Προηγμένη Σχεδίαση Αλγορίθμων και Δομών Δεδομένων: 1ο σύνολο ασκήσεων

Άσκηση 1

```
public class MonotonePQ {
    int n;
    boolean[] A;

    public MonotonePQ(int n) {
        this.n = n;
        A = new boolean[n];
        for (int i=0; i<n; i++) {
            A[i] = true;
        }
    }

    void Delete(int i) {
        A[i] = false;
    }

    int DeleteMin() {
        int i=0;
        while (A[i]==false) {
            i++;
            if (i==n) return -1;
        }
        Delete(i);
        return i;
    }
}
```

Δεδομένα:

>Πρόγραμμα: υλοποιεί μονότονη P.Q. για ένα σύνολο A από integers.
> A = {0, 1, . . . , n −1} <==> A[i] = True for every i , 0<= i <= n-1
> Έχουμε ένα Constructor **MonotonePQ** που με δεδομένο το μέγεθος του πίνακα, τον αρχικοποιεί ολόκληρο μέσω ενός for loop.

> Έχουμε την μέθοδο **Delete** την μέθοδο που με δεδομένο το index ενός element το διαγράφει άμεσα.
> Τέλος έχουμε την μέθοδο **DeleteMin** που μέσω ενός while loop ψάχνει τον ελάχιστο αριθμό.
> Αν ολόκληρος ο πίνακας A είναι empty, επιστρέφει -1.
> Διαφορετικά διαγράφει το element & επιστρέφει το index του.

(a) Worst Case:

• Προφανώς το worst case για τη **Delete** είναι **O(1)** γιατί δεν υπάρχει εξάρτηση από το μέγεθος του πίνακα. Γνωρίζουμε το index του element που θέλουμε να διαγράψουμε και το διαγράφουμε άμεσα. Ενώ το worst case για τη **DeleteMin** είναι **O(n)** ακριβώς γιατί υπάρχει αυτή η εξάρτηση και μπορεί να χρειαστεί να διατρέξουμε ολόκληρο το πίνακα μεγέθους **n**.

Amortized Cost:

• Ιδέα αντισταθμιστικής ανάλυσης: Έχουμε μια Δομή που περιέχει διάφορες τύπου λειτουργίες(μεθόδους/ρουτίνες) και η καθεμία έχει το δικό της κόστος. Μας ενδιαφέρει η χειρότερη περίπτωση **μέσου κόστους/λειτουργία** έχοντας μια ακολουθία από λειτουργίες.
==> **C(n) = (worst-case total cost of a sequence of n operations)/n**
(Πιθανό να έχουμε κάποια συνθήκη για το πόσες φορές μπορεί να "τρέξει" κάθε λειτουργία σε μια ακολουθία.)

• Ο αντισταθμιστικός χρόνος για τη **Delete** είναι **O(1)** προφανώς, γιατί η μέθοδος τρέχει σε σταθερό χρόνο αναξάρτητα από την ακολουθία από λειτουργίες.

• Τώρα σχετικά με την **DeleteMin** : Για μια ακολουθία από n DeleteMin το πραγματικό κόστος θα είναι O(n) γιατί κάθε element μπορεί να διαγραφεί 1 φορά! Άρα το αντισταθμιστικό κόστος θα είναι O(1). Βέβαια με την DeleteMin καλείτε η Delete. Αν όμως "αγνοηθεί" η Delete σε μια ακολουθία **n** πράξεων, το αντισταθμιστικό κόστος θα είναι τελικά **O(n)**.

(b)

```
public class MonotonePQ {

    LinkedList<Integer> q;  // we want to know the “True” indexes
    boolean[] A;
    int n;

    public MonotonePQ(int n) {
        this.n = n;
        A = new boolean[n];
        q = newLinkedList();
```

```
        for (int i=0; i<n; i++) {
            q.addLast(i);  // Initially, all values are true, so add all indices to the queue.
            A[i] = true
        }
    }

    boolean isEmpty(){ // returns if P.Q. is empty
        if(q.isEmpty()){return true;}
        return false;
    }

    void Delete(int i) {
        A[i] = false;
    }// fetch 1st true value

     int DeleteMin(){
         int nonEmptyIndex = -1;

         if(isEmpty(){ return -1;}
         while(!A[q.getfirst()] && !isEmpty()){
              q.removeFirst(); // remove false values 1 by 1
          }

          nonEmptyIndex = q.removeFirst();  // fetch 1st true value
          A[nonEmptyIndex] = false;// make it false
          return nonEmptyIndex;
       }
}
```

• Προφανώς στη **Delete** δεν υπάρχει κάποια αλλαγή.
• Στη **DeleteMin**, μέσω της μεθόδου isEmpty() και της συνδεδεμένης λίστας φροντίζουμε να “διώχνουμε” τα indexes που είναι άδεια και έτσι δεν ξοδεύουμε χρόνο να ψάχνουμε κάθε φορά τον πίνακα A από την αρχή.

Σχόλιο: Μια απλούστερη ιδέα είναι η παρακάτω.
Να δηλώσουμε μια extra μεταβλητή, έστω **minElementIntex** που θα παρακολουθεί "ποιό είναι το στοιχείο που υπάρχει στην Δομή μας κρατώντας το index του".
Η χρήση της μεταβλητής θα γίνει κυρίως σε μια while λούπα στην DeleteMin(). Το θέμα είναι ότι χωρίς μιας ποιό sophisticated βοηθητική δομή δεν μπορούμε να μειώσουμε το αντισταθμιστικό κόστος της DeleteMin() σε O(1).

(c)
> Θα εκμεταλευτούμε το γεγονός ότι κάθε στοιχείο είναι μοναδικό στην δομή μας.

> Θα κάνουμε χρήση της τιμής που αντιστοιχει σε κάθε στοιχείο με την βοήθεια ενός HashMap.

```
public class MonotonePQ {
   LinkedList<Integer> queue;
   HashMap<Integer, Integer> map; // To Map the value
   int min;
   public MonotonePQ(int n) {
      queue = new LinkedList<>();
      map = new HashMap<>();
      for (int i = 0; i < n; i++) {
         map.put(i, i);
         queue.add(i);
      }
   }

   void Delete(int i) {
      if (map.containsKey(i)) {
         queue.remove(map.get(i));
         map.remove(i);
      }
   }

   int DeleteMin() {
      if (queue.isEmpty()){
        return -1;
      }
      min = queue.poll(); // Retrieves and removes the head (1st element).
      map.remove(min);
      return min;
   }

}
```

Μέ την βοήθεια των 2 δομών απλο τη συλλογή της java καταφέρνουμε τα ακόλουθα:
> Η **linked list** μας επιτρέπει διαγραφή του ελαχίστου στοιχείου σε **O(1)**
> Ενώ η **HashMap** βρίσκει το στοιχείο(τη θέση του) προς διαγραφή σε **O(1)**

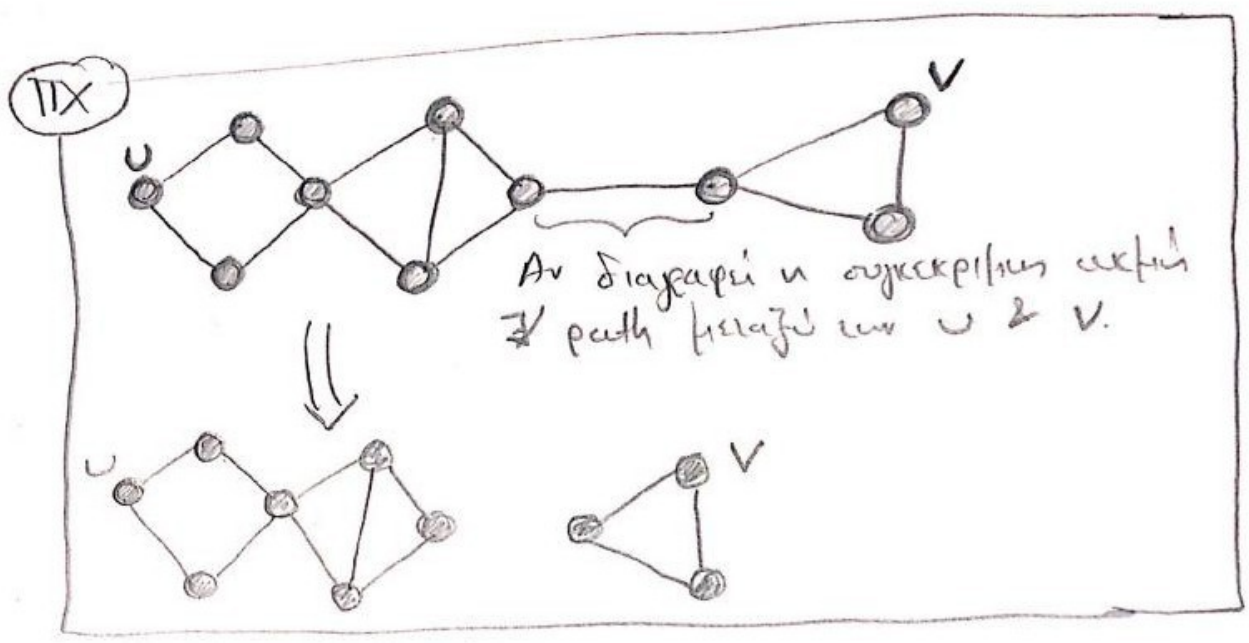
Άσκηση 4

Άσκηση 4 (Συνεκτικότητα γραφημάτων)

α) Έστω $G = (V, E)$ ένα συνεκτικό μη κατευθυνόμενο γράφημα. Σχεδιάστε έναν **γραμμικό αλγόριθμο** (δηλαδή με χρόνο εκτέλεσης $O(|V| + |E|)$) που να προσδιορίζει εάν υπάρχει κάποια ακμή $e \in E$ τέτοια ώστε το G να παραμένει συνεκτικό μετά τη διαγραφή της e. Μπορείτε να κάνετε τον αλγόριθμο σας να εκτελείται σε χρόνο $O(|V|)$;

β) Μας δίνεται ένα κατευθυνόμενο γράφημα $G = (V, E)$ για το οποίο θέλουμε να βρούμε εάν υπάρχει κόμβος $v \in V$ από τον οποίο είναι προσβάσιμοι όλοι οι κόμβοι του G. (Δηλαδή για κάθε κόμβο $v \in V$, πρέπει να υπάρχει μονοπάτι από τον s στον v.) Δώστε έναν **αποδοτικό αλγόριθμο** που να επιστρέφει ένα τέτοιο s αν υπάρχει. Ποιος είναι ο χρόνος εκτέλεσης του αλγορίθμου σας;

(a)



> Στο παραπάνω πχ κ=1
> Γραφημά με κ-συνεκτικότητα ακμών: "Παραμένει συνεκτικό" μετά τη διαγραφή οποιουδηποτε συνόλου ακμών υποσύνολο S του συνόλου E με |S| <= k-1
> Μια 1η ιδέα είναι να κάνουμε χρήση αλγορίθμου διερεύνησης.

Έστω κάνουμε χρήση του **DFS :**

(1) **Διαγράφουμε** ακμή **e** από το **G(V,E)** ==> **G'(V,E-{e})**

(2) **Επιλογή source** κορυφής **s** από την ακμή που μόλις διαγράψαμε.
Εφόσον ο γράφος ήταν συνεκτικός πριν την διαγραφή της ακμής, αυτό που μας ενδιαφέρει είναι το **αν: "μετά την διαγραφή της εν λόγω ακμής μπορούμε να επισκευτούμε όλους τους άλλους κόμβους"**.
(3) **Τρέχουμε DFS** αρχίζοντας από τον s που έχουμε διαλέξει στο βήμα (2).
Αποθηκεύουμε όλες τις κορυφές στις οποίες φτάσαμε από τον s σε ένα πίνακα.
(4) **Ελέγχουμε** αν μπορέσαμε να επισκεφτούμε όλες τις κορυφές με μια απλή διαπέραση του πίνακά μας.
If Yes ==> Graph G'(V,E-{e}) is connected
Otherwise ==> Graph G'(V,E-{e}) is disconnected

> Προφανώς η προηγούμενη λογική της διερεύνησης δεν αρκεί. Επιπλέον δεν έχουμε κάποια επιπλέον πληροφορία για το γράφο. Για παραδειγμα την σχέση κορυφών ακμών ως προς το πλήθος τους.

> Στις αρχικές διαφάνειες του κεφ. της Αντισταθμιστικής Ανάλυσης βλέπουμε το πρόβλημα **Disjoint Set Union(Union Find)**.
> Ενδεικτικό πρόβλημα του γεγονότος ότι η αποτελεσματικότητα ενος Αλγορίθμου εξαρταται από την χρήση μιας αποτελεσματικής δομής.

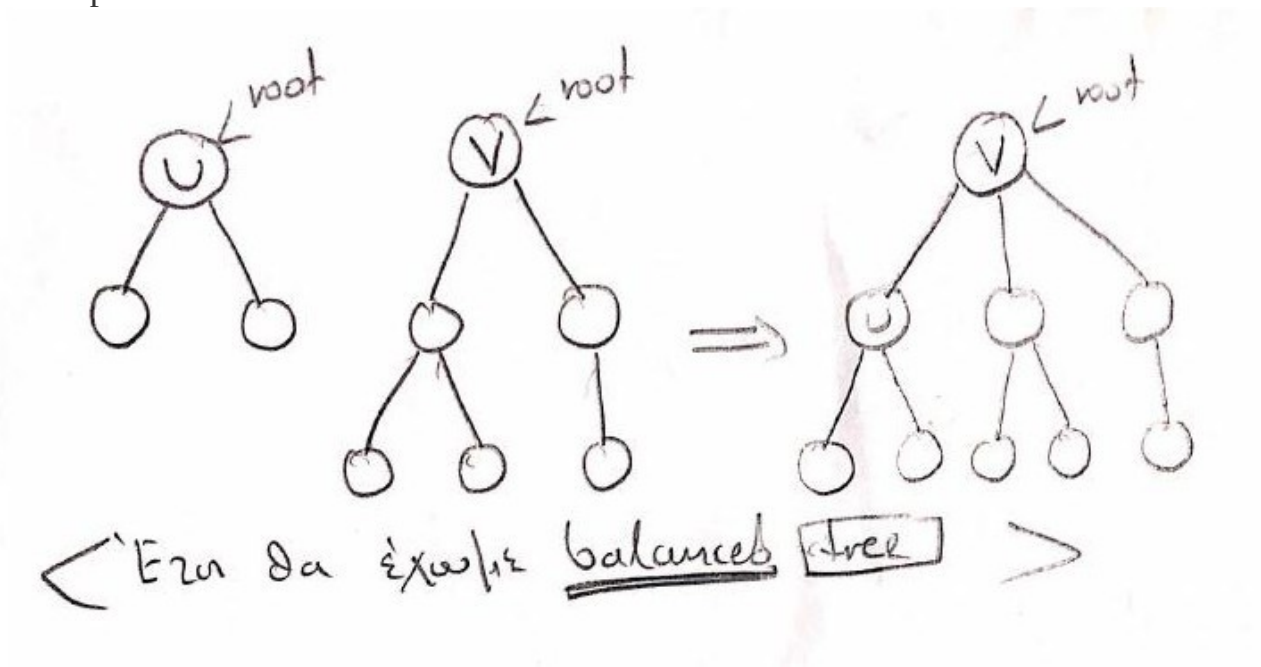
> Ας πούμε ότι έχουμε 1 σετ από N στοιχεία τα οποία πιθανώς χωρίζονται σε υπο-σετς.
> Θέλουμε να γνωρίζουμε κάθε στιγμή την συνεκτικότητα κάθε στοιχείου.

> **union(u,v)**: connects u and v
find(u,v): returns True if there exists a path between u and v
returns False otherwise

> Τα **υπο-σετς** μπορούμε να τα θεωρήσουμε ως **connected components** ενώ τα στοιχεία μπορούμε να τα θεωρήσουμε ως κόμβους ενός γράφου.
Αρχικά κάθε στοιχείο είναι το δικό του σετ(ή υπο-σετ).
> Μπορούμε να θεωρήσουμε root στοιχείο σε κάθε υπο-σετ.
Αρχικά κάθε στοιχείο είναι root του εαυτού του.
union(u,v) : θέσε ως root(v) ως γονέα του root(u).

> Μια καλή ιδέα είναι να ελέγχουμε το πλήθος των στοιχείων που έχει κάθε υπο-σετ. Έτσι θα συνδέσουμε τη ρίζα του υπο-σετ με τα λιγότερα στοιχεία με τη ρίζα του υπο-σετ με τα παραπάνω στοιχεία (**smarter union**).

> Example:




```
// Αρχικά κάθε κορυφή είναι το δικό της σετ
void init(int Array[], int N){
    for(int i=0; i<N; i++){
        Array[i] = I;  size[i] = I;
    }
}

// Όσο διερευνούμε το αρχικό γράφο G(V,E) κάνουμε union τα σετς που συνδέονται
void union(int Array[], int size[], int u, int v){
    int root_u = root(u);
    int root_v = root(v);

    if(size[root_u] <  size[root_v]{ // κοιτάμε το μέγεθος κάθε υπο-σετ
        Array[root_u] = Array[root_v];
        size[root_v] += size[root_u];
    }else{
        Array[root_v] = Array[root_u];
        size[root_u] += size[root_v];
    }
}


bool find(int u, int v){ // Βρες αν συνδέονται
    if(root(u) == root(v)){
```

```
        return True;
    }
    return False;
}

int root(int Array[], int u){  // Βρες root κάθε element
    while(Array[u] != u){
        u = Array[u];
    }
    return u;
}// while computing the root of u, set every u to point to it's grandparent
  // ==> new path length = old*(1/2)
```

Αλγόριθος(): // high level περιγραφή

(1) Πριν από την διαγραφή της ακμής θα κάνουμε χρήση του Union-Find ώστε να "χτίσουμε" τις συνεκτικές συνιστώσες του αρχικού γράφου.

(2) Υστερα στο νέο γράφο όπου έχουμε αφαιρέσει την ακμή θα κάνουμε χρήση της ρουτίνας/συνάρτησης find(int u, int v) ώστε να δουμε αν οι δύο κορυφές της διαγραφείσας ακμής συνδέονται-δηλαδή αν βρίσκονται στην ίδια συνεκτική συνιστώσα.

Με την find(int u, int v) θα μπορέσουμε να προσεγγίσουμε το O(|V|). Το πόσο κοντά θα φτάσουμε στο ζητούμενο υπολογιστικό κόστος προφανώς έχει να κάνει με το πόσο "πυκνό" είναι το γράφημα. Δηλαδή αυτο που θέλουμε ιδανικά είναι: |**E**| << |**V**|

(b)

Αλγόριθος():

(1): > Διαλέγουμε 1 τυχαίο κόμβο u κ΄εκτελούμε DFS.
> Αν δεν καταφέρουμε να "φτάσουμε" σε όλους τους κόμβους(κρατάμε την πληροφορία προσπέλασης σε 1 boolean πίνακα reached) τότε δεν θα υπάρχει τέτοιος κόμβος.
> print(“ δεν υπάρχει τέτοιος κόμβος ”)
> exit(EXIT_FAILURE)
> Διαφορετικά goto (2)

(2): > Τότε θα είναι δυνατό να πάμε σε όλους τους κόμβους έχοντας ως source τον u.
> Σε αυτή την περίπτωση θα πρέπει να τσεκάρουμε αν ο κόμβος u με σύνολο κορυφών V-{v}.
> Δημιουργούμε ένα αντεστραμένο γράφο G’(V,E’) , E΄: κάθε ακμή είναι αντεστραμένη πλέον.
> Αν μπορέσουμε να επισκεφτούμε όλους του κόμβους του G’
> print(“success”)
> exit(EXIT_SUCCESS)

Ουσιαστικά εκτελούμε DFS 2 φορές σε γράφους με ίδιο πλήθος κορυφών & ακμών! Συνεπώς το κόστος θα είναι O(2*(|V| + |E|)) = **O(|V| + |E|)**

Άσκηση 2

Άσκηση 2 (Αντισταθμιστική ανάλυση δομής διατεταγμένων πινάκων)

Θέλουμε να κατασκευάσουμε μια δομή δεδομένων που να υποστηρίζει δυαδική αναζήτηση αλλά και την αποδοτική εισαγωγή νέων στοιχείων.

Θυμηθείτε ότι για τη δυαδική αναζήτηση ενός στοιχείου x σε ένα ταξινομημένο πίνακα $A[1 \dots n]$ συγκρίνουμε το x με το μεσαίο στοιχείο $A[m]$ όπου $m = \left\lfloor \frac{n+1}{2} \right\rfloor$: (α) αν $x = A[m]$ τότε το στοιχείο έχει βρεθεί, (β) αν $x < A[m]$ τότε ψάχνουμε αναδρομικά τον υποπίνακα $A[1 \dots m-1]$ και (γ) αν $x > A[m]$ τότε ψάχνουμε αναδρομικά τον υποπίνακα $A[m+1 \dots n]$.

Ωστόσο η εισαγωγή ενός νέου στοιχείου y στο διατεταγμένο πίνακα A δεν είναι αποδοτική, ακόμα και αν ο πίνακας έχει κενές θέσεις στο τέλος (όπως θα συνέβαινε με μία υλοποίηση με δυναμικό πίνακα). Π.χ., αν $y = 1$ και $A = [2\ 5\ 19\ 22\ 35\ 48\ 81\ 95\ 101\ 110\ 134\ 149\ 256\ \blacksquare]$, όπου το ■ συμβολίζει κενή θέση στον πίνακα, τότε όλα τα στοιχεία του A πρέπει να μετακινηθούν μία θέση δεξιά, με αποτέλεσμα η εισαγωγή να γίνεται σε $O(n)$ χρόνο.

Για αυτό το λόγο προτείνουμε την ακόλουθη δομή. Έστω $k = \lceil \lg(n+1) \rceil$, ο αριθμός των bits που χρειάζονται για τη δυαδική αναπαράσταση $\langle n_{k-1}, n_{k-2}, \dots, n_1, n_0 \rangle$ του n. Π.χ., για $n = 13$ έχουμε $k = \lceil \lg(n+1) \rceil = \lceil \lg 14 \rceil = 4$ και $n = 13 = \langle 1101 \rangle$. Η δομή αποτελείται από k διατεταγμένους πίνακες A_i, $0 \le i \le k-1$. Αν $n_i = 0$ τότε ο A_i είναι κενός, διαφορετικά (όταν $n_i = 1$) ο A_i περιέχει 2^i διατεταγμένα στοιχεία. Π.χ., για το παραπάνω παράδειγμα με $n = 13$ θα μπορούσαμε να έχουμε τους πίνακες $A_0 = [35]$, $A_1 = []$ (κενός πίνακας), $A_2 = [2\ 48\ 81\ 149]$ και $A_3 = [5\ 19\ 22\ 95\ 101\ 110\ 134\ 256]$. Προσέξτε ότι τα στοιχεία ενός πίνακα είναι διατεταγμένα αλλά δεν υπάρχει κάποια συγκεκριμένη σχέση μεταξύ των στοιχείων διαφορετικών πινάκων.

α) Δώστε ένα αλγόριθμο αναζήτησης σε αυτή τη δομή. Ποιος είναι ο χρόνος εκτέλεσης χειρότερης περίπτωσης;

β) Περιγράψτε ένα αλγόριθμο εισαγωγής ενός νέου στοιχείου στη δομή μας. Αναλύστε το χρόνο εκτέλεσης χειρότερης περίπτωσης καθώς και τον αντισταθμιστικό χρόνο εκτέλεσης της εισαγωγής.

(a)

> Για ένα αριθμό α ψάχνουμε κάθε πίνακα αφού δεν υπάρχει σχέση μεταξύ των πινάκων.
> Σε κάθε πίνακα μπορούμε να χρησιμοποιήσουμε **Binary Search**(O(log(n)).

input: element a, list if arrays
output: the result

```
search(int a, int list[A's]){
    for(every A[i], 0<=i<=k-1){
        result = binary_search(A[i], a);
        if(result == True){ // element found a specific array
            return(A[i], index_of_A[i]);
            break; // terminates early to avoid unnecessary computations
        }
    }

    if (result == False){
        print("element not found");
        return -1;
    }
}
```

Worst case cost: >Στο απευκταίο σενάριο θα πρέπει να γίνει σε όλους τους πινακές μας!
> Η binary search σε ένα πίνακα μεγέθους n έχει κόστος **O(logn)**.
> Έστω k το πληθος των bits για την δυαδική αναπαράσταση και n το πλήθος των στοιχείων που έχουν "διαμοιραστεί" στους πίνακες.
> Το μέγεθος κάθε πίνακα θα είναι <= **2^(k-1)**
> Άρα το συνολικό κόστος θα είναι: **O(k*log(2^(k-1)))=O(k^2)**

(b)

Insertion():

```
x = produce_element();
// insert element x:
  (1). n += 1;
       Update k
  (2). Find the correct array A for the insertion by looking at the new
       binary representation of n
  (3). Insert element into the chosen A[i]
```

// Δεν έχω καταλάβει αν κ πως μπορεί να γίνει merge πινάκων.

Άσκηση 3

Άσκηση 3 (Διερεύνηση γραφημάτων)

Στο πρόβλημα CNF-SAT δίνεται μία λογική πρόταση φ σε συζευκτική κανονική μορφή (δηλαδή, η πρόταση αποτελείται από συζεύξεις διαζεύξεων) και ζητείται να αποφασιστεί αν υπάρχει ανάθεση αληθοτιμών στις μεταβλητές της φ, η οποία να ικανοποιεί τη φ.

Σχεδιάστε έναν **αποδοτικό αλγόριθμο** για την ειδική περίπτωση όπου η φ αποτελείται από m φράσεις και n λεκτικά (μεταβλητές x_i ή αρνήσεις τους $\neg x_i$) και κάθε φράση έχει ακριβώς 2 λεκτικά. Για παράδειγμα, η πρόταση

$$\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

είναι σε κανονική συζευκτική μορφή, αποτελείται από 3 φράσεις και 6 λεκτικά και επιπλέον κάθε φράση έχει ακριβώς 2 λεκτικά. Η φ ικανοποιείται από 2 αναθέσεις αληθοτιμών: $x_1 = A, x_2 = A, x_3 = \Psi$ και $x_1 = \Psi, x_2 = \Psi, x_3 = A$.

Υπόδειξη: Θυμηθείτε ότι η φράση $(\neg x_1 \vee x_2)$ γράφεται ισοδύναμα ως $(x_1 \rightarrow x_2)$. Επομένως, αν η μεταβλητή x_1 είναι αληθής, τότε συνεπάγεται ότι και η x_2 θα πρέπει να

είναι αληθής. Με βάση αυτή την παρατήρηση, μπορούμε να μετατρέψουμε το πρόβλημά μας σε ένα κατάλληλο πρόβλημα σε κατευθυνόμενο γράφημα.

Λύση Ας εξετάσουμε την πρόταση φ του παραδείγματος που μας δώσαμε στην εκφώνηση για καλύτερη κατανόηση.

- $x \to \begin{matrix} T \\ \dot{\eta} \\ F \end{matrix}$, $\begin{cases} \text{if } x = T \iff \neg x = F \\ \text{if } x = F \iff \neg x = T \end{cases}$ } Δηλ. x & ¬x δεν μπορούν να έχουν ίδια τιμή.

- $\varphi = \underbrace{(x_1 \vee \neg x_2)}_{\text{at least 1 } T} \wedge \underbrace{(x_2 \vee x_3)}_{\varphi_2 = T} \wedge \underbrace{(\neg x_1 \vee \neg x_3)}_{\varphi_3 T}$ (clause)

$\varphi_1 = T$

$T \Rightarrow \varphi$ ικανοποιείται

- Μπορούμε να βρούμε 1 συνδυασμό τιμών που θα δώσουν στο πρόβλημα μια ικανοποιήσιμη (satisfiable) τιμή φ? $\Rightarrow$ Έτσι έχουμε πίνακα αλήθειας (SAT problem)

- Στην περίπτωσή μας ∀ φ_i έχει ακριβώς 2 μεταβλητές

- $(x_1 \to x_2)$ $\begin{cases} \text{if } x_1 = T \text{, then } x_2 = T \\ \text{if } x_1 = F \text{, then } x_2 = T \ \dot{\eta} \ F \end{cases}$

- $x_1 \vee x_2 \iff \neg x_1 \to x_2 \iff \neg x_2 \to x_1$

- Για τον πίνακα αλήθειας του παραδείγματος:

$0 == \varphi$ & $1 == A$

x_1	x_2	x_3	$\neg x_1$	$\neg x_2$	$\neg x_3$	$(x_1 \vee \neg x_2)$	$(x_2 \vee x_3)$	$(\neg x_1 \vee \neg x_3)$	φ	
0	0	0	1	1	1	1	0	1	0	
0	0	1	1	1	0	1	1	1	1	>
0	1	0	1	0	1	0	1	1	0	
0	1	1	1	0	0	0	1	1	0	
1	0	0	0	1	1	1	0	1	0	
1	0	1	0	1	0	1	1	0	0	
1	1	0	0	0	1	1	1	1	1	>
1	1	1	0	0	0	1	1	0	0	

Όπως: $x_1 = \varphi, x_2 = \varphi, x_3 = \varphi$ & $x_1 = T, x_2 = T, x_3 = \varphi$

είναι οι μοναδικές μεταβλητές που ικανοποιούν τη φ

• Αν έχουμε $\neg x_1 \vee x_2$ ($\Leftrightarrow x_1 \to x_2$) & δοθεί $x_1 = T$, τότε πρέπει να αναθέσουμε το x_2 πάλι T ώστε να ικανοποιηθεί το $\varphi_i = \neg x_1 \vee x_2$.

• Για 3 μεταβλητές: Αν έχουμε $x_1 \to x_2$ & $x_2 \to x_3$ & δοθεί $x_1 = T$, τότε πρέπει $x_2 = T$ και κατά συνέπεια $x_3 = T$

$$(x_1 \to x_2) \wedge (x_2 \to x_3)$$
$$\uparrow \quad \downarrow \qquad \downarrow$$
$$T \quad T \qquad T$$

• $x_i \vee x_j \iff \underbrace{\neg x_i \to x_j \iff \neg x_j \to x_i}$

2 τρόποι σχέσης θα έχουμε για ∀ φ_i

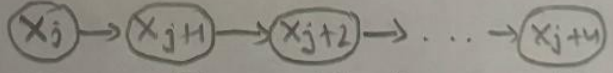
• Είναι προφανώς πλέον ότι θα χρησιμοποιήσουμε ένα κατευθυνόμενο γράφο που να "απεικονίζει αυτές τις σχέσεις".

> Οι κορυφές θα είναι οι μεταβλητές

> Οι ακμές θα είναι οι φράσεις σχέσης (Πx) $x_i \to x_j$: x_i implies x_j)

> Αν έχουμε n μεταβλητές, τότε ο γράφος θα έχει 2^n κορυφές γιατί για ∀ μεταβλητή x_i έχουμε x_i & $\neg x_i$

> Αν έχουμε ένα τμήμα του γράφου μιας από τις μορφές:



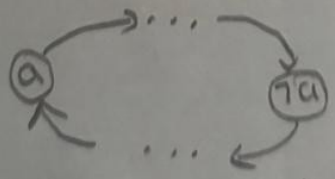
Τότε: $T \to T \to T \to \ldots \to T$.

Αν $x_j = a = T$ & $x_{j+4} = \neg a = T$ (!)

$\Rightarrow$ Άτοπο

Η φ δεν μπορεί να ικανοποιηθεί!

Διότι:



> Για να ελέγξουμε αν ∃ path μεταξύ των 2 κόμβων μπορούμε να χρησιμοποιήσουμε ή αλγόριθμο διερεύνησης έστω τον DFS για να απεικονίσουμε περιττούς στον τον (!) γράφο. Πρέπει να κάνουμε διερεύνηση 2 φορές από μεταβλητή βάσεως. $\Rightarrow$ Cost $= O(2(|E|+|V|)) = O(|2V|+1)$

a ή $\neg a$

> Σε ένα κατευθυνόμενο γράφο: Αν υπάρχει path από τον κόμβο u στον κόμβο v & από τον v στο u τότε αυτοί οι 2 κόμβοι θα ανήκουν στην ίδια SCC. (συνεκτική συνιστώσα)

> Κατά συνέπεια για να μπορέσουμε να τσεκάρουμε αν x & ¬x ανήκουν στην ίδια SCC!

> (Όπως γνωρίζουμε από την Complexity Theory το 2-SAT problem είναι το μόνο Sat problem που μπορεί να λυθεί σε πολυωνυμικό χρόνο.)

Άσκηση 5

Δεδομένα: > Connected Graph G=(V,E)
> θετικά βάρη στις ακμές w
> source node s, end node t, $s,t \in V$
> Για οποιαδήποτε ακμή e, συμβολίζουμε με $G - e$ το γράφημα που προκύπτει από το G με τη διαγραφή της e.

Ζητούμενο: Να βρούμε για κάθε ακμή $e \in E$ το συντομότερο μονοπάτι από τον s στον t στο $G - e$.

> Μια 1η λύση είναι να κάνουμε χρήση του Dijkstra's algorithm |E| φορές αφαιρώντας κάθε φορά διαφορετική ακμή από τον αρχικό γράφο.

> Ο "βασικός" Αλγόριθμος είναι ο ακόλουθος:

input: > **list** # an adjacency list of our weighted graph
> **s** # index of our source node
> **N,E** # number of vertices & edges
output: > will return an array with the shortest distance to every node from the source

```
Dijksta(G(V,E), N, E, s){
    distance = [oo, oo,…,oo]; # distance array of size N
    distance[s] = 0; # because we're already there
    visited = [False, False, …, False]; # have we visited a specific node(size N)
    PQ = (); # initialize an empty priority queue
    PQ.insert([s,0]); # insert source node to kickstart the process

    while(PQ.size() != 0){ # when PQ is empty exit loop
        index, minValue = PQ.remove(); # get most promising pair [index,value]
        visited[index] = True;
        for(every edge in list[index]){
            if(visited[edge.to] {
                continue; # we don't want to visit them again
            }
             newDistance = distance[index] + edge.cost;
             if(newDistance < distance[edge.to]{
                distance[edge.to] = newDistance;
                PQ.insert([edge.to, newDistance]);
             }
```

```
        }
    }
  return distance;
}
```

> Για να βρούμε το βέλτιστο μονοπάτι, δηλαδή στην περίπτωση που δεν μας ενδιαφέρει μόνο η βέλτιστη απόσταση για ένα συγκεκριμένο κόμβο αλλα θέλουμε την ακολουθία των κόμβων για να φτάσουμε σε αυτον πρεπει να κρατήσουμε επιπλέον ένα πίνακα που κρατά τον προηγούμενο κόμβο για να φτάσουμε στον τωρινό.

```
# the index of the previous node we used to get to the current node
prev = [] # initially empty(size N)
…
if(newDistance < distance[edge.to]){
    prev[edge.to] = index;
    distance[edge.to] = newDistance;
    PQ.insert([edge.to, newDistance]);
…
return distance, prev; # now we'll return 2 arrays
```

> Άρα τώρα μας ενδιαφέρει το πως θα βρούμε το shortest path μεταξύ 2 κόμβων.

```
findShortestPath(G, source, end){
    distance, prev = Dijksta(G(V,E), N, E, s);
    path[]; // will be in reverse order
    if (distance[e] == oo){
        return path;
    }
    for(i=e; i not empty; i=prev[i]){
        path.add(i);
    }
    path.reverse(); // simply reverse the array
    return path;
}


main(){
  for(i=1; i<|E|; i++){ // run the algorithm for different E each time
    path =  findShortestPath(G(V,E-{ei}), source, end);
    print ("path between source & end without" + ei + "edge is:" + path);
  }
}
```

> Ο Αλγόριθμος με χρήση **Heap Priority Queue** έχει κόστος**: O((E+V)*log(V))**
Άρα αν εκτελεστεί |E| φορές**: O(E*(E+V)*log(V))**
> Το κόστος είναι μάλλον υψηλό και αυτό γιατί κάθε φορά που αφαιρούμε μια ακμή επαναλαμβάνουμε την ίδια διεργασία.

> Για να δουλέψει σωστά ο παραπάνω Αλγόριθμος υπάρχουν 2 βασικές προϋποθέσεις:
1) Ότι ο γράφος έχει θετικα βάρη, κάτι που μας δίνεται ως δεδομένο.
2) Οτι ο γράφος κάθε φορά που θα διαγράφουμε μια ακμή θα εξακολουθεί να είναι connected.

> Συνεπώς μπορούμε να αφαιρέσουμε την ακμή **e = uv** από τον γράφο και να χρησιμοποιήσουμε τον DFS για να τσεκάρουμε αν μπορούμε να "φτάσουμε" από τον u στον v ή από τον v στον u. (1 **DFS** ==> Cost **O(|V| + |E|)**)
Cost: **O(|E|*(|V| + |E|))**

Πηγές:
1) Διαφάνειες διδάσκοντα
2) https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2012/83b82d45beb3776da72b7f3e1b3f42df_MIT6_046JS12_lec11.pdf
3) https://vaibhavkarve.github.io/satisfiability/index.html
4) CNF-SAT graph problem: https://www.cs.uoi.gr/~cnomikos/courses/cc/Computational-Complexity-Notes.pdf
5) Βιντεο-διαλέξεις του WilliamFiset για επανάληψη πάνω σε βασικούς Αλγορίθμους.