με συγχώνευση (merge-sort) είναι ένας αλγόριθμος δυναμικού προγραμματισμού.
   d. Η γρήγορη ταξινόμηση είναι ένας αλγόριθμος δυναμικού προγραμματισμού.

   Απάντηση: a (ΣΩΣΤΟ), b (ΣΩΣΤΟ), c (ΛΑΘΟΣ), d (ΛΑΘΟΣ)

2. Καταγράψτε όλες τις μη κενές υποσυμβολοσειρές και όλες τις μη κενές υποακολουθίες της λέξης «ΤΑΞΗ».

   Απάντηση:
   Υποσυμβολοσειρές: "Τ", "ΤΑ", "ΤΑΞ", "ΤΑΞΗ", "Α", "ΑΞ", "ΑΞΗ", "Ξ", "ΞΗ" (9 υποσυμβολοσειρές)
   Υποακολουθίες: "Τ", "Α", "Ξ", "Η", "ΤΑ", "ΤΞ", "ΤΗ", "ΑΞ", "ΑΗ", "ΞΗ", "ΤΑΞ", "ΤΑΗ", "ΤΞΗ", "ΑΞΗ", "ΤΑΞΗ" (15 υποακολουθίες)

3. Κώδικας που υπολογίζει αναδρομικά όλες τις υποακολουθίες της συμβολοσειράς που δέχεται ως παράμετρο (BRUTE FORCE).

```
subsequences = []

def all_subsequences(string, index, c):
    if index == len(string):
        subsequences.append(c)
        return
    all_subsequences(string, index + 1, c + string[index])
    all_subsequences(string, index + 1, c)


all_subsequences("ΤΑΞΗ", 0, "")
print(subsequences)
```

4. Τι σημαίνει το χαρακτηριστικό «βελτιστότητα υποπροβλημάτων» στο δυναμικό προγραμματισμό;
   Απάντηση: Σημαίνει ότι η βέλτιστη λύση του προβλήματος μπορεί να εκφραστεί με όρους βέλτιστων λύσεων υποπροβλημάτων.

5. Αναφέρατε 4 προβλήματα που λύνονται αποδοτικά με δυναμικό προγραμματισμό.
   Απάντηση: Μέγιστη κοινή υποακολουθία, 0-1 σακίδιο, γινόμενα αλυσίδας πινάκων, προγραμματισμός εργασιών με διάρκειες και βάρη (π.χ. προγραμματισμός τηλεσκοπίων).

6. Δίνονται οι δύο συμβολοσειρές X=AGCACGTC και Y=GACCGACT. Συμπληρώστε τον ακόλουθο πίνακα με τον οποίο ο αλγόριθμος δυναμικού προγραμματισμού εντοπίζει τη μέγιστη κοινή ακολουθία ανάμεσα στις δύο συμβολοσειρές. Δίνεται ότι:
   L[i,j] = max(L[i-1,j], L[i,j-1]) αν X[i]!=Y[j]
   L[i,j] = L[i-1,j-1] + 1 αν X[i]=Y[j]

			G	A	C	C	G	A	C	T
	L	-1	0	1	2	3	4	5	6	7
	-1									
A	0									
G	1									
C	2									
A	3									
C	4									
G	5									
T	6									
C	7									

Απάντηση: Πίνακας L εντοπισμού της μέγιστης υποακολουθίας για τις συμβολοσειρές X=GACCGACT και Y=AGCACGTC.

			G	A	C	C	G	A	C	T
		-1	0	1	2	3	4	5	6	7
	-1	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1
G	1	0	↖1	1	1	1	2	2	2	2
C	2	0	↑1	1	2	2	2	2	3	3
A	3	0	1	↖2	2	2	2	3	3	3
C	4	0	1	2	↖3	←3	3	3	4	4
G	5	0	1	2	3	3	↖4	4	4	4
T	6	0	1	2	3	3	↑4	←4	4	5
C	7	0	1	2	3	4	4	4	↖5	←5

Περιγραφή διαδικασίας εξαγωγής μέγιστης κοινής υποακολουθίας από τον πίνακα L: Εκκίνηση από την κάτω δεξιά γωνία του πίνακα L. Αν X[i]=Y[j] τότε καταγραφή του κοινού χαρακτήρα και μετακίνηση στο L[i-1,j-1], αλλιώς μετακίνηση στο μεγαλύτερο από τα L[i-1,j] και L[i,j-1] (αν τα L[i-1,j] και L[i,j-1] είναι ίσα, επιλέγουμε να μετακινηθούμε στο L[i,j-1]).

Μέγιστη κοινή υποακολουθία: GACGC (μήκους 5)

			G	A	C	C	G	A	C	T
		-1	0	1	2	3	4	5	6	7
	-1	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1
G	1	0	1	1	1	1	2	2	2	2
C	2	0	1	1	2	2	2	2	3	3
A	3	0	1	2	2	2	2	3	3	3
C	4	0	1	2	3	3	3	3	4	4
G	5	0	1	2	3	3	4	4	4	4
T	6	0	1	2	3	3	4	4	4	5
C	7	0	1	2	3	4	4	4	5	5

Εικόνα 1 – Οπτικοποίηση από το https://www.cs.usfca.edu/~galles/visualization/DPLCS.html

7. Γράψτε κώδικα που να πραγματοποιεί πολλαπλασιασμό του πίνακα A (p x q) με τον πίνακα B (q x r).

Απάντηση:

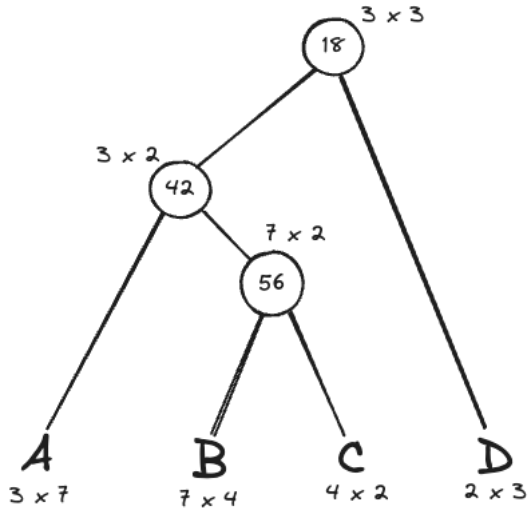
```
def matrix_multiply(matrix1, matrix2):
    p = len(matrix1)
    q = len(matrix1[0])
    q2 = len(matrix2)
    r = len(matrix2[0])

    # Έλεγχος διαστάσεων πινάκων
    if q != q2:
        raise ValueError(
            "Το πλήθος των στηλών στο matrix1 πρέπει να είναι ίσο με το πλήθος των γραμμών
στο matrix2."
        )

    #  Δημιουργία πίνακα p x r με μηδενικά στοιχεία για το αποτέλεσμα
    result = [[0] * r for _ in range(p)]

    # Πολλαπλασιασμός πινάκων
    for i in range(p):
        for j in range(r):
            for k in range(q):
                result[i][j] += matrix1[i][k] * matrix2[k][j]

    return result


# Παράδειγμα χρήσης
matrix1 = [[1, 2, 3], [4, 5, 6]]
matrix2 = [[7, 8], [9, 10], [11, 12]]
result = matrix_multiply(matrix1, matrix2)
print(result)
```

8. Δίνονται οι πίνακες A (3x7), B (7x4), C (4x2), D (2x3). Σχεδιάστε το δυαδικό δένδρο που υπολογίζει στη ρίζα του το πλήθος των πολλαπλασιασμών για το γινόμενο πινάκων με την ακόλουθη παρενθετοποίηση: ((A(BC))D). Ποιο είναι το πλήθος των πολλαπλασιασμών που θα χρειαστούν σε αυτή την περίπτωση; Πόσες διαφορετικές παρενθετοποιήσεις υπάρχουν, καταγράψτε τις.

Απάντηση:

(BC) = 7*4*2 = 56
(A(BC)) = 3*7*2 = 42
((A(BC))D) = 3*2*3 = 18

Σύνολο πολλαπλασιασμών: 56 + 42 + 18 = 116



Για 4 πίνακες, υπάρχουν 5 διαφορετικές παρενθετοποιήσεις
(A(B(CD)))
(A((B(CD))
((AB)(CD))
((A(BC))D)
(((AB)C)D)

9. Για το πρόβλημα υπολογισμού του βέλτιστου τρόπου τοποθέτησης παρενθέσεων σε μια αλυσίδα γινομένου πινάκων, συμπληρώστε τον άνω τριγωνικό πίνακα N, για το γινόμενο πινάκων A*B*C*D, όπου A (3x7), B (7x4), C (4x2), D (2x3).

N	1	2	3	4
1	0			
2		0		
3			0	
4				0

Δίνεται ότι:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

Απάντηση:
d1=3, d2=7, d3=4, d4=2, d5=3

N	1	2	3	4
1	0	84/1	98/1	116/3
2		0	56/2	96/3
3			0	40/3
4				0

N12 → i=1, j=2, k=1
N12 = N11+N22+d1*d2*d3 = 0+0+3*7*4 = 84

N23 → i=2, j=3, k=2
N23 = N22+N33+d2*d3*d4 = 0+0+7*4*2 = 56

N34 → i=3, j=4, k=3
N34 = N33+N44+d3*d4*d5 = 0+0+4*2*5 = 40

N13 → i=1, j=3, k=1,2
Για k=1: N11+N23+d1*d2*d4 = 0+56+3*7*2 = 56+42 = 98 ← min
Για k=2: N12+N33+d1*d3*d4 = 84+0+3*4*2 = 84+24 = 108

N24 → i=2, j=4, k=2,3
Για k=2: N22+N34+d2*d3*d5 = 0+40 + 7*4*3 = 40+84 = 124
Για k=3: N23+N44+d2*d4*d5 = 56+0 + 7*2*3 = 56+42 = 96 ← min

N14 → i=1, j=4, k=1,2,3
Για k=1: N11+N24+d1*d2*d5 = 0+96+3*7*3 = 96+63 = 159
Για k=2: N12+N34+d1*d3*d5 = 84+40+3*4*3 = 124+36 = 160
Για k=3: N13+N44+d1*d4*d5 = 98+0+3*2*3 = 98+18 = 116 ← min

Εξαγωγή λύσης: η τιμή k για τη λύση είναι 3, άρα προκύπτει (A*B*C)*D. Για το A*B*C, η τιμή του k είναι 1, άρα προκύπτει συνολικά (A*(B*C))*D.

10. Γράψτε μια αναδρομική και μια μη αναδρομική συνάρτηση υπολογισμού του νιοστού όρου της ακολουθίας Fibonacci.

Απάντηση:

Αναδρομική λύση	Μη αναδρομική λύση

`def fibonacci(n):`
`  if n <= 0:`
`  return 0`
`  elif n == 1:`
`  return 1`
`  else:`
`  return fibonacci(n-1) + fibonacci(n-2)`
	`def fibonacci_dynamic(n):`
	`  fib = [0] * (n+1)`
	`  fib[1] = 1`
	
	`  for i in range(2, n + 1):`
	`  fib[i] = fib[i-1] + fib[i-2]`
	
	`  return fib[n]`

11. Για το πρόβλημα 0/1 σακιδίου. Έστω η ακόλουθη λίστα 5 αντικειμένων με βάρη και τιμές και ένα σακίδιο με χωρητικότητα W=10 κιλά.

Αντικείμενο	Βάρος	Αξία
1	2	6
2	3	8
3	4	5
4	2	3
5	5	7

Συμπληρώστε τον ακόλουθο πίνακα B. Δίνεται ότι:
B[k,w]=B[k-1,w] αν $w_k > w$

B[k,w]=max(B[k-1,w], B[k-1,w-w_k]+b_k) αν w_k <= w

B	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											
5											

Απάντηση:

B	0	1	2	3	4	5	6	7	8	9	10
0	**0**	0	0	0	0	0	0	0	0	0	0
1	0	0	**6**	6	6	6	6	6	6	6	6
2	0	0	6	8	8	**14**	14	14	14	14	14
3	0	0	6	8	8	**14**	14	14	14	19	19
4	0	0	6	8	9	**14**	14	17	17	19	19
5	0	0	6	8	9	14	14	17	17	19	**21**

Σημείωση: Μπορείτε να ελέγξετε τον παραπάνω πίνακα στο https://knapsack.masao.io/

Διαδικασία εντοπισμού των αντικειμένων που επιλέγονται από τον πίνακα B:
Εκκίνηση από την κάτω δεξιά γωνία του πίνακα και ορισμός της ως τρέχουσας τιμής. Αν η τρέχουσα τιμή είναι ίση με την ακριβώς από πάνω της τότε ορισμός της τιμής αυτής ως τρέχουσας. Αν η τρέχουσα τιμή είναι διαφορετική από την ακριβώς από πάνω της τότε επιλογή του αντικειμένου της ίδια γραμμής με την τρέχουσα τιμή και μετακίνηση της θέσης του τρέχοντος στοιχείου στην προηγούμενη γραμμή μείον το βάρος του επιλεγμένου στοιχείου. Τερματισμός, όταν η τρέχουσα τιμή βρεθεί στη στήλη 0.

Για τον παραπάνω πίνακα, επιλογή των αντικειμένων 1, 2 και 5 με συνολική αξία 21.

Κεφάλαιο 13 (Γράφοι και διασχίσεις)

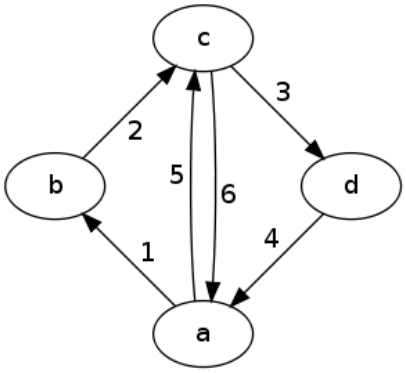
1. Απαντήστε με ΣΩΣΤΟ/ΛΑΘΟΣ
    a. Η αναπαράσταση γράφων με πίνακα γειτονικότητας προτιμάται σε αραιούς γράφους.
    b. Η πολυπλοκότητα χώρου για γράφο που αναπαρίσταται με πίνακα γειτονικότητας είναι O(V^2).
    c. Η πολυπλοκότητα χώρου για γράφο που αναπαρίσταται με λίστα γειτονικότητας είναι O(V+E).
    d. Αν ένας γράφος αναπαρίσταται με πίνακα γειτονικότητας ο εντοπισμός του εάν μια κορυφή συνδέεται απευθείας με μια άλλη κορυφή είναι ταχύτερος από ότι αν ο γράφος αναπαρίσταται με λίστα γειτονικότητας.
    e. Αν ένας γράφος αναπαρίσταται με πίνακα γειτονικότητας ο εντοπισμός των γειτονικών κορυφών μιας κορυφής είναι ταχύτερος από ότι αν ο γράφος αναπαρίσταται με λίστα γειτονικότητας.
    f. Μια απλή διαδρομή σε έναν γράφο είναι μια σειρά από κορυφές και ακμές που ξεκινά από μια κορυφή και τελειώνει σε μια κορυφή.
    g. Κάθε κύκλος σε ένα γράφο αποτελεί και μια διαδρομή.
    h. Η αναδρομική υλοποίηση του αλγορίθμου DFS (αναζήτηση πρώτα κατά βάθος) είναι εύκολη.
    i. Ο αλγόριθμος DFS (αναζήτηση πρώτα κατά βάθος) υλοποιείται με χρήση ουράς.
    j. Η πολυπλοκότητα του αλγορίθμου DFS είναι O(V+E).
    k. Οι αλγόριθμοι DFS και BFS (αναζήτηση πρώτα κατά πλάτος) έχουν την ίδια χρονική πολυπλοκότητα.
    l. Ο αλγόριθμος DFS εντοπίζει τη διαδρομή με το ελάχιστο πλήθος ακμών μεταξύ δύο κορυφών.
    m. Ο αλγόριθμος DFS και ο αλγόριθμος BFS μπορούν να χρησιμοποιηθούν για την εύρεση των συνεκτικών συνιστωσών ενός γραφήματος.
    n. Ο αλγόριθμος DFS έχει μικρότερες απαιτήσεις μνήμης από ότι ο αλγόριθμος BFS.

Απάντηση: a (ΛΑΘΟΣ), b (ΣΩΣΤΟ), c (ΣΩΣΤΟ), d (ΣΩΣΤΟ), e (ΛΑΘΟΣ), f (ΛΑΘΟΣ), g (ΣΩΣΤΟ), h (ΣΩΣΤΟ), i (ΛΑΘΟΣ), j (ΣΩΣΤΟ), k (ΣΩΣΤΟ), l (ΛΑΘΟΣ), m (ΣΩΣΤΟ), n (ΣΩΣΤΟ)

2. Με τι ισούται το άθροισμα των βαθμών (degrees) ενός μη κατευθυνόμενου γράφου;
   Απάντηση: 2*E

3. Ποιο είναι το άνω όριο για το πλήθος των ακμών σε ένα μη κατευθυνόμενο γράφο χωρίς self-loops και παράλληλες ακμές;
   Απάντηση: V*(V-1)/2

4. Δείξτε σχηματικά την αναπαράσταση του ακόλουθου γραφήματος α) με πίνακα γειτονικότητας και β) με λίστα γειτονικότητας.



Απάντηση:

α) πίνακας γειτονικότητας

	a	b	c	d
a	NULL	1	5	NULL
b	NULL	NULL	2	NULL
c	6	NULL	NULL	3
d	4	NULL	NULL	NULL

β) λίστα γειτονικότητας

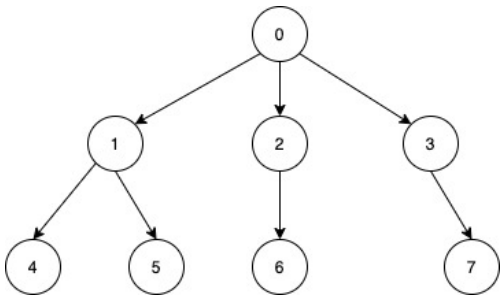
a	[(b,1), (c,5)]
b	[(c,2)]
c	[(a,6), (d,3)]
d	[(a,4)]

5. Περιγράψτε λεκτικά το πως μπορεί να χρησιμοποιηθεί ο αλγόριθμος DFS για τον εντοπισμό ενός κύκλου σε έναν κατευθυνόμενο γράφο.

   Απάντηση:

   Αρχικά όλες οι κορυφές σημειώνονται ως unvisited και επιλέγεται μια τυχαία κορυφή ως κορυφή εκκίνησης. Εφαρμογή του DFS, καθώς επισκέπτεται την κάθε κορυφή η κορυφή σημειώνεται ως visited. Αν κατά την εξερεύνηση των γειτονικών κορυφών μιας κορυφής εντοπιστεί κορυφή που είναι visited τότε υπάρχει κύκλος. Αν η διαδικασία ολοκληρωθεί και δεν έχουν επισκεφθεί όλες οι κορυφές, τότε έναρξη ξανά από μια κορυφή που δεν έχει επισκεφθεί ακόμα (για την περίπτωση που ο γράφος αποτελείται από μη συνδεδεμένα τμήματα).

6. Δίνεται ο ακόλουθος γράφος (που είναι και δέντρο). Ποια είναι η διαδρομή που επιστρέφει ο DFS και ποια η διαδρομή που επιστρέφει ο BFS;



Απάντηση:
Ο DFS επιστρέφει 0,1,4,5,2,6,3,7
Ο BFS επιστρέφει 0,1,2,3,4,5,6,7

12. Δίνεται ο ακόλουθος κώδικας που υπολογίζει τη διαδρομή που εντοπίζει ο αλγόριθμος αναζήτησης πρώτα κατά πλάτος (breadth first search) από μια κορυφή αφετηρία s προς μια κορυφή προορισμό e. Τροποποιήστε τον κώδικα έτσι ώστε να υλοποιεί τον αλγόριθμο αναζήτησης πρώτα κατά βάθος (depth first search).

```
graph = {
    "A": ["B", "C"],
    "B": ["C", "D"],
    "C": ["D"],
    "D": ["E"],
    "E": []
}

def bfs(s, e):
    """Αναζήτηση κατά πλάτος από το s στο  e"""
    frontier = deque()  # μέτωπο αναζήτησης
    frontier.append(s)
    visited = set()  # κορυφές που έχουν επισκεφτεί
    visited.add(s)
    prev = {s: None}  # για κάθε κορυφή, η προηγούμενη κορυφή
    while frontier:  # όσο το μέτωπο αναζήτησης δεν είναι κενό
        current_node = frontier.popleft()
        for next_node in graph[current_node]:
            if not next_node in visited:
                frontier.append(next_node)
                visited.add(next_node)
                prev[next_node] = current_node

    # κατασκευή του μονοπατιού από το e στο s
    path = []
    at = e
    while at != None:
        path.append(at)
        at = prev[at]

    # αντιστροφή του μονοπατιού για να προκύψει το μονοπάτι από το s στο e
    path = path[::-1]
```

```
    # αν τα s και e είναι συνδεδεμένα επιστροφή του μονοπατιού
    if path[0] == s:
        return path
    return []

bfs("A","E")
```

```
>>> ['A', 'B', 'D', 'E']
```

Απάντηση:

```
from collections import deque


def dfs(g, s, e):
    """Αναζήτηση πρώτα κατά βάθος από το s στο  e"""
    frontier = deque()  # μέτωπο αναζήτησης
    frontier.append(s)
    visited = set()  # κορυφές που έχουν επισκεφτεί
    visited.add(s)
    prev = {s: None}  # για κάθε κορυφή, η προηγούμενη κορυφή
    while frontier:  # όσο το μέτωπο αναζήτησης δεν είναι κενό
        current_node = frontier.pop()
        for next_node in g[current_node]:
            if not next_node in visited:
                frontier.append(next_node)
                visited.add(next_node)
                prev[next_node] = current_node

    # κατασκευή του μονοπατιού από το e στο s
    path = []
    at = e
    while at != None:
        path.append(at)
        at = prev[at]

    # αντιστροφή του μονοπατιού για να προκύψει το μονοπάτι από το s στο e
    path = path[::-1]

    # αν τα s και e είναι συνδεδεμένα επιστροφή του μονοπατιού
    if path[0] == s:
        return path
    return []


graph = {"A": ["B", "C"], "B": ["C", "D"], "C": ["D"], "D": ["E"], "E": []}

path = dfs(graph, "A", "E")
print("DFS:", path)
```

```
>>> DFS: ['A', 'C', 'D', 'E']
```

8. Γράψτε συνάρτηση με ορίσματα ένα κατευθυνόμενο γράφο G και μια κορυφή του γράφου s, που να επιστρέφει το μικρότερο αριθμό βημάτων (διασχίσεις ακμών) που απαιτούνται για τη μετάβαση από την κορυφή s προς όλες τις άλλες κορυφές του γράφου.

Απάντηση:

```
from collections import deque


def bfs_distance(g, s):
    """Αναζήτηση κατά πλάτος από το s στο  e"""
    frontier = deque()  # μέτωπο αναζήτησης
    frontier.append(s)
    visited = set()  # κορυφές που έχουν επισκεφτεί
    visited.add(s)
    hops = {s: 0}
    while frontier:  # όσο το μέτωπο αναζήτησης δεν είναι κενό
        current_node = frontier.popleft()
        for next_node in g[current_node]:
            if not next_node in visited:
                frontier.append(next_node)
                visited.add(next_node)
                hops[next_node] = hops[current_node] + 1
    return hops


graph = {"A": ["B", "C"], "B": ["C", "D"], "C": ["D"], "D": ["E"], "E": []}

hops = bfs_distance(graph, "A")
print("HOPS: ", hops)
```

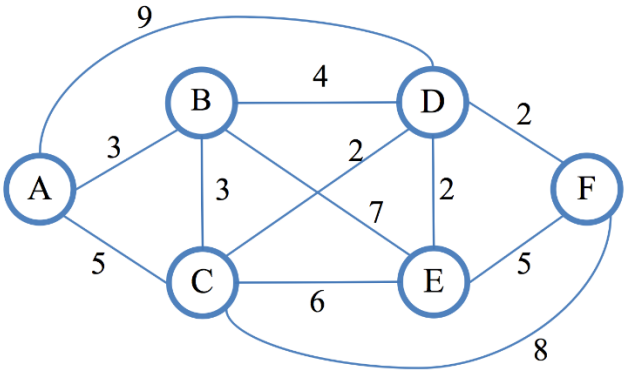
```
>>> HOPS:  {'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 3}
```

Κεφάλαιο 14 (Συντομότερες διαδρομές)

1. Απαντήστε με ΣΩΣΤΟ/ΛΑΘΟΣ
    a. Μια υποδιαδρομή της συντομότερης διαδρομής είναι η ίδια μια συντομότερη διαδρομή.
    b. Ο αλγόριθμος του Dijkstra είναι ένας «άπληστος» αλγόριθμος.
    c. Ο αλγόριθμος του Dijkstra λειτουργεί ορθά και σε γράφους με αρνητικά βάρη ακμών.
    d. Ο αλγόριθμος των Bellman-Ford λειτουργεί ορθά και σε γράφους με αρνητικά βάρη ακμών.
    e. Ο αλγόριθμος του Dijkstra μπορεί να εντοπίζει κύκλους αρνητικού βάρους σε γράφους.
    f. Η αναζήτηση των συντομότερων διαδρομών σε έναν γράφο μπορεί να επιταχυνθεί αν γνωρίζουμε ότι ο γράφος είναι DAG (Κατευθυνόμενος Ακυκλικός Γράφος).

Απάντηση: a (ΣΩΣΤΟ), b (ΣΩΣΤΟ), c (ΛΑΘΟΣ), d (ΣΩΣΤΟ), e (ΛΑΘΟΣ), f (ΣΩΣΤΟ)

2. Εφαρμόστε τον αλγόριθμο του Dijkstra για την εύρεση των συντομότερων διαδρομών στον ακόλουθο γράφο με αφετηρία την κορυφή A. Καταγράψτε για κάθε κορυφή v όλες τις τιμές που σταδιακά λαμβάνει η ετικέτα D[v] (δλδ η απόσταση της κορυφής v από την κορυφή αφετηρίας).



Απάντηση:

A	0
B	∞, 3
C	∞, 5
D	∞, 9, 7
E	∞, 10, 9
F	∞, 13, 9

3. Ποια είναι η πολυπλοκότητα του αλγορίθμου του Dijkstra;
   Απάντηση: O((V+E)log V) με ουρά προτεραιότητας (min-heap), αλλιώς O(V^2)

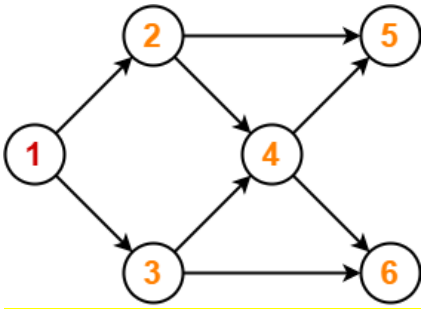
4. Ποια είναι η πολυπλοκότητα του αλγορίθμου των Bellman-Ford;
   Απάντηση: O(EV)

5. Ποια είναι η πολυπλοκότητα του αλγορίθμου των Floyd-Warshall;
   Απάντηση: O(V^3)

6. Με ποια κριτήρια επιλέγεται ο καταλληλότερος αλγόριθμος εύρεσης συντομότερων διαδρομών ανάμεσα στους αλγόριθμους: Dijkstra, Bellman-Ford, Floyd-Warshall;
   Απάντηση: Αν δεν υπάρχουν ακμές με αρνητικά βάρη και ζητείται η εύρεση συντομότερων διαδρομών από μια κορυφή προς τις άλλες κορυφές τότε ενδείκνυται η χρήση του αλγορίθμου του Dijkstra, ενώ αν υπάρχουν αρνητικά βάρη τότε ενδείκνυται η χρήση του αλγορίθμου των Bellman Ford. Αν ζητείται η εύρεση όλων των διαδρομών από οποιαδήποτε κορυφή προς οποιαδήποτε άλλη κορυφή τότε ενδείκνυται η χρήση του αλγορίθμου των Floyd-Warshall.

7. Για ποιο λόγο ενδείκνυται η χρήση ουράς προτεραιότητας στην υλοποίηση του αλγορίθμου του Dijkstra;
   Απάντηση: Ενδείκνυται η χρήση της ουράς προτεραιότητας λόγω του ότι πρέπει να επιλέγεται σε κάθε επανάληψη του αλγορίθμου η κορυφή v με την μικρότερη απόσταση D[v] που δεν έχει ακόμα ενταχθεί στις λυμένες κορυφές. Συνεπώς, χρειάζεται από ένα σύνολο που συνεχώς ενημερώνεται να εξάγεται επαναληπτικά το στοιχείο με τη μικρότερη τιμή.

8. Παραθέστε όλες τις τοπολογικές ταξινομήσεις για τον ακόλουθο γράφο.



Απάντηση:
α) 1,2,3,4,5,6
β) 1,2,3,4,6,5
γ) 1,3,2,4,5,6
δ) 1,3,2,4,6,5

Κεφάλαιο 15 (Δέντρα επικάλυψης ελάχιστου κόστους)

1. Απαντήστε με ΣΩΣΤΟ/ΛΑΘΟΣ
    a. Το ελάχιστο συνεκτικό δέντρο ενός γράφου είναι μοναδικό.
    b. Το ελάχιστο συνεκτικό δέντρο ενός γράφου είναι μοναδικό αν τα βάρη του γράφου είναι διακριτά.
    c. Το ελάχιστο συνεκτικό δέντρο ενός γράφου μπορεί να περιλαμβάνει κύκλους.
    d. Ο αλγόριθμος του Prim είναι παρόμοιος με τον αλγόριθμο του Dijkstra.
    e. Ο αλγόριθμος του Prim είναι ένας άπληστος αλγόριθμος.
    f. Ο αλγόριθμος του Kruskal ξεκινά από μια κορυφή και σταδιακά κατασκευάζει από αυτή το ελάχιστο συνεκτικό δέντρο.
    g. Ο αλγόριθμος του Boruvka έχει περισσότερες ομοιότητες με τον αλγόριθμο του Kruskal παρά με τον αλγόριθμο του Prim.
    h. Οι ακμές του συνεκτικού δέντρου όλων των συνεκτικών γραφημάτων με 20 κορυφές είναι 19.

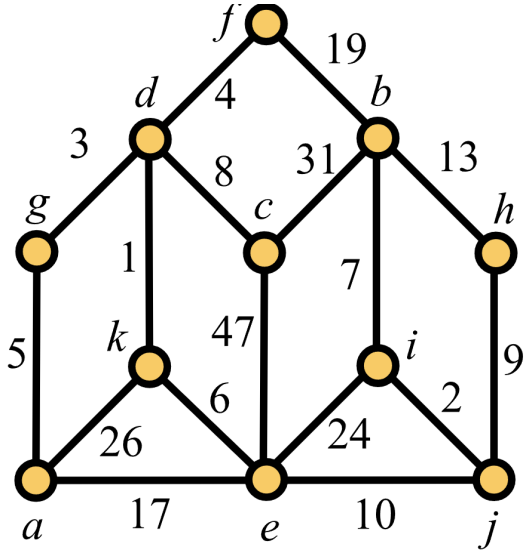
Απάντηση: a (ΛΑΘΟΣ), b (ΣΩΣΤΟ), c (ΛΑΘΟΣ), d (ΣΩΣΤΟΣ), e (ΣΩΣΤΟ), f (ΛΑΘΟΣ), g (ΣΩΣΤΟ), h (ΣΩΣΤΟ)

2. Ποια δομή δεδομένων επιταχύνει την εκτέλεση του αλγορίθμου του Kruskal εφόσον ο αλγόριθμος υλοποιηθεί κάνοντας χρήση της;
   Απάντηση:
   Ξένα σύνολα (δομή ένωσης εύρεσης)

3. Ποια είναι η πολυπλοκότητα του αλγορίθμου του Prim;
   Απάντηση:
   O((E+V) log V)

4. Ποια είναι η πολυπλοκότητα του αλγορίθμου του Kruskal;
   Απάντηση:
   O((E+V) log V)
   O(E a(V)) όπου a η αντίστροφη συνάρτηση της συνάρτησης Ackermann (πολύ αργή αύξηση) και υποθέτοντας ότι δίνεται ταξινομημένη λίστα των ακμών βάσει βαρών.

5. Υπολογίστε το ελάχιστο συνεκτικό δέντρο για τον ακόλουθο γράφο α) με τον αλγόριθμο του Prim ξεκινώντας από την κορυφή a και β) με τον αλγόριθμο του Kruskal. Και για τους δύο αλγόριθμους καταγράψτε τις ακμές που ορίζουν το ελάχιστο συνεκτικό δέντρο με τη σειρά με την οποία ο αλγόριθμος τις προσαρτά σε αυτό.



Απάντηση:

α) Αλγόριθμος του Prim

a	b	c	d	e	f	g	h	i	j	k
∞ 0	∞ 19 7	∞ 8	∞ 3	∞ 17 6	∞ 4	∞ 5	∞ 9	∞ 24 2	∞ 10	∞ 26 1

Ακμές συνεκτικού δέντρου με τη σειρά που προσαρτώνται σε αυτό: (a,g), (g,d), (d,k), (d,f), (k,e), (d,c), (e,j), (j,i), (i,b), (j,h)

β) Αλγόριθμος του Kruskal

Ακμές του συνεκτικού δέντρου με τη σειρά που προσαρτώνται σε αυτό: (d,k), (i,j), (d,g), (d,f), (a,g), (e,k), (i,b), (c,d), (j,h), (e,j)