Δομές Δεδομένων Εργασία

Λογοθέτης Φακίνος

06-2024

1 Εισαγωγή

Η παρούσα εργασία είναι ομαδική, τα ονόματα και τα AM βρίσκονται πάνω στους authors του αρχείου. Έχουμε επισυνάψει μαζί με το παρακάτω αρχείο latex, κώδικα python στον φάκελο Exercises, κάθε αρχείο τρέχει κανονικά. Υπάρχουν αρχεία που αντιστοιχούν άμεσα σε ασκήσεις και αρχεία με δομές δεδομένων που χρησιμοποιούνται για να μην επαναλαμβάνονται κάποια πράγματα πολλές φορές (LinkedList, DoubleLinkedList, ...). Όλα τα αρχεία ακόμα και οι υλοποιημένες δομές δεδομένων έχουν έτοιμες δοκιμές που θα φανούν στην κονσόλα κατόπιν εκτέλεσης του αρχείου. Βρίσκονται στο if name == main κλάδο και για αυτό πρέπει να εκτελεστούν τα ίδια (όχι με imports και άλλους έμεσσους τρόπους), για να λειτουργήσουν οι δοκιμές, οπότε μπορείτε απλά να τα εκτελέσεται δοκιμαστικά χωρίς κόπο! Γενικά στον κωδικά υπάρχουν κάποια σχόλια, αλλά υπάρχουν περισσότερα στην αντίστοιχη άσκηση στο παρών αρχείο latex. Οι περισσότερες ασκήσεις, που δέν είναι θεωρητικές, έχουν υλοποιηθεί. Κάποια κομμάτια κώδικα βρίσκονται κι εδώ.

Οι δομές δεδομένων που έχουν υλοποιηθεί (επαναχρησιμοποιούνται στις ασκήσεις) είναι οι εξής (τα ονόματα αρχείων τους ξεκινάνε με _ εκτός απο την ουρά προτεραιότητας):

1. Απλή συνδεδεμένη λίστα (Linked List) στο Exercises/_LinkedList.py
2. Διπλά συνδεδεμένη λίστα (Double Linked List) στο Exercises/_DoubleLinkedList.py
3. Ουρά υλοποιημένη με πίνακα (Queue) στο Exercises/_Queue.py
4. Στοίβα υλοποιημένη με απλή λίστα (Stack) στο Exercises/_Stack.py

5. Heap υλοποιημένο με πίνακα στο Exercises/_Heap.py
6. Ουρά προτεραιότητας υλοποιημένη με Heap (Priority Queue) στο Exercises/Ex25.py (Με ένα θεματάκι που εξηγείται στην άσκηση 25)
7. Binary Tree στο Exercises/_BinaryTree.py
8. Γράφημα με πίνακα γειτνίασης στο Exercises/_GraphAdjMat.py
9. Γράφημα με λίστα γειτνίασης στο Exercises/_GraphAdjLi.py

Μερικώς/Ολικώς υλοποιημένες ασκήσεις (σε κώδικα, υπάρχουν κι άλλες λυμένες θεωρητικού περιεχομένου όπως πχ η πρώτη):

1. Exercises/Ex6.py Άσκηση 6
2. Exercises/Ex7.py Άσκηση 7
3. Exercises/Ex9.py Άσκηση 9
4. Exercises/_LinkedList.py και Exercises/_DoubleLinkedList.py Άσκηση 8 (ώς μέθοδοι)
5. Exercises/_Queue.py Άσκηση 16 ώς κλάση
6. Exercises/Ex14.py Άσκηση 14
7. Exercises/Ex15.py Άσκηση 15
8. Exercises/Ex20.py Άσκηση 20
9. Exercises/Ex22.py Άσκηση 22
10. Exercises/Ex23.py Άσκηση 23
11. Exercises/Ex25.py Άσκηση 25
12. Exercises/Ex31.py Άσκηση 31
13. Exercises/Ex32.py Άσκηση 32
14. Exercises/Ex33.py Άσκηση 33

2 Άσκηση 1

Λύση:

Παρατηρούμε ότι οι παρακάτω συναρτήσεις είναι όλες θετικές για αρκετά μεγάλο n, έστω n_0, αυτό το n, που τις κάνει όλες θετικές.

Θα εργαστούμε πολύ με όρια, γενικά αν $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$, τότε $\forall c > 0$, $\exists n_1$, τέτοιο ώστε $\forall n \geq n_1$, $|\frac{f(n)}{g(n)}| < c$, δηλαδή $\forall n \geq \max\{n_0, n_1\}$, $f(n) < cg(n)$, άρα $f(n) = o(g(n))$ και επίσης $f(n) = O(g(n))$, κατευθείαν απο τον ορισμό του O, πχ για $c = \frac{1}{2}$.

Αν τώρα $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty$, τότε $\forall c > 0$, $\exists n_1$, τέτοιο ώστε $\forall n \geq n_1$, $|\frac{f(n)}{g(n)}| > c$, δηλαδή $\forall n \geq \max\{n_0, n_1\}$, $f(n) > cg(n)$, άρα $f(n) = \omega(g(n))$ και επίσης $f(n) = \Omega(g(n))$, παρομοίως.

Ενώ αν $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = L > 0$ και $L \in \mathbb{R}$, τότε για $\varepsilon = \frac{L}{2} > 0$, $\exists n_1$, τέτοιο ώστε $\forall n \geq n_1$, $|\frac{f(n)}{g(n)} - L| < \frac{L}{2} \implies \frac{L}{2} \leq \frac{f(n)}{g(n)} \leq \frac{3L}{2}$, άρα απο ορισμό του O, Ω, $f(n) = O(g(n))$ και $f(n) = \Omega(g(n)) \implies f(n) = \Theta(g(n))$.

Τέλος, αν $f(n) = o(g(n))$ ή $f(n) = \omega(g(n))$, τότε προφανώς $f(n) \neq \Theta(g(n))$, καθώς απο τους ορισμούς τον O, Ω, αν $f(n) = o(g(n))$, τότε $f(n) \neq \Omega(g(n))$ και αν $f(n) = \omega(g(n))$, τότε $f(n) \neq O(g(n))$.

Έτσι:

2.1 (i)

$$f(n) = n - 100, g(n) = n - 200$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n-100}{n-200} = 1$$

Άρα $f(n) = \Theta(g(n))$

2.2 (ii)

$$f(n) = n^{\frac{1}{2}}, g(n) = n^{\frac{2}{3}}$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n^{\frac{1}{2}}}{n^{\frac{2}{3}}} = \lim_{n\to+\infty} n^{-\frac{1}{6}} = 0$$

Άρα $f(n) = o(g(n))$

2.3 (iii)

$$f(n) = 100n + logn, g(n) = 10nlog(10n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{100n + logn}{10nlog(10n)} = \lim_{n\to+\infty} \frac{10}{log(10n)} + \frac{logn}{10nlog(10n)} =$$

$$0 + \lim_{n\to+\infty} \frac{log(10n) - log(10)}{10nlog(10n)} = \lim_{n\to+\infty} \frac{1}{10n} - \frac{log(10)}{10nlog(10n)} = 0$$

Άρα $f(n) = o(g(n))$

2.4 (iv)

$$f(n) = 100n + log(n), g(n) = n + log^2(n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{10n + log(n)}{n + log^2(n)} = \lim_{n\to+\infty} \frac{10 + \frac{log(n)}{n}}{1 + \frac{log^2(n)}{n}} = 10$$

Χρησιμοποιώντας τα παρακάτω όρια (υπολογισμένα με DLH αφού είναι της μορφής $\frac{\infty}{\infty}$)

$$\lim_{n\to+\infty} \frac{log(n)}{n} = \lim_{n\to+\infty} \frac{\frac{1}{n}}{1} = 0$$

$$\lim_{n\to+\infty} \frac{log^2(n)}{n} = \lim_{n\to+\infty} \frac{2log(n)}{n} = 0$$

Άρα $f(n) = \Theta(g(n))$

2.5 (v)

$$f(n) = log(2n), g(n) = log(3n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{log(n) + log(2)}{log(n) + log(3)} = 1$$

Άρα $f(n) = \Theta(g(n))$

2.6 (vi)

$$f(n) = 10log(n), g(n) = log(n^2) = 2log(n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{10log(n)}{2log(n)} = 5$$

Άρα $f(n) = \Theta(g(n))$

2.7 (vii)

$$f(n) = n^{1.01}, g(n) = nlog^2(n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n^{0.01}}{log^2(n)} = \lim_{n\to+\infty} \frac{0.01n^{-0.99}}{\frac{2log(n)}{n}} = \lim_{n\to+\infty} \frac{0.01n^{0.01}}{2log(n)} =$$

$$\lim_{n\to+\infty} \frac{0.0001n^{0.01}}{2} = +\infty$$

Άρα $f(n) = \omega(g(n))$

2.8 (ix)

$$f(n) = n^{0.1}, g(n) = log^{10}n$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n^{0.1}}{log^{10}(n)} = \lim_{n\to+\infty} \frac{0.1n^{-0.9} \cdot n}{10log^9(n)} = \lim_{n\to+\infty} \frac{0.1n^{0.1}}{10log^9(n)} = \cdots$$

$$\lim_{n\to+\infty} \frac{(0.1)^{10}n^{0.1}}{10!} = +\infty$$

Άρα $f(n) = \omega(g(n))$

2.9 (x)

$$f(n) = (log(n))^{log(n)}, g(n) = \frac{n}{log(n)}$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{(log(n))^{log(n)}}{\frac{n}{log(n)}}$$

Θέτω $u = log(n)$, τότε $\lim_{n\to+\infty} u = +\infty$

$$\lim_{u\to+\infty} \frac{u^{u+1}}{e^u} = \lim_{u\to+\infty} \frac{e^{(u+1)lnu}}{e^u} = \lim_{u\to+\infty} e^{ulnu-u+lnu} = +\infty$$

Καθώς, $\lim_{u\to+\infty} ulnu - u + lnu = \lim_{u\to+\infty} u(lnu-1) + lnu = +\infty$

Άρα $f(n) = \omega(g(n))$

2.10 (xi)

$$f(n) = n^{\frac{1}{2}}, g(n) = log^3(n)$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n^{0.5}}{log^3(n)} = \lim_{n\to+\infty} \frac{0.5n^{0.5}}{3log^2(n)} = \lim_{n\to+\infty} \frac{(0.5)^2n^{0.5}}{6log^1(n)} =$$
$$\lim_{n\to+\infty} \frac{(0.5)^2n^{0.5}}{6log^1(n)} = \lim_{n\to+\infty} \frac{(0.5)^3n^{0.5}}{6} = +\infty$$

Άρα $f(n) = \omega(g(n))$

2.11 (xii)

$$f(n) = n^{0.5}, g(n) = 5^{log(n)}$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n^{0.5}}{5^{log(n)}} = L$$

ϑέτω $u = log(n)$, και $\lim_{n\to+\infty} log(n) = +\infty$, επίσης $n = e^u$.

$$0 \leq L = \lim_{u\to+\infty} \frac{e^{0.5u}}{5^u} \leq \lim_{u\to+\infty} \frac{e^u}{5^u} = 0$$

Καθώς $e^{0.5u} \leq e^u$, $\forall u \geq 0$ και $5 > e$.

Άρα $f(n) = o(g(n))$

2.12 (xiii)

$$f(n) = n2^n, g(n) = 3^n$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n2^n}{3^n} = \lim_{n\to+\infty} \frac{ne^{nlog2}}{e^{nlog3}} = \lim_{n\to+\infty} \frac{n}{e^{n(log3-log2)}} =$$

$$\lim_{n\to+\infty} \frac{1}{(log3-log2)e^{n(log3-log2)}} = 0$$

Καθώς $log3 - log2 > 0$.

Άρα $f(n) = o(g(n))$

2.13 (xiv)

$$f(n) = 2^n, g(n) = 2^{n+1}$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{1}{2} = \frac{1}{2}$$

Άρα $f(n) = \Theta(g(n))$

2.14 (xv)

$$f(n) = n!, g(n) = 2^n$$

Για $n = 9$, με κομπιουτεράκι παρατηρούμε ότι $9! > 4^9$. Έστω ότι για $n \geq 9$, $n! > 4^n$, τότε $4n! > 4^{n+1}$, όμως αφού $n \geq 9 > 4$, $nn! = (n+1)! > 4n! > 4^{n+1}$, άρα επαγωγικά δείξαμε ότι για κάθε $n \geq 9$, $n! > 4^n$, άρα $\frac{n!}{2^n} > 2^n$, $\forall n \geq 9$. Αυτό σημαίνει ότι:

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{n!}{2^n} \geq \lim_{n\to+\infty} 2^n = +\infty \implies \lim_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty$$

Άρα $f(n) = \omega(g(n))$

2.15 (xvi)

$$f(n) = (log(n))^{log(n)}, g(n) = 2^{(log(n))^2}$$

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{(log(n))^{log(n)}}{2^{log^2(n)}}$$

θέτω $u = log(n)$, και $\lim_{n\to+\infty} log(n) = +\infty$.

$$\lim_{u\to+\infty} \frac{u^u}{2^{u^2}} = \lim_{u\to+\infty} \frac{e^{ulogu}}{e^{u^2log2}} = \lim_{u\to+\infty} e^{u(logu-ulog2)} = 0$$

Διότι $u(logu - ulog2) \to +\infty \cdot (-\infty) = -\infty$, καθώς $logu - ulog2 = u(\frac{logu}{u} - log2) \to +\infty(0 - ln2) = -\infty$

Άρα $f(n) = o(g(n))$

2.16 (xvii)

$$f(n) = \sum_{i=1}^{n} i^k, g(n) = n^{k+1}$$

$$f(n) = \sum_{i=1}^{n} i^k = \sum_{j=1}^{k} j!S(k,j)\binom{n+1}{j+1}$$

Απο διακριτά μαθηματικά, το $S(n,k)$ είναι οι αριθμοί Stirling, δηλαδή το πλήθος των διαμερίσεων του $[n]$ σε k μέρη.

$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \lim_{n\to+\infty} \frac{\sum_{j=1}^{k} j!S(k,j)\binom{n+1}{j+1}}{n^{k+1}} =$$

Παρατηρούμε ότι το πάνω άθροισμα είναι άθροισμα πολυωνύμων, των οποίων ο βαθμός καθορίζεται απο το $\binom{n+1}{j+1} = \dfrac{(n+1)!}{(n-j)!(j+1)!}$ και για κάθε $1 \le j \le k$, ο βαθμός είναι $j+1$, άρα για κάθε $j \neq k$, ο βαθμός του πολυωνύμου είναι το πολύ k και $k+1$ για $j=k$. Συνεπώς:

$$f(n) = \sum_{j=1}^{k} j!S(k,j)\binom{n+1}{j+1} = k!S(k,k)\binom{n+1}{k+1} + O(n^k) =$$

Υπάρχει ακριβώς μια διαμέριση του $[k]$ σε $[k]$ μέρη, άρα $S(k,k) = 1$

$$k!\binom{n+1}{k+1} + O(n^k) = \frac{1}{k+1}\cdot\frac{(n+1)!}{(n-k)!} = \frac{1}{k+1}\cdot(n+1)\cdots(n-k+1) + O(n^k) =$$

$$\frac{n^{k+1}}{k+1} + \frac{O(n^k)}{k+1} + O(n^k) = \frac{n^{k+1}}{k+1} + O(n^k)$$

Τελικά:

$$f(n) = \frac{n^{k+1}}{k+1} + O(n^k)$$

$$\lim_{n\to+\infty}\frac{f(n)}{n^{k+1}} = \lim_{n\to+\infty}\frac{n^{k+1}}{(k+1)n^{k+1}} + \frac{O(n^k)}{n^{k+1}} = \frac{1}{k+1} + 0 = \frac{1}{k+1} \in \mathbb{R}$$

Άρα $f(n) = \Theta(g(n))$.

3 Άσκηση 2

Έστω $f(n) = 1 + c + c^2 + \cdots + c^n$, όπου $n \in \mathbb{N}$ και $c > 0$.

(i) αν $c < 1, f(n) = \Theta(1)$,

(ii) αν $c = 1, f(n) = \Theta(n)$ και

(iii) αν $c > 1, f(n) = \Theta(c^n)$

Λύση:

3.1 (i)

Ώς γνωστόν: $f(n) = \sum_{i=0}^{n} c^i = \dfrac{c^{n+1}-1}{c-1}$, για $c \neq 1$

$$\lim_{n\to+\infty} f(n) = \lim_{n\to+\infty} \frac{c^{n+1}-1}{c-1} = \frac{1}{1-c}$$

Αφού $c < 1 \implies \lim_{n\to+\infty} c^n = 0$.

Συνεπώς για $\varepsilon = 1$, $\exists n_0$, τέτοιο ώστε $\forall n \geq n_0$:

$$|f(n) - \frac{1}{1-c}| \leq 1 \implies -1 + \frac{1}{1-c} \leq f(n) \leq 1 + \frac{1}{1-c}$$

$$\frac{c}{1-c} \leq f(n) \leq \frac{2-c}{1-c}$$

Άρα απο τον ορισμό του O και του Ω ($|f(n)| = f(n)$), $f(n) = O(\frac{2-c}{1-c}) = O(1)$ και $f(n) = \Omega(\frac{c}{1-c}) = \Omega(1)$, άρα $f(n) = \Theta(1)$.

3.2 (ii)

Αν c = 1, τότε $f(n) = \sum_{i=0}^{n} 1 = n + 1$, προφανώς τότε $f(n) = O(n)$ και $f(n) = \Omega(n) \implies f(n) = \Theta(n)$.

3.3 (ii)

Υπολογίζουμε:

$$\lim_{n\to+\infty} \frac{f(n)}{c^n} = \lim_{n\to+\infty} \frac{c^{n+1}-1}{c^{n+1}-c^n} = 1$$

Τότε για $\varepsilon = \frac{1}{2}$, $\exists n_0$, τέτοιο ώστε $\forall n \geq n_0$:

$$|\frac{f(n)}{c^n} - 1| \leq \frac{1}{2}$$

$$\frac{1}{2}c^n \leq f(n) \leq \frac{3}{2}c^n$$

Άρα απο τον ορισμό του O και του Ω, $f(n) = O(\frac{3}{2}c^n) = O(c^n)$ και $f(n) = \Omega(\frac{1}{2}c^n) = \Omega(c^n)$, άρα $f(n) = \Theta(c^n)$.

4 Άσκηση 3

Έστω (απλό κι ακατεύθυντο) γράφημα $G = (V, E)$

(i) Να βρεθεί άνω φράγμα στο πλήθος $|E|$ των ακμών που μπορεί να έχει και να αποδειχθεί η εγκυρότητά του.

(ii) Να υπολογισθεί το $\sum_{v \in V} deg_V(v)$

(iii) Να βρεθεί άνω φράγμα στο πλήθος των κύκλων που μπορεί να έχει.

Λύση:

4.1 (i)

Αφού το γράφημα είναι απλό, δεν περιέχει παράλληλες ακμές και θηλίες, επομένως κάθε ακμή είναι της μορφής $\{v_i, v_k\}$, και $\forall v_i, v_k \in V$, υπάρχει το πολύ μια ακμή που τις συνδέει. Επομένως, υπολογίζουμε το πλήθος των μη διατεταγμένων ζεύγων κορυφών το οποίο είναι ίσο με $\binom{|V|}{2} = \frac{|V|(|V|-1)}{2}$, το οποίο είναι ακριβώς το άνω φράγμα των ακμών, καθώς αν έχουμε παραπάνω απο $\binom{|V|}{2}$ ακμές, απο αρχή περιστερώνα θα υπάρχουν τουλάχιστον δύο που θα συνδέουν τις ίδιες κορυφές, κάνοντας τις παράλληλες, επομένως το γράφημα μας δεν θα είναι απλό.

4.2 (ii)

Θα υπολογίσουμε το ζητούμενο άθροισμα με τη μέθοδο της διπλομέτρησης. Θεωρούμε τα ζευγάρια (e, v), όπου $e \in E$ και $v \in e$, δηλαδή η ακμή e προσπίπτει στην κορυφή $v \in V$. Έστω n το πλήθος των ζευγαριών αυτών.

Απο τη μία έχουμε $|E|$ επιλογές για την ακμή e και κατόπιν 2 επιλογές για την κορυφή v, άρα $n = 2|E|$ (απο πολλαπλασιαστική αρχή).

Απο την άλλη για κάθε επιλογή μιας κορυφής $v \in V$ έχουμε ακριβώς $deg_V(v)$ επιλογές για την ακμή e, επομένως απο προσθετική αρχή, έχουμε $n = \sum_{v \in V} deg_V(v)$.

Συνδυάζουμε τα απο πάνω κι έχουμε: $\sum_{v \in V} deg_V(v) = 2|E|$.

4.3 (iii)

Προφανώς, το μέγιστο πλήθος κύκλων προκύπτει όταν το γράφημα είναι πλήρες, δηλαδή έχει τον μέγιστο αριθμό ακμών και κάθε κορυφή συνδέεται με κάθε άλλη, εφόσον για τυχόν γράφημα Z με κορυφές V, αν C κύκλος στο Z, τότε θα είναι και κύκλος στο πλήρες γράφημα G με κορυφές V, άρα θα βρούμε άνω φράγμα για το πλήθος κύκλων στο πλήρες γράφημα με κορυφές V, G.

Ένας κύκλος αναπαριστάται ώς ένα διατεταγμένο σύνολο κορυφών

$(v_{i_0}, v_{i_1}, v_{i_2}, \cdots, v_{i_k}, v_{i_0})$, όπου v_{i_0} η αφετηρία και οι υπόλοιπες κορυφές v_j εμφανίζονται το πολύ μια φορά σε αυτόν. Παρατηρούμε ότι κύκλοι διαφορετικού μήκους δεν μπορούν να ταυτίζονται, επομένως θα αναζητήσουμε το πλήθος των κύκλων μήκους n για φιξαρισμένο n, όπου n, το πλήθος κορυφών που εμφανίζονται σε αυτόν, χωρίς να μετρήσουμε ξανά την αφετηρία στο τέλος (δεν θα την γράφουμε απο εδώ και πέρα καθώς το γράφημα είναι πλήρες πάντα θα μπορεί να επιστρέψει ο κύκλος στην αφετηρία και δεν επηρεάζει το πλήθος των επιλογών που θα κάνουμε στη μέτρηση), άρα $2 \leq n \leq |V|$, εφόσον δεν υπάρχει κύκλος με μια κορυφή και αν υπήρχε κύκλος με περισσότερες απο $|V|$ κορυφές, μια θα εμφανιζόταν τουλάχιστον 2 φορές, απο αρχή περιστερώνα, άρα δεν θα ήταν κύκλος. Επιπλέον, πρέπει να λάβουμε υπό όψη οτι κύκλοι που προκύπτουν απο την ίδια κυκλική μετάθεση ταυτίζονται δηλαδή $(v_{i_0}, v_{i_1}, v_{i_2}, \cdots, v_{i_{n-1}}, v_{i_n}) \equiv (v_{i_n}, v_{i_0}, v_{i_1}, v_{i_2}, \cdots, v_{i_{n-1}})$.

Παρατηρούμε ότι για φιξάρισμενο n, για κάθε υποσυνόλων κορυφών του $|V|$ με n στοιχεία οι κύκλοι που μπορούν να σχηματίσουν, έρχεται σε αντιστοιχία με το πλήθος κυκλικών μεταθέσεων του $\{1, \cdots, n\}$, που είναι $(n-1)!$ σε πλήθος. Άρα για αυτό το n, υπάρχουν $\binom{|V|}{n}(n-1)!$ κύκλοι.

Απο πολλαπλασιαστική αρχή το πλήθος των κύκλων του πλήρους γραφήματος είναι:

$$\sum_{n=2}^{|V|} \binom{|V|}{n} (n-1)! = \sum_{n=2}^{|V|} \frac{|V|!}{n(|V|-n)!}$$

Το οποίο αποτελεί άνω φράγμα του πλήθους των κύκλων, όπως εξηγήσαμε προηγουμένως.

5 Άσκηση 6

Μία συμβολοσειρά $w = w_1 w_2 \cdots w_n$ ονομάζεται καρκινική, αν $w_i = w_{n-i+1}, i = 1, \cdots, n$

(i) Περιγράψτε αλγόριθμο που κατασκευάζει μία απλή συνδεδεμένη λίστα, κάθε κόμβος της οποίας περιέχει κι ένα από τα σύμβολα της w.

(ii) Περιγράψτε αλγόριθμο που ελέγχει σε μια τέτοια λίστα, αν η w είναι καρκινική. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

(iii) Μπορείτε να βελτιώσετε τον αλγόριθμο σε περίπτωση που η λίστα είναι διπλή;

(iv) Μπορείτε να βελτιώσετε τον αλγόριθμο σε περίπτωση που η λίστα είναι κυκλική;

Λύση

Αυτή η άσκηση το (i, ii, ιι) έχει υλοποιηθεί σε python, στο αρχείο Exercises/Ex6.py, το οποίο λειτουργεί κανονικά και με έτοιμες δοκιμαστικές περιπτώσεις, χρησιμοποιώντας τις υλοποίησεις Exercises/_LinkedList.py και Exercises/_DoubleLinkedList.py αντίστοιχα.

5.1 (i)

Αρχικοποιούμε μια κενή συνδεδεμένη λίστα κι απλά διατρέχουμε τους χαρακτήρες της συμβολοσειράς w, προσθέτοντας τους στο τέλος της (tail) ως κόμβους συνδεδεμένης λίστας, κατά τα γνωστά. Αποθηκεύοντας την τοποθεσία της ουράς (tail) στη μνήμη, επιτυγχάνουμε $O(n)$, χρονική πολυπλοκότητα, με $n = |w|$.

```
def stringToLinkedList(string: str) -> LinkedList:
        linkedList = LinkedList()

        linkedList.addNodesAtTail(
        [LinkedListNode(char) for char in string]
        )

        return linkedList
```

5.2 (ii)

Στον αλγόριθμο αυτό διατρέχουμε κάθε στοιχείο της συνδεδεμένης λίστας έως το κέντρο της (χωρίς το κέντρο, αν είναι περιττού μήκους) και ”ταξιδεύουμε” στο συμμετρικό της ως πρός τη λίστα το w_{n-i+1}, για να ελέγξουμε άν οι τιμές τους ταυτίζονται.

```
    def isPalindromeLinkedList(string: str):
        linkedList = stringToLinkedList(string)

        node1 = linkedList._head

        for i in range(floor(len(string) / 2)):

            node2 = node1
            for _ in range(len(string) - 2 * i - 1):
                node2 = node2._next

            if node2.value != node1.value:
                return False

            node1 = node1._next

        return True
```

Η χρονική πολυπλοκότητα (αριθμός επαναλήψεων στα for, εφόσον όλες οι συγκρίσεις/αναθέσεις είναι $O(1)$) δίνεται απο τή συνάρτηση:

$$f(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} n - 2i - 1 = n\lfloor \frac{n}{2} \rfloor - \lfloor \frac{n}{2} \rfloor(\lfloor \frac{n}{2} \rfloor + 1) - \lfloor \frac{n}{2} \rfloor$$

$$f(n) = n\lfloor \frac{n}{2} \rfloor - \lfloor \frac{n}{2} \rfloor^2 - 2\lfloor \frac{n}{2} \rfloor = \lfloor \frac{n}{2} \rfloor(n - \lfloor \frac{n}{2} \rfloor - 2)$$

Για την χρονική πολυπλοκότητα $f(n)$ ξεκάθαρα ισχύει $f(n) = O(n^2)$.

5.3 (iii)

Αν η λίστα γίνει διπλή, μπορούμε να βελτιώσουμε τον αλγόριθμο μας ως εξής:

```
def isPalindromeDoubleLinkedList(string: str) -> bool:
    doubleLinkedList = stringToDoubleLinkedList(string)

    node1 = doubleLinkedList._head
    node2 = doubleLinkedList.getTail()

    for _ in range(floor(len(string) / 2)):

        if node1.value != node2.value:
            return False

        node1 = node1._next
        node2 = node2._previous

    return True
```

Εκμεταλλευόμαστε την ικανότητα να ταξιδεύσουμε πρώς τα πίσω, ώστε να μη χρειάζεται να πάμε απο τον τρέχον κόμβο στον συμμετρικό του κόμβο, μέσω ολόκληρης της λίστας κάθε φορά. Αυτό γίνεται με τον εξής τρόπο: κάθε φορά που κάνουμε ένα βήμα μπροστά στον τρέχον κόμβο, κάνουμε κι ένα πίσω στον συμμετρικό κόμβο, έτσι σε κάθε βήμα οι δύο κόμβοι είναι συμμετρικοί (βρίσκονται στις θέσεις i, n - i + 1).

Η χρονική πολυπλοκότητα τώρα ορίζεται ώς:

$$f(n) = \lfloor \frac{n}{2} \rfloor + O(n) = O(n)$$

Καθώς θα χρειασούμε $O(n)$ χρόνο για να φτάσουμε στην ουρά και μετά $\lfloor\frac{n}{2}\rfloor$ βήματα για να κάνουμε όλες τις συγκρίσεις ($O(1)$ πολυπλοκότητας).

Άρα $f(n) = O(n)$, που είναι γραμμική!

5.4 (iv)

Αν η λίστα είναι κυκλική, χρησιμοποιούμε τον αλγόριθμο του (ii), τον οποίο δεν μπορούμε να βελτιώσουμε, καθώς η ικανότητα να πάμε απο το τελευταίο στοιχείο στο πρώτο, δεν μας βοηθάει κάπου, καθώς η επαναφορά στο τρέχον κόμβο είναι ούτως ή άλλως στιγμιαία, καθώς έχουμε αποθηκευμένη τη διεύθυνση του και η κυκλικότητα δεν μας βοηθάει στο να φτάσουμε στον συμμετρικό κόμβο πρός σύγκριση, επομένως η χρονική πολυπλοκότητα παραμένει $O(n^2)$.

6 Άσκηση 7

Έστω L μια λίστα που περιέχει το αλφαριθμητικό w_i στον i-οστό της κόμβο, $i = 1, \cdots, n$.

(i) Περιγράψτε αλγόριθμο που αντιστρέφει την σειρά των αλφαριθμητικών, ώστε τώρα ο i-οστός της κόμβος να περιέχει το w_{n-i+1}, $i = 1, \cdots, n$.

(ii) Μπορείτε να βελτιώσετε τον αλγόριθμο σε περίπτωση που η λίστα είναι διπλή;

(iii) Μπορείτε να βελτιώσετε τον αλγόριθμο σε περίπτωση που η λίστα είναι κυκλική;

Λύση:

Η παρακάτω άσκηση είναι επίσης υλοποιημένη σε python στη διεύθυνση Exercises/Ex7.py (η i και ii), χρησιμοποιώντας υλοποίησεις LinkedList και DoubleLinkedList στον ίδιο φάκελο.

6.1 (i)

Αρχικά χρησιμοποιούμε τον αλγόριθμο της άσκησης 6 (i), ώστε να μετατρέψουμε τη συμβολοσειρά μας σε απλή συνδεδεμένη λίστα, και παρομοίως με την άσκηση 6 (ii), διατρέχουμε όλους τους κόμβους εώς το κέντρο της λίστας, το συμπεριλαμβάνουμε άν είναι άρτιου μήκους (γνωρίζουμε το μήκος της απο τη συμβολοσειρά), και για καθέναν πάμε στο συμμετρικό τους ως πρός τη θέση κόμβο και ανταλλάσουμε τις τιμές τους. Ομοίως με την άσκηση 6 (ii) η πολυπλοκότητα είναι $O(n^2)$

```
def invertStringAsLinkedList(string: str):
    linkedList = stringToLinkedList(string)

    node1 = linkedList._head

    for i in range(floor(len(string) / 2)):

        node2 = node1
        for _ in range(len(string) - 2 * i - 1):
            node2 = node2._next

        swap = node1.value
        node1.value = node2.value
        node2.value = swap

        node1 = node1._next

    return linkedList
```

6.2 (ii)

Ομοίως με την άσκηση 6 (iii), εκμεταλλευόμαστε την ικανότητα να κινηθούμε πρός τα πίσω και κάνουμε τον αλγόριθμο μας γραμμικό $O(n)$, καθώς δεν χρειάζεται να ταξιδεύουμε απο κόμβο σε συμμετρικό κόμβο.

```
def invertStringAsDoubleLinkedList(string: str) -> bool:
    doubleLinkedList = stringToDoubleLinkedList(string)

    node1 = doubleLinkedList._head
    node2 = doubleLinkedList.getTail()

    for _ in range(floor(len(string) / 2)):

        swap = node1.value
```

```
        node1.value = node2.value
        node2.value = swap

        node1 = node1._next
        node2 = node2._previous

    return doubleLinkedList
```

6.3 (iii)

Ομοίως με την άσκηση 6 (iv), η ικανότητα να πάμε στιγμιαία απο την ουρά στην κεφαλή, δεν μας ωφελεί σε κάτι, καθώς δεν μπορούμε να τη χρησιμοποιήσουμε για να φτάσουμε πιο γρήγορα στον προορισμό μας.

7 Άσκηση 8

(i) Περιγράψτε αλγόριθμο που διαγράφει το (len(L)-k)-οστό στοιχείο μίας διπλής λίστας L. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

(ii) Περιγράψτε αλγόριθμο που διαγράφει το (len(L)-k)-οστό στοιχείο μίας απλής λίστας L. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

(iii) Περιγράψτε αλγόριθμο που διαγράφει το (len(L)-k)-οστό στοιχείο μίας κυκλικής λίστας L. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

Λύση:

Η άσκηση αυτή επίσης έχει υλοποιηθεί σε python (το i και ii) και βρίσκονται οι υλοποίησεις αντίστοιχα στο Exercises/_DoubleLinkedList.py και Exercises/_LinkedList.py, ενσωματωμένη στις αντίστοιχες κλάσεις των δομών δεδομένων ως μέθοδοι. Για λόγους ευκολίας στην υλοποίσηση της τωρινής άσκησης, οι αλγόριθμοι διαγράφουν το στοιχείο k, αντί για len(L) - k.

7.1 (i)

Πρώτα ελέγχουμε άν το δεδομένο index είναι 0, τότε απλά διαγράφουμε την κεφαλή, θέτοντας τον δεύτερο κόμβο ώς κεφαλή και κάνοντας τον δείκτη previous να μη δείχνει πουθενά. Αν το index είναι κάτι άλλο, αρχικοποιηούμε έναν μετρητή και προχωράμε έως ότου γίνει 1, που σημαίνει ότι φτάσαμε στον προηγούμενο κόμβο, ή να φτάσουμε στην ουρά, τότε ελέγχουμε αν το index αντιστοιχεί στην ουρά, ή κάτι μεταγενέστερο, σε αυτή τη περίπτωση επιστρέφουμε (η πετάμε OutOfBounds error), άν δεν γίνει αυτό, τότε βάζουμε τον προηγούμενο κόμβο να δείχνει στον επόμενο απο αυτόν που θέλουμε να διαγράψουμε και τον επόμενο να δείχνει στον προηγούμενο απο αυτόν που θέλουμε να διαγράψουμε, εφόσον υπάρχει. (Μπορεί να μην υπάρχει αν ο κόμβος που θέλουμε να διαγράψουμε είναι η ουρά).

Ο αλγόριθμος αυτός (όπως είναι απο κάτω), είναι γραμμικός ως πρός το μέγεθος της λίστας, $O(n)$, καθώς στη χειρότερη περίπτωση θα χρειαστεί να τη διατρέξουμε όλη μέχρι την ουρά

```
def removeAtHead(self) -> DoubleLinkedListNode:
    head = self._head
    self._head = self._head._next
    self._head._previous = None
    return head

def removeAtIndex(self, index: int) -> DoubleLinkedList:

    if index == 0:
        return self.removeAtHead()

    node = self._head

    while index > 1 and node._next is not None:
        node = node._next
        index -= 1

    if node._next is None:  # index out of range
        return None

    nodeToDelete = node._next
    node._next = nodeToDelete._next

    if node._next is not None:
        node._next._previous = node
```

```python
        return nodeToDelete
```

7.2 (ii)

Η λογική κι ο αλγόριθμος είναι ίδιοι, με τη μόνη διαφορά, ότι δεν θέτουμε τον previous δείκτη του διάδοχου κόμβου απο αυτόν που θέλουμε να διαγράψουμε στον προηγούμενο του, καθώς δεν υπάρχει δείκτης previous σε μια απλά συνδεδεμένη λίστα.

Η πολυπλοκότητα είναι $O(n)$, καθώς στη χειρότερη περίπτωση θα χρειαστεί να διατρέξουμε ολόκληρη τη λίστα.

```python
    def removeAtIndex(self, index: int) -> LinkedListNode:

        if index == 0:
            return self.removeAtHead()

        node = self._head

        while index > 1 and node._next is not None:
            node = node._next
            index -= 1

        if node._next is None:  # index out of range
            return None

        nodeToDelete = node._next
        node._next = nodeToDelete._next
        return nodeToDelete
```

7.3 (iii)

Άν η λίστα μας είναι κυκλική, τότε εφαρμόζουμε ακριβώς τον ίδιο αλγόριθμο με το υποερώτημα (ii), με τη μόνη διαφορά, ότι στην περίπτωση που χρειαστεί να διαγράψουμε την ουρά της λίστας, αντί να βάλουμε τον δείκτη του προτελευταίου κόμβου να δείχνει το τίποτα (null), τον βάζουμε να δείχνει την κεφαλή, ώστε να διατηρηθεί η κυκλικότητα της λίστας. (Καταλαβαίνουμε ότι είμαστε στην ουρά αν ο διάδοχος κόμβος απο αυτόν που θέλουμε να διαγράψουμε είναι null και επίσης το index δεν είναι out of bounds, δηλαδή index = 1, μετά απο το while loop).

Η πολυπλοκότητα παραμένει $O(n)$, για τον ίδιο λόγο.

8 Άσκηση 9

(i) Περιγράψτε αλγόριθμο που δέχεται τις κεφαλές 1, 2 δύο απλών λιστών και επιστρέφει μία απλή λίστα που ξεκινάει με τους κόμβους της H_1 και τελειώνει με αυτούς της H_2. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

(ii) Περιγράψτε αλγόριθμο που παίρνει ως είσοδο μια κυκλική λίστα C ακεραίων και την επιστρέφει, προσθέτοντας i στο i-οστό της στοιχείο, i = 1, . . . , len(C). Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του.

Λύση:

Ο πρώτος αλγόριθμος έχει υλοποιηθεί σε python στο αρχείο Exercises/Ex9.py χρησιμοποιώντας μια υλοποίηση απλής συνδεδεμένης λίστας, στο Exercises/_LinkedList.py

8.1 (i)

Πρώτα ελέγχουμε αν η κεφαλή της H_1, είναι κενή, τότε απλά επιστρέφουμε την H_2, καθώς η H_1 είναι άδεια. Αν όχι ταξιδεύουμε στην ουρά (η χρησιμοποιούμε κάποια αναφορά της ουράς), αφού φτάσουμε θέτουμε τον δείκτη της ουράς να δείχνει την κεφαλή της H_2 και κατόπιν επιστρέφουμε την H_1. Έτσι απλά έχουμε ενώσει τις δύο λίστες.

Η χρονική πολυπλοκότητα αυτής της υλοποίσης είναι $O(n_1)$, όπου n_1 το πλήθος των κόμβων της H_1. Άν "κλέψουμε" και χρησιμοποιήσουμε μια υλόποιηση απλής συνδεδεμένης λίστας που αποθηκεύει αναφορά στην ουρά μπορούμε να κάνουμε αυτόν τον αλγόριθμο $O(1)$, με κόστος δυσκολότερη υλοποίηση και πιθανόν πιο πολλά σφάλματα φυσικά.

```
def combineLinkedLists(l1: LinkedList, l2: LinkedList):
    l1Tail = l1._head

    if l1._head is None:
        return l2
```

```
    while l1Tail._next is not None:
        l1Tail = l1Tail._next

    l1Tail._next = l2._head

    return l1
```

8.2 (ii)

Προσθέτουμε 1 στην κεφαλή, εφόσον δεν είναι κενή, και ύστερα διατρέχουμε την λίστα απο κόμβο σε κόμβο, προσθέτοντας i στον καθένα, όπου i ο τρέχον μετρητής, ώσπου να ξαναφτάσουμε στην κεφαλή, αυτό θα το διαπιστώσουμε συγκρίνοντας τη διεύθυνση στη μνήμη του τρέχον κόμβου με τη διεύθυνση της κεφαλής, την οποία έχουμε αποθηκεύσει σε κάποια μεταβλητή.

Η πολυπλοκότητα σε χρόνο είναι προφανώς O(n), όπου n, το πλήθος των κόμβων της λίστας, καθώς θα πρέπει να περάσουμε απο όλους μία φορά.

9 Άσκηση 14

(i) Εξηγήστε λεπτομερώς την υλοποίηση με στοίβα του αλγορίθμου δυαδικής αναζήτησης στοιχείου σε ταξινομημένο πίνακα, για τις εισόδους:

A = [0, 2, 4, 6, 8, 10, 12, 14], key = 3,

B = [1, 2, 3, 4, 5, 6, 7, 8, 9], key = 4.

(ii) Σχεδιάστε αλγόριθμο που χρησιμοποιεί στοίβα, υλοποιημένη ως λίστα, για την δυαδική αναζήτηση

Λύση:

Ο αλγόριθμος του ερωτήματος (ii) έχει υλοποιηθεί στο αρχείο Exercises/Ex14.py με έτοιμες δοκιμές.

9.1 (i)

Θα γράφουμε την απλή συνδεδεμένη λίστα της στοίβας όπου το σημείο εξαγωγής και εισαγωγής θα είναι η κεφαλή.

9.1.1 A

A = [0, 2, 4, 6, 8, 10, 12, 14], key = 3,

Αρχικοποιούμε τη στοίβα και εισάγουμε τα αρχικά φράγματα, 0 και len(A) - 1 = 7, (πάντα πρώτα το κάτω μετά το άνω, ώστε το άνω να είναι πρώτο για εξαγωγή) άρα S = 7 $\rightarrow$ 0.

Ξεκινάμε την πρώτη επανάληψη και εξάγουμε τα φράγματα απο τη στοίβα: low = 0 και high = 7.

Υπολογίζουμε τον μέσο ώς εξής: middle = $\lfloor \frac{\text{high} + \text{low}}{2} \rfloor$, στην προκειμένη περίπτωση middle = 3.

Συγκρίνουμε τώρα το στοιχείο στη θέση 3, A[3] = 6 με το key = 3, και παρατηρούμε ότι key $<$ A[3], άρα αν υπάρχει το key βρίσκεται αριστερά απο τον μέσο, θέτουμε low = low = 0 και high = middle - 1 = 2, και τα εισάγουμε στη στοίβα, πρώτα το low μετά το high, άρα S = 2 $\rightarrow$ 0.

Ξανα εξάγουμε τα φράγματα και βρίσκουμε middle = 1, παρατηρούμε ότι A[1] = 2 $<$ 3, άρα αν υπάρχει το key είναι δεξιά του μέσου, θέτουμε low = middle + 1 = 1 + 1 = 2 και το high μένει ίδιο. Τα εισάγουμε στη στοίβα και πάμε στην επόμενη επανάληψη.

Εξάγουμε τα φράγματα, και βρίσκουμε ότι ο μέσος είναι middle = 2, όμως A[2] = 4 $>$ 3 = key, άρα θέτουμε low = 2 και high = 1, εισάγουμε στη στοίβα και πάμε στην επόμενη επανάληψη.

Εξάγουμε τα φράγματα και βλέπουμε ότι το κάτω φράγμα είναι μεγαλύτερο απο το άνω, άρα η αναζήτηση τελειώνει και το στοιχείο που αναζητάμε key δεν εμφανίζεται στο array.

9.1.2 B

B = [1, 2, 3, 4, 5, 6, 7, 8, 9], key = 4.

Όλα τα βήματα παραμένουν ίδια με πρίν, μέχρι το προτελευταίο όπου low = high = 2, δηλαδή S = 2$\rightarrow$ 2. Τότε A[2] = 4 = key, άρα επιστρέφουμε τη θέση στην οποία το βρήκαμε, την 4 και ο αλγόριθμος τελειώνει έχοντας βρεί το κλειδί στη θέση 2.

9.2 (ii)

Στον παρακάτω αλγόριθμο χρησιμοποιούμε μια στοίβα υλοποιημένη με απλή συνδεδεμένη λίστα (στο Exercises/_Stack.py), στην οποία αποθηκεύουμε το άνω και κάτω φράγμα αναζήτησης του αλγορίθμου μας. Τα οποία αρχικοποιούμε στην αρχή και το τέλος του array. Αφού, βεβαιωθούμε ότι το array δεν είναι άδειο, ξεκινάμε το while loop μας, όπου υπολογίζουμε τον μέσο που ορίζουν τα φράγματα, τα οποία εξάγουμε προηγουμένως απο την στοίβα. Ελέγχουμε αν το κάτω φράγμα δεν πέρασε το πάνω, καθώς τότε αυτό σημαίνει ότι δεν υπάρχει το key στο array, εφόσον ψάξαμε όλες τις πιθανές θέσεις και δεν το βρήκαμε. Ύστερα ελέγχουμε αν έτυχε να πέσουμε πάνω σε αυτό που ψάχναμε, τότε επιστρέφουμε τη θέση του, αν όχι ελέγχουμε αν το στοιχείο του μέσου υπερβαίνει αυτό που αναζητάμε, αν ναί τότε επειδή το array είναι αύξουσας σειράς, το κλειδί μας αποκλείεται να βρίσκεται δεξιά του μέσου και επίσης δεν είναι ο μέσος, άρα διαμορφώνουμε τα νέα άκρα του διαστήματος αναζήτησης, ως εξής: διατηρούμε το ίδιο κάτω φράγμα και θέτουμε ως άνω φράγμα τον μέσο - 1. Αντίστοιχα, στην περίπτωση που στον μέσο το στοιχείο είναι μικρότερο του κλειδιού, εργαζόμαστε συμμετρικά και ως κάτω φράγμα θέτουμε τον μέσο + 1 και διατηρούμε το ίδιο άνω φράγμα. (Σε κάποια στιγμή το άνω και κάτω φράγμα θα συμπέσουν, αν δεν βρεθεί το κλειδί ούτε εκεί, τότε στην επόμενη επανάληψη το κάτω φράγμα θα υπερβεί το άνω και θα τελειώσει η αναζήτηση)

```
def binarySearchStack(array, key) -> int:
    s = Stack()

    if len(array) == 0:
        return None

    s.push(0)
    s.push(len(array) - 1)

    while True:
        high = s.pop()
        low = s.pop()

        middle = floor((high + low) / 2)

        if high < low:
            return None
```

```
        if array[middle] == key:
            return middle

        if array[middle] > key:
            s.push(low)
            s.push(middle - 1)
        else:
            s.push(middle + 1)
            s.push(high)
```

10 Άσκηση 15

$$L(n) = L(n-1) + L(n-2),\ L(0) = 2,\ L(1) = 1$$

(i) Περιγράψτε αναδρομικό αλγόριθμο για την ακολουθία Lucas.

(ii) Τρέξτε τον αλγόριθμό σας, υλοποιημένο με στοίβα, για $n = 4$.

(iii) Σχεδιάστε αλγόριθμο που υλοποιεί στοίβα για τον υπολογισμό του $L(n)$.

Λύση:

Η λύση είναι υλοποιημένη σε python στο Exercises/Ex15.py και χρησιμοποιηούμε μια υλοποίηση Stack απο το Exercises/_Stack.py

10.1 (i)

```
def recursiveLucasAlg(n: int):

    if n == 0:
        return 2

    if n == 1:
        return 1

    return recursiveLucasAlg(n - 1) + recursiveLucasAlg(n - 2)
```

Ο οποίος είναι εκθετικός, καθώς εκτελεί τον ίδιο υπολογισμό πολλές φορές.

10.2 (iii)

```
def stackLucasAlg(n):

    stack = Stack()
    stack.push(2)

    if n > 0:
        stack.push(1)

    for _ in range(n - 1):
        cur = stack.pop()
        old = stack.pop()
        new = cur + old
        stack.push(old)
        stack.push(cur)
        stack.push(new)

    return stack.pop()
```

Σε αυτόν τον αλγόριθμο, άν το n δεν είναι 0 (τότε απλά θα μπεί το 2 μόνο του στη στοίβα και κατόπιν θα εξαχθεί και θα επιστραφεί), εισάγεται πρώτα το $L(0) = 2$ και κατόπιν το $L(1) = 1$, ύστερα με ένα Loop, υπολογίζουμε τα $L(2), L(3), \cdots, L(n)$, ώς εξής, αν βρισκόμαστε στο i βήμα, εξάγουμε απο τη στοίβα το $L(i-1)$ και κατόπιν το $L(i-2)$, υπολογίζουμε το $L(i)$ και τοποθετούμε τα $L(i-2), L(i-1), L(i)$, με αυτή τη σειρά για την επόμενη εξαγωγή, μετά απο $n-1$ βήματα (το i τρέχει απο τό 0 έως και το n - 2), καθώς στο βήμα 0, υπολογίζουμε το $L(2)$, καθώς τα προηγούμενα 2 είναι γνωστά, άρα υπολείπονται $n-2$ βήματα. Τέλος επιστρέφουμε μια εξαγωγή της στοίβας, η οποία θα περιέχει το $L(n)$ στην κορυφή της.

Αυτός ο αλγόριθμος είναι γραμμικός ως πρός το n, $O(n)$.

10.3 (ii)

```
if __name__ == "__main__":

    print(stackLucasAlg(4))
```

Τρέχουμε τη συνάρτηση που κατασκευάσαμε στο (ii) και διαπιστώνουμε, πώς $L(4) = 7$, πράγματι $L(4) = L(3) + L(2) = L(2) + L(1) + L(1) + L(0) = L(1) + L(0) + 2L(1) + L(0) = 3L(1) + 2L(0) = 3 + 4 = 7$

11 Άσκηση 16

Υπάρχει υλοποίηση σε python του (ii), η οποία βρίσκεται στο Exercises/_Queue.py

(i) Περιγράψτε αλγόριθμο που σε κάθε enqueue διορθώνει τον πίνακα κατάλληλα, ώστε να μην προκύπτει ζήτημα με πίνακα που μοιάζει γεμάτος ενώ δεν είναι.

(ii) Περιγράψτε αλγόριθμο που σε κάθε dequeue διορθώνει τον πίνακα κατάλληλα, ώστε να μην προκύπτει ζήτημα με πίνακα που μοιάζει γεμάτος ενώ δεν είναι.

Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά των αλγορίθμων σας.
Λύση:

Έστω ουρά υλοποιημένη με πίνακα, τότε θα έχουμε δύο δείκτες Q_{front}, Q_{rear}, ο Q_{front} θα είναι υπευθυνός να δείχνει την τοποθεσία στην οποία θα εξάγουμε στοιχεία (dequeue), δηλαδή θα δείχνει το μπροστά μέρος της ουράς και ο Q_{rear} θα κάνει το αντίθετο, δα δείχνει το πίσω μέρος, όπου εισάγουμε στοιχεία (enqueue). Όταν καλούμε την dequeue, θα επιστρέφεται το στοιχείο που δείχνει ο δείκτης Q_{front} και κατόπιν ο δείκτης θα μετακινείται δεξιά, όπου βρίσκεται το επόμενο στοιχεία που έχει σειρά, όμως μετά απο πολλές κλήσεις της dequeue, ο Q_{front} θα φτάσει πολύ δεξιά και έτσι ο Q_{rear}, ο οποίος είναι περιορισμένος να είναι δεξιά του Q_{front}, δεν θα έχει χώρο να βάζει καινούργια στοιχεία, και ο πίνακας θα μοιάζει γεμάτος, άρα θα αποτυγχάνει η enqueue. Για να το αντιμετωπίσουμε αυτό, πρέπει να δώσουμε μεγαλύτερη ελευθερία στους δείκτες, το οποίο και θα κάνουμε ως εξής:

Κάνουμε τον πίνακα μας ''κυκλικό'', δηλαδή με πράξεις modulo, επιτρέπουμε στους δείκτες Q_{front}, Q_{rear}, να μπορούν να πάνε απο το τέλος στην αρχή του πίνακα σε μια κίνηση. Αν αυτός είναι μεγέθους s, τότε η πράξη μετακίνησης δεξιά του δείκτη Q θα είναι $Q \leftarrow Q + 1 \mod s$, άρα πχ αν $Q = s - 1$ (τελευταίο κελί, καθώς η αρίθμηση ξεκινάει απο το 0).

11.1 (i)

Στην enqueue, ελέγχουμε αν ο δείκτης Q_{front}, είναι ακριβώς δεξιά απο τον Q_{rear}, σε αυτή την περίπτωση, δεν επιτρέπουμε το enqueue, καθώς η ουρά είναι όντως γεμάτη, αν έχουμε χώρο, τοποθετούμε το καινούργιο στοιχείο και μετακινούμε τον δείκτη Q_{rear} μια θέση δεξιά, όπως περιγράψαμε παραπάνω. $Q_{rear} \leftarrow Q_{rear} + 1 \bmod s$.

11.2 (ii)

Στην dequeue, ελέγχουμε πρώτα αν ο δείκτης Q_{front} και Q_{rear} ταυτίζονται, σε αυτή την περίπτωση εξάγουμε το στοιχείο στη θέση τους, αν υπάρχει (μπορεί να είναι κενή). Αν όχι εξάγουμε το στοιχείο στη θέση του Q_{front} και μετακινούμε τον δείκτη Q_{front} δεξιά, $Q_{front} \leftarrow Q_{front} + 1 \bmod s$.

11.3 (iii)

Με τους παραπάνω αλγορίθμους, εξασφαλίσαμε ότι ο πίνακας θα χρησιμοποιείται αποτελεσματικά. Τώρα επειδή το μόνο που κάνουμε είναι εισαγωγές, εξαγωγές (χωρίς διαγραφή του κελιού, απλά το κάνουμε κενό) και modulo πράξεις, έχουμε χρονική πολυπλοκότητα $O(1)$.

Παραθέτουμε την υλοποίηση σε python απο κάτω, η οποία βρίσκεται στο Exercises/_Queue.py

```
class Queue:

    def __init__(self, maxsize) -> None:
        self.maxsize = maxsize
        self.array = [math.nan] * maxsize
        self.qfront = 0
        self.qrear = 0
        self.length = 0

    def rightMove(self, pos: int) -> int:
        pos += 1
        if pos == self.maxsize:
            pos = 0

        return pos

    def enqueue(self, val):
        nextQrear = self.rightMove(self.qrear)
```

```
        if nextQrear == self.qfront:

            if self.array[self.qrear] is math.nan:
                self.array[self.qrear] = val
                self.length += 1

            return

        self.array[self.qrear] = val
        self.length += 1
        self.qrear = nextQrear

    def dequeue(self):
        v = self.array[self.qfront]
        self.array[self.qfront] = math.nan

        if self.qrear == self.qfront:
            if v is not math.nan:
                self.length -= 1

            return v

        self.qfront = self.rightMove(self.qfront)
        self.length -= 1
        return v

    def __str__(self) -> str:
        return str(self.array)
```

12 Άσκηση 19

(i) Δείξτε ότι ένα σχεδόν πλήρες δυαδικό δέντρο με n κορυφές, όπου στο κάτω κάτω επίπεδο οι κορυφές βρίσκονται όσο πιο αριστερά γίνεται, έχει τουλάχιστον $\frac{n}{2}$ φύλλα.

(ii) Έστω δυαδικό δέντρο $T = (V, E)$ με ρίζα, όπου κάθε κορυφή έχει είτε 0 είτε 2 παιδιά. Έστω $f(T)$ το πλήθος των κορυφών του T με δύο παιδιά και l(T) το πλήθος των φύλλων του. Δείξτε ότι $l(T) = f(T) + 1$.

Λύση:

12.1 (i)

Έστω $h = \lceil log(n) \rceil \implies log(n) \leq h < log(n) + 1$ (το δέντρο είναι δυαδικό σχεδόν πλήρες), το ύψος του δέντρου, τότε στο επίπεδο $h - 1$, εφόσον το δέντρο είναι δυαδικό σχεδόν πλήρες θα υπάρχουν 2^{h-1} κόμβοι, οι οποίοι έχουν 0-2 παιδία. Τώρα $log(n) \leq h < log(n) + 1 \implies \frac{n}{2} \leq 2^{h-1}$, δηλαδή στο επίπεδο $h - 1$, υπάρχουν τουλάχιστον $\frac{n}{2}$ κόμβοι, αν δέν έχουν παιδία, τότε είναι φύλλα, αν έχουν έστω κι ένα, τότε αυτό βρίσκεται στο επίπεδο h, άρα και πάλι είναι φύλλο, που σημαίνει ότι έχουμε τουλάχιστον $\frac{n}{2}$ φύλλα στο δέντρο μας, αφού κάθε ένας απο τους $\frac{n}{2}$ κόμβους στο επίπεδο h - 1, εξασφαλίζει την ύπαρξη ενός ξεχωριστού φύλλου (ακόμα και 2, σε περίπτωση που έχει 2 παιδιά), όπως και θέλαμε να δείξουμε.

12.2 (ii)

Αρχικά, σημειώνουμε, πως ένα τέτοιο δέντρο πρέπει να έχει περιττό πλήθος κόμβων, αφού το δέντρο είναι ριζομένο δυαδικό, τότε $\sum_{v \in V} deg_{out}v = |E| = |V| - 1$, καθώς κάθε αν μετρήσουμε τα ζευγάρια (v, e), όπου η $e \in E$, εξέρχεται απο το v, τότε για κάθε v, υπάρχουν $deg_{out}v$ τέτοιες επιλογές, άρα απο πολλαπλασιαστική αρχή το πλήθος τους είναι $\sum_{v \in V} deg_{out}v$, απο την άλλη για κάθε ακμή, υπάρχει ένας ακριβός κόμβος απο τον οποίο εξέρχεται άρα το πλήθος των ζευγαριών αυτών είναι $|E|$, όμως αφού το γράφημα είναι δέντρο ώς γνωστόν $|E| = |V| - 1$.

Όμως $deg_{out}v = 0$ ή 2, απο τον περιορισμό του δέντρου. Επομένως $\sum_{v \in V} deg_{out}v = 2k$, για κάποιο $k \in \mathbb{N}$. Άρα $|V| = 2k + 1$, συνεπώς το πλήθος κόμβων του δέντρου είναι περιττός αριθμός.

Για δένδρο με $|V| = 1$, δηλαδή ένα δέντρο μόνο με τη ρίζα του, προφανώς $l(T) = 1$ και $f(T) = 0$, άρα η ισότητα $l(T) = f(T) + 1$, ισχύει. Έστω πως ισχύει γιά όλα τα ριζομένα δυαδικά δέντρα που ικανοποιούν την προϋπόθεση που θέσαμε με $|V| = n$, με n περιττό αριθμό, θεωρούμε ένα με $|V| = n + 2$ και βρίσκουμε μια κορυφή με δύο παιδία φύλλα*, αφαιρούμε και τα δύο και έχουμε ένα υπόδεντρο που ικανοποιεί τις παραπάνω προϋποθέσεις με $|V| = n$, τότε απο επαγωγική υπόθεση, $l(T') = f(T') + 1$, όμως $l(T) = l(T') + 2$ και $f(T) = f(T') + 1$, άρα $l(T) = l(T') + 2 = f(T') + 1 + 1 = f(T) + 1$, άρα ολοκληρώνεται η επαγωγή κι έχουμε ότι για κάθε δέντρο περιττού μήκους, με κόμβους που είτε έχουν δύο είτε κανένα παιδί ισχύει η ισότητα, κι αφού αυτά τα δέντρα έχουν αναγκαστικά περιττό μήκος, ισχύει για όλα.

*Μια τέτοια κορυφή, πάντα θα υπάρχει. Έστω πρός άτοπο, ότι για κάθε κορυφή με δύο παιδιά, αυτά δεν είναι φύλλα, δηλαδή έχουν τουλάχιστον ένα παιδί, όμως λόγω του περιορισμού του δέντρου αυτά θα έχουν άλλα δύο παιδία, τα οποία δεν μπορούν να είναι φύλλα, άρα ο ίδιος συλλογισμός επαναλαμβάνετε, συνεπώς το πλήθος κόμβων αυτού του δέντρου είναι $+\infty$, εφόσον υπάρχει έστω και μία τέτοια κορυφή, άτοπο καθώς έχουμε να κάνουμε με πεπερασμένα δέντρα, εναλλακτικά δεν υπάρχει κορυφή με δύο παιδία κι άρα λόγω του περιορισμού του δέντρου, όλες οι κορυφές δεν έχουν παιδία, άρα βρισκόμαστε στην περίπτωση $|V| = 1$, η οποία όπως είδαμε πληρεί τις προϋποθέσεις, άρα χωρίς βλάβη της γενικότητας μια τέτοια κορυφή υπάρχει σίγουρα.

13 Άσκηση 20

Να γραφούν αλγόριθμοι που, σε ένα δέντρο υλοποιημένο με συνδεδεμένη λίστα, υπολογίζει:

1. το πλήθος των φύλλων του δέντρου,
2. το πλήθος των εσωτερικών κόμβων του δέντρου,
3. το ύψος του δέντρου.

Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά των αλγορίθμων σας.

Λύση:

Στο αρχείο Exercises/Ex20.py βρίσκεται η υλοποίηση σε python, με έτοιμο test case. Χρησιμοποιούμε μια υλοποίηση Binary Tree, που βρίσκεται στο Exercises/_BinaryTree.py

13.1 (1)

Αναδρομικά, θα υπολογίσουμε το πλήθος των φύλλων του δέντρου, αθροίζοντας το πλήθος των φύλλων του αρίστερου και δεξιού υπόδεντρου. Ως base cases (δηλαδή ως το πιο μικρό υπόδεντρο που μπορούμε να σπάσουμε αναδρομικά), θα υπολογίζουμε 0 φύλλα αν αυτο είναι κενό και 1 αν αυτό είναι ένας κόμβος χωρίς ακμές που εξέρχονται, δηλαδή φύλλο. Δεν προσθέτουμε τίποτα περισσότερο οταν καλούμε τις δύο αναδρομές, καθώς η τρέχουσα ρίζα δεν είναι φύλλο εφόσον έχει μη κενό αριστερό ή/και δεξί υπόδεντρο.

```
def leafCount(T: BinaryTree):

    if T.root is None:
        return 0

    if T.root.isLeaf():
        return 1

    return (leafCount(BinaryTree(T.root.right)) +
    leafCount(BinaryTree(T.root.left)))
```

13.2 (2)

Αναδρομικά, θα υπολογίσουμε το ύψος του δέντρου, βρίσκοντας το ύψος του αριστερόυ και δεξιού υπόδεντρου και παίρνοντας το μεγαλύτερο εκ των δύο, προσθέτοντας 1, για τον κόμβο απο τον οποίο έπονται τα δύο υπόδεντρα. Επιστρέφουμε μηδενικό ύψος 0, άν φτάσουμε σε κενό υπόδεντρο και 1 αν φτάσουμε σε μοναχικό κόμβο, δηλαδή φύλλο, ως base cases.

```
def height(T: BinaryTree):

    if T.root is None:
        return 0

    if T.root.isLeaf():
        return 1

    return (max([height(BinaryTree(T.root.right)),
```

```
    height(BinaryTree(T.root.left))]) + 1)
```

13.3 (3)

Αναδρομικά, υπολογίζουμε το πλήθος των εσωτερικών κόμβων, αθροίζοντας το πλήθος εσωτερικών κόμβων του δεξιού κι αριστερού υπόδεντρου, προσθέτοντας 1 για την τρέχουσα ρίζα(εφόσον δεν αποτελεί την παρακάτω περίπτωση), καθώς ο κόμβος απο τον οποίο εξέρχονται τα υπόδεντρα είναι εσωτερικός. Αν φτάσουμε σε υπόδεντρο που είτε είναι κενό ή αποτελείται απο έναν κόμβο, φύλλο, επιστρέφουμε 0, καθώς δεν έχει κανένα εσωτερικό κόμβο.

```
def innerNodes(T: BinaryTree):

    if T.root is None or T.root.isLeaf():
        return 0

    return (innerNodes(BinaryTree(T.root.right)) +
    innerNodes(BinaryTree(T.root.left)) + 1)
```

13.4 (4)

Και οι 3 αλγόριθμοι έχουν την ίδια χρονική πολυπλοκότητα την οποία θα υπολογίσουμε, καθώς λειτουργόυν πρακτικά με την ίδια λογική, με μικρές διαφοροποίησεις. Αρχικά για να ενώσουμε τα αποτελέσματα αναδρομικών κλήσεων απαιτείται δουλεία της τάξης $O(1)$, εφόσον κάνουμε απλά λίγες συγκρίσεις και προσθέσεις, τώρα όσο για το πλήθος των αναδρομικών κλήσεων. Παρατηρούμε ότι αναδρομικά ο αλγόριθμος μας θα περάσει απο κάθε κορυφή ακριβώς μια φορά, μπορεί να περάσει και απο ανύπαρκτες κορυφές (null) το πολύ n φορές στη χειρότερη περίπτωση που το δέντρο είναι περίπατος (ευθεία) αλλά ασυμπτωτικά το 2n είναι το ίδιο με το n, επομένως συμπεραίνουμε ότι έχουμε πολυπλοκότητα $O(n \cdot 1) = O(n)$, γραμμική.

14 Άσκηση 22

Σχεδιάστε αλγόριθμο που, με είσοδο ένα δέντρο αναζήτησης και τις διευθύνσεις δύο κορυφών του δέντρου, επιστρέφει το μικρότερο δυνατό υποδέντρο που περιέχει και τις δύο κορυφές. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του αλγορίθμου σας.

Λύση:

Η άσκηση έχει υλοποιηθεί στο αρχείο Exercises/Ex22.py. (Υποθέτουμε ότι τα ίσα στοιχεία πάνε δεξία στο Binary Search Tree)

Ο παρακάτω αλγόριθμος, αρχικά σε κάθε βήμα ελέγχει, αν είμαστε πάνω σε έναν απο τους δύο κόμβους, που θέλουμε να βρούμε το ελάχιστο υπόδεντρο που τους περιέχει, αν όντως πέσουμε σε έναν (πράγμα που γίνεται αν ο ένας είναι απόγονος του άλλου), τότε επιστρέφουμε κατόπιν το τρέχων δέντρο, καθώς κανένα απο τα δύο υπόδεντρα δεν περιέχει τον κόμβο στον οποίο πέσαμε πάνω. Σε άλλη περίπτωση, ελέγχουμε αν και οι δύο κόμβοι βρίσκονται στο ίδιο υπόδεντρο*, αν ναί κάνουμε αυτό το βήμα δεξιά ή αριστερά, αν όχι επιστρέφουμε το τρέχων δέντρο, καθώς ο ένας κόμβος βρίσκεται στο δεξί υπόδεντρο ενώ ο άλλος στο αριστερό του τρέχοντας δέντρου, άρα δεν μπορούμε να βρούμε μικρότερο υπόδεντρο που περιέχει και τους δύο.

Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(h)$, όπου h το ύψος του δέντρου, καθώς στη χειρότερη περίπτωση, θα εκτελέσουμε h βήματα, αυτή προκύπτει άν οι δύο κόμβοι ταυτίζονται και βρίσκονται σε ύψος h απο τη ρίζα.

*Σε ένα δυαδικό δέντρο αναζήτησης, γνωρίζουμε αν ένας κόμβος βρίσκεται στο αριστερό ή το δεξί υπόδεντρο, συγκρίνοντας την τιμή του κατάλληλα.

```
def leastSubtree(T: BinaryTree, node1: TreeNode, node2: TreeNode)
->
BinaryTree:
    node2InNextSubtree = True

    while node2InNextSubtree:
        next = None
        node2InNextSubtree = False

        if node1 is T.root or node2 is T.root:
            return T

        if node1.value >= T.root.value and
        node2.value >= T.root.value:
            next = T.root.right
```

```
            node2InNextSubtree = True

        elif node1.value < T.root.value and
        node2.value < T.root.value:
            next = T.root.left
            node2InNextSubtree = True

        if node2InNextSubtree:
            T = BinaryTree(next)

    return T
```

15 Άσκηση 23

Σχεδιάστε αλγόριθμο που, με είσοδο ένα δέντρο αναζήτησης και την διεύθυνση μιας κορυφής του δέντρου, βρίσκει την διεύθυνση της κορυφής εκείνης με την αμέσως μεγαλύτερη τιμήμ αν αυτή υπάρχει. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά του αλγορίθμου σας.

Λύση:

Ο ακόλουθος αλγόριθμος είναι υλοποιημένος στο Exercises/Ex23.py, κι αποτελείται απο 2 υπορουτίνες και την κύρια υπορουτίνα μας, με έτοιμα test cases στον κώδικα.

Η υπορουτίνα findLastLeftTurn, βρίσκει τη διαδρομή απο τη ρίζα στον κόμβο μας και επιστρέφει την τελευταία αριστερή κίνηση που κάναμε.

Η υπορουτίνα findLeftMostChild, βρίσκει το αριστερότερο παιδί ενός κόμβου, ακολουθώντας τους αριστερούς δείκτες τους.

Ο αλγόριθμος στηρίζεται στην εξής λογική:

Έστω w ο κόμβος του οποιού αναζητούμε τον αμέσως μεγαλύτερο. Έστω l ο κόμβος στη διαδρομή, απο την ρίζα στον w, στον οποίο κάναμε την τελευταία αριστερή στροφή, (εφόσον αυτός υπάρχει). Έστω v ο κόμβος που βρίσκεται αριστερότερα απο όλους στο δεξί υπόδεντρο του w, (εφόσον αυτός υπάρχει)

Παρατηρούμε ότι $w < l$ (ο w είναι στο αριστερό υπόδεντρο του l).

Έστω η διαδρομή προς τον w, ότι είναι η $s_1, \cdots, s_k, l, j_1, \cdots, j_m$. Επειδή στον l, κάναμε το τελευταίο αριστερό βήμα, το w, βρίσκεται στο δεξί υπόδεντρο των $j_1, \cdots, j_m$, άρα $w \geq j_i$, για κάθε κόμβο στα αριστερά τους υπόδεντρα h $w \geq j_i > h$, όμως αυτά τα αριστερά υπόδεντρα είναι το υπόδεντρο του l πλήν το υπόδεντρο του w, επομένως το στοιχείο που αναζητάμε είτε θα είναι στο δεξί υπόδεντρο του w (προφανώς όχι στο αριστερό, γιατί περιέχει μικρότερα στοιχεία) ή θα είναι το l, μπορεί να είναι και εκτός αυτού του υπόδεντρου και του l? Ισχυριζόμαστε πως όχι διότι (επαγωγή στο k, μήκος διαδρομής μέχρι l):

Αν $k = 1$ (μήκος διαδρομής μέχρι τελευταία αριστερή στροφή), τότε αν κάνουμε απο τον j_1 δεξί βήμα για να φτάσουμε στον l, τότε για κάθε κόμβο d στο αριστερό του υπόδεντρο, επειδή ο w θα είναι στο δεξί υπόδεντρο του j_1 αφού ο l είναι, $w \geq j_1 > d$, άρα οι υπόλοιποι κόμβοι δεν είναι καν υποψήφιοι καθώς έχουν μικρότερη τιμή, άρα ο ισχυρισμός που κάναμε ισχύει, άν κάναμε αριστερό βήμα για να φτάσουμε στον l, τότε για κάθε κόμβο d στο δεξί του υπόδεντρο $d \geq j_1 > l > w$, άρα ο l, είναι καλύτερη επιλογή, καθώς εξακολουθεί να είναι μεγαλύτερος του w και μικρότερος απο τις εναλλακτικές, δηλαδή δεν γίνεται ο d να είναι ο αμέσως μεγαλύτερος του w καθώς ο l παρεμβάλλεται ανάμεσα τους, συνεπώς σε κάθε περίπτωση ο l είναι ο αμέσως επόμενος μεγαλύτερος κόμβος του w (εκτός του υπόδεντρου του w). Έστω οτι ισχύει ο ισχυρισμός για n βήματα, θα εξετάσουμε τι συμβαίνει για $n + 1$.

Για το υπόδεντρο του s_1, χωρίς τον s_{n+1} και το αντίθετο υπόδεντρο του s_{n+1} (αυτό που δεν περιέχει τον l), ισχύει ότι ο l είναι ο αμέσως μεγαλύτερος του w, καθώς αν αφαιρέσουμε τον κόμβο s_{n+1}, και το αντίθετο υπόδεντρο (αυτό που δεν περιέχει τον l) και κολλήσουμε κατάλληλα τον l στο s_n (ανάλογα τις τιμές τους, βάζουμε τον l δεξιά αν $l \geq s_n$, αλλιώς αριστερά του s_n), απο την επαγωγική υπόθεση, ο l είναι η καλύτερη επιλογή για αυτό που θέλουμε (η διαδρομή μέχρι τον l έχει τώρα n βήματα), άρα μένει να δούμε αν υπάρχει καλύτερη στο αντίθετο υπόδεντρο που αφαιρέσαμε (το αντίθετο υπόδεντρο του s_{n+1}).

Με την ίδια ακριβώς λογική που εφαρμόσαμε στην περίπτωση $k = 1$, βλέπουμε, ότι πάλι ο l είναι καλύτερη επιλογή απο όλους τους κόμβους του υπόδεντρου που αφαιρέσαμε πάνω, καθώς αυτοί είτε είναι μικρότεροι του w ή ο l παρεμβάλλεται ανάμεσα τους και του w.

Εξασφαλίσαμε ότι ο l είναι ο αμέσως μεγαλύτερος κόμβος απο τον w, εκτός του δεξιού υπόδεντρου του w.

Θα πάρουμε τώρα τον κόμβο που βρίσκεται αριστερότερα απο όλους στο δεξί υπόδεντρο, τον v, προφανώς $v \geq w$, καθώς για τυχαίο κόμβο στο δεξί υπόδεντρο του u, ισχύει $w \leq v \leq u$, άρα αυτός μόνο αυτός μπορεί να είναι ο αμέσως μεγαλύτερος κόμβος.

Παρατηρούμε τέλος ότι $w \leq v < l$, καθώς ο v, βρίσκεται στο αριστερό υπόδεντρο του l, αφού ο w βρίσκεται στο ίδιο υπόδεντρο και ο v απόγονος του w.

Άρα ο κόμβος που αναζητούμε είναι ο v, άν υπάρχει δεξί υπόδεντρο του w, άν όχι τότε ο l είναι ο κόμβος που αναζητάμε, αν ούτε αυτός υπάρχει ο w είναι ο μεγαλύτερος σε όλο το δέντρο καθώς βρίσκεται δεξιότερα απο όλους, άρα δεν υπάρχει ο κόμβος που αναζητάμε.

```
def findLastLeftTurn(T: BinaryTree, node: TreeNode,
lastLeftTurn: TreeNode = None) -> TreeNode:

    if node is T.root:
        return lastLeftTurn

    if node.value >= T.root.value:
        return findLastLeftTurn(BinaryTree(T.root.right)
        , node, lastLeftTurn)

    return findLastLeftTurn(BinaryTree(T.root.left), node, T.root)


def findExactGreater(T: BinaryTree, node: TreeNode) -> TreeNode:
    right = node.right

    if right is not None:
        return findLeftMostChild(right)

    return findLastLeftTurn(T, node)


def findLeftMostChild(node: TreeNode) -> TreeNode:

    while node.left is not None:
```

```
        node = node.left

    return node
```

Ο αλγόριθμος βρίσκει το αρίστερο παιδί στο δεξί υπόδεντρο του κόμβου, αν έχει δεξί υπόδεντρο, ή ακολουθει τη διαδρομή μέχρι τον κόμβο, και στις δύο περιπτώσεις έχουμε χρονική πολυπλοκότητα $O(h)$, καθώς το μόνο που κάνουμε είναι να ακολουθούμε μονοπάτια με συγκρίσεις $O(1)$ πολυπλοκότητας, και το μεγαλύτερο μονοπάτι ενός δέντρου έχει μήκος h.

16 Άσκηση 25

Να υλοποιηθεί ουρά προτεραιότητας με σωρό και να περιγραφούν αλγόριθμοι για τις enqueue και dequeue. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά των αλγορίθμων σας.

Λύση:

Η υλοποίηση βρίσκεται στο exercises/PriorityQueue.py και χρησιμοποιεί την υλοποίηση σωρού στο exercises/_Heap.py.

Η ουρά προτεραιότητας είναι ουσιαστικά ένα wrapper για τον σωρό.

Ένας αλγόριθμος enqueue, απλά εισάγει ένα στοιχείο (μια τιμή) με μια προτεραιότητα, η οποία χρησιμοποιείται για συγκρίσεις στον σωρό (είναι πραγματικός αριθμός).

Ένας αλγόριθμος dequeue, απλά εξάγει το στοιχείο (την τιμή του) της ρίζας, κι εφόσον το heap μας είναι maxHeap, αυτό θα έχει την μεγαλύτερη προτεραιότητα.

(Για τη μη αναμενόμενη χρήση dictionaries παρέχεται εξήγηση στο τέλος της παρούσας άσκησης)

```
class PriorityQueue:

    def __init__(self, maxHeapHeight: int = 10) -> None:
        self._heap = Heap(maxHeapHeight)
        self.priorityToValue: dict[float, Any] = {}
        # dict to save dev time read the note
        # at the end of this exercise
```

```
def enqueue(self, value, priority: float):
    self._heap.insert(priority)
    self.priorityToValue[priority] = value

def dequeue(self):
    prio = self._heap.popRoot()
    return self.priorityToValue.pop(prio)
```

Εκτελώντας το αρχείο, με ένα παράδειγμα (στο if name == main κλάδο κάτω) παρατηρούμε οτι λειτουργεί, όπως ήταν αναμενόμενο.

Αρχικά σημειώνουμε οτι η εισαγωγή ενός στοιχείου στο Heap και η εξαγωγή της ρίζας του, απαιτούν χρόνο $O(logn)$, καθώς με τις υπορουτίνες heapifyUp και heapifyDown (οι οποίες βρίσκονται στο Heap.py), εξασφαλίζουμε ότι ο σωρός παραμένει πάντα σχεδόν πλήρες δυαδικό δέντρο, που σημαίνει ότι όλες οι βασικές πράξεις γίνονται εξαιρετικά γρήγορα ($O(logn)$.

Τώρα η enqueue και η dequeue, βασίζονται στην εισαγωγή και εξαγωγή απο σωρό άρα έχουν χρονική πολυπλοκότητα $O(logn)$.

Σημείωση: εδώ η υλοποίηση του σωρού έγινε λανθασμένα, με τα στοιχεία του να είναι απλοί αριθμοί κι όχι δυάδες τιμής και προτεραιότητας, που σημαίνει ότι η μοναδική πληροφορία που μπορούμε να περάσουμε στον σωρό είναι η προτεραίοτητα, επομένως λόγω έλλειψης χρόνου, αποφασίστηκε για να μην χρειαστεί να ξαναυλοποιηθεί ο σωρός απο την αρχή, να "κλέψουμε", χρησιμοποιώντας ένα dictionary, το οποίο δεδομένο μια τιμή προτεραιότητας, βρίσκει την τιμή που της αντιστοιχεί, αυτό γίνεται σε χρόνο λιγότερο απο $O(logn)$, μέσω κατακερματισμού (hashing), υλοποιημένο στην ίδια την python κι άρα δεν επηρεάζει την χρονική πολυπλοκότητα και ελάχιστα την ποιότητα της λύσης, καθώς υπάρχει πρόβλημα αν δύο στοιχεία έχουν ίδια τιμή προτεραιότητας, το οποίο εύκολα λύνεται με πολύ "μικρές" διαταραχές σε αυτές τις τιμές, ώστε να μην ταυτίζονται ακριβώς.

17 Κεφάλαιο 5

Έχουν υλοποιηθεί γραφήματα με πίνακα και λίστα γειτνίασης στο exercises/_GraphAdjMat.py και exercises/_GraphAdjLi.py αντίστοιχα, με έτοιμα παραδείγματα. (όχι με πίνακα πρόσπτωσης)

18 Άσκηση 31

Εκτελέστε τους DFS και BFS για το γράφημα, υλοποιώντας τους με στοίβα και ουρά αντίστοιχα.

Λύση:

Η άσκηση είναι υλοποιημένη στο αρχείο Exercises/Ex31.py χρησιμοποιώντας, την υλοποίηση του γραφήματος με λίστα γειτνίασης απο την ακριβώς πάνω άσκηση. Επίσης χρησιμοποιούμε την υλοποίηση Stack στο Exercises/_Stack.py και την υλοποίηση Queue στο Exercises/_Queue.py. Απο κάτω παραθέτουμε και μια στοιχειώδη υλοποίηση DFS, BFS.

```
def dfs(start: Node, graph: Graph, onVisit):
    visited = [False] * len(graph.nodes)
    visited[start.index] = True
    s = Stack()
    s.push(start)

    while s.length > 0:
        v: Node = s.pop()
        onVisit(v)

        head = v.outgoingNeighbors._head
        while head is not None:

            if not visited[head.value.index]:
                visited[head.value.index] = True
                s.push(head.value)

            head = head._next

def bfs(start: Node, graph: Graph, onVisit):
    visited = [False] * len(graph.nodes)
    visited[start.index] = True
    s = Queue(len(graph.nodes))
    s.enqueue(start)

    while s.length > 0:
        v: Node = s.dequeue()
```

```
onVisit(v)

head = v.outgoingNeighbors._head
while head is not None:

    if not visited[head.value.index]:
        visited[head.value.index] = True
        s.enqueue(head.value)

    head = head._next
```

19 Άσκηση 32

Να γραφεί αλγόριθμος που με είσοδο ένα κατευθυνόμενο γράφημα D = (V, E), ελέγχει αν υπάρχουν κύκλοι σε αυτό. Μπορείτε να τον τροποποιήσετε ώστε να τους επιστρέφει; Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά των αλγορίθμων σας.

Λύση:

Η υλοποίηση του πρώτου μέρος βρίσκεται στον Exercises/Ex32.py.

Για να βρούμε αν απλά υπάρχουν κύκλοι σε ένα γράφημα, αρχικά θα κατασκευάσουμε μια υπορουτίνα, η οποία ελέγχει αν υπάρχει κύκλος με αφετηρία έναν κόμβο του γραφήματος. Αυτό θα γίνει παίρνοντας τον αλγόριθμος DFS απο την προηγούμενη άσκηση, και τροποποιώντας τον, ώστε αν κατά την εξερεύνηση, βρεί ακμή σε έναν κόμβο s, η οποία οδηγεί σε έναν εξερευνημένο κόμβο w, τότε αυτό σημαίνει ότι υπάρχει μονοπάτι απο τον s στον w και λόγω της ακμής αυτής, με την πρόσθεση της, βρίσκουμε κύκλο. Αυτές οι ακμές λέγονται back ακμές. Τώρα για να βρούμε αν το γράφημα έχει κύκλο, αρκεί να εκτελέσουμε την υπορουτίνα μας για κάθε κορυφή. Η χρονική πολυπλοκότητα της υπορουτίνας είναι ίδια με του DFS, άρα ως γνωστόν $O(|V| + |E|)$ και επειδή την εκτελούμε στη χειρότερη $|V|$ φορές η συνολική χρονική πολυπλοκότητα είναι $O(|V|^2 + |E||V|)$, τετραγωνική.

```
def hasCycleFrom(start: Node, graph: Graph):

    visited = [False] * len(graph.nodes)
    visited[start.index] = True
```

```
    s = Stack()
    s.push(start)

    while s.length > 0:
        v: Node = s.pop()

        head = v.outgoingNeighbors._head
        while head is not None:

            if visited[head.value.index]:
                return True

            if not visited[head.value.index]:
                visited[head.value.index] = True
                s.push(head.value)

            head = head._next

    return False

def hasCycle(graph: Graph):

    for node in graph.nodes:

        if hasCycleFrom(node, graph):
            return True

    return False
```

Γίνεται να τροποποιηθεί ώστε να επιστρέφονται και οι ίδιοι οι κύκλοι, όπως βλέπει κανείς και στο βιβλίο/σύγγραμμα Αλγόριθμοι των Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, κατασκευάζοντας το δέντρο του DFS (γραμμικοποιημένο), και για κάθε back ακμή υπάρχει και ένας κύκλος, οπότε μπορούμε να τους κατασκευάσουμε. Η υλοποίηση ενός τέτοιου αλγορίθμου είναι ιδιαίτερα δύσκολη και για αυτό δεν μπορέσαμε να την υλοποίησουμε, όμως υπάρχει.

20 Άσκηση 33

Να γραφεί αλγόριθμος που με είσοδο ένα κατευθυνόμενο γράφημα D = (V, E), να βρίσκει τις Ισχυρά Συνεκτικές του Συνιστώσες. Υπολογίστε και εξηγήστε την χρονική πολυπλοκότητά των αλγορίθμων σας.

Λύση:

Η άσκηση είναι υλοποιημένη στο αρχείο Exercises/Ex33.py, με έτοιμο παράδειγμα πρός εκτέλεση, το οποίο αποτελείται απο το ίδιο γράφημα με την άσκηση 31.

Σημείωση, χρησιμοποιούμε μια ελαφρώς διαφορετική υπορουτίνα DFS, σε αυτήν την άσκηση, φτιαγμένη για να γίνει κομψότερη.

Για να βρούμε τις ισχυρά συνεκτικές συνιστώσες ενός κατευθυνόμενου γραφήματος, δεν αρκεί ένα απλό DFS, όπως στην περίπτωση ακατεύθυντου γραφήματος. Ύστερα απο έρευνα στο διαδίκτυο, βρήκαμε μέσω των εξής συνδέσμων:

https://www.geeksforgeeks.org/strongly-connected-components/

https://www.topcoder.com/thrive/articles/kosarajus-algorithm-for-strongly-connected-components

https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

https://www.youtube.com/watch?v=5wFyZJ8yH9Q

Τον αλγόριθμο του Kosaraju για την εύρεση των ισχυρά συνεκτικών συνιστώσων, ο οποίος λειτουργεί ως εξής:

Κατασκευάζουμε μια στοίβα, και εκτελούμε διαδοχικά DFS, στα οποία τρέχουμε ως υπορουτίνα postvisit (εκτελείται αφού φτάσουμε σε κορυφή με όλους τους γείτονες τις εξερευνημένους ήδη), να βάζουμε την κορυφή στη στοίβα. Τρέχουμε DFS κατά αυτόν τον τρόπο, έως ότου όλοι μπούν στην στοίβα. Ύστερα αντιστρέφουμε την φορά όλων των ακμών του γραφήματος, δηλαδή αν $(u, v) \in E$, τότε $(v, u) \in E'$. Αρχίζουμε τώρα να εξάγουμε κάθε στοιχείο της στοίβας και να εκτελούμε το ίδιο DFS, αλλά τώρα ως υπορουτίνα postvisit, βάζουμε τους κόμβους σε μια ισχυρά συνεκτική συνιστώσα εφόσον δεν έχουν μπεί ήδη σε μιά άλλη (το βλέπουμε αυτό με μια λίστα τιμών αλήθειας) και επιστρέφουμε μια λίστα με τις συνεκτικές συνιστώσες (στην υλοποίηση το κάναμε array για ευκολία).

Ο αλγόριθμος σε python είναι ο εξής:

```
def dfs(start: Node, visited, postVisit):
    explore(start, visited, postVisit)

def explore(node: Node, visited, postVisit):
    visited[node.index] = True

    head = node.outgoingNeighbors._head
    while head is not None:

        if not visited[head.value.index]:
            explore(head.value, visited, postVisit)

        head = head._next

    postVisit(node)

def Kosaraju(graph: Graph):
    stack = Stack()
    visited = [False] * len(graph.nodes)

    def postvisit(x: Node):
        stack.push(x)

    i = 0
    while stack.length < len(graph.nodes):
        start = graph.nodes[i]
        if visited[i]:
            i += 1
            continue
        dfs(start, visited, postvisit)
        i += 1

    graph.reverseEdges()

    added = [False] * len(graph.nodes)
    remaining = [len(graph.nodes)]
    components = []
```

```
while stack.length != 0:
    component = []
    def post(x: Node):
        if added[x.index]:
            return

        added[x.index] = True
        remaining[0] -= 1
        component.append(x)

    dfs(stack.pop(), [False] * len(graph.nodes), post)
    components.append(component)

    if remaining[0] == 0:
        break

return components
```

Όσο για την χρονική πολυπλοκότητα, σύμφωνα με τις παραπάνω πηγές ο αλγόριθμος Kosaraju τρέχει σε γραμμικό χρόνο $O(|V| + |E|)$, επειδή βασίζεται σε κλήσεις DFS. Τώρα ενδεχομένως, αυτή η υλοποίηση να είναι πιο αργή, λόγω είτε κάποιου λάθους που δεν βλέπουμε, ή της σχετικά αργής υλοποίησης του graph.reverse(), που αντιστρέφει τις κατευθύνσεις των ακμών, το οποίο στην προκειμένη περίπτωση είναι τετραγωνικού χρόνου, σίγουρα υπάρχει τρόπος να γίνει γραμμικό, αλλά λόγω χρόνου, δυσκολίας και πολλών λυμένων ασκήσεων, δεν θα γίνει εδώ.