Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργασία 1 - Ανάλυση και Σχεδιασμός Αλγορίθμων

Αθανασιάδου Χριστίνα
athanchris@ece.auth.gr
AEM 10328

Μανακίδης Παύλος
mppavlos@ece.auth.gr
AEM 10436

Χατζηγεωργίου Σπυρίδων
spyrchat@ece.auth.gr
AEM 10527

April 6, 2023

CONTENTS

1 Εισαγωγή

Το παρόν έγγραφο αποτελεί την αναφορά της πρώτης εργασίας του μαθήματος "Ανάλυση και Σχεδιασμός Αλογορίθμων" του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του ΑΠΘ κατά το εαρινό εξάμηνο του ακαδημαϊκού έτους 2022-2023. Η αναφορά συντάχθηκε από την ομάδα 06, της οποίας τα μέλη είναι : Χριστίνα Αθανασιάδου, Παύλος Μανακίδης και Σπυρίδων Χατζηγεωργίου.

2 Πρόβλημα 1

2.1 Πρώτη ερώτηση

Σκοπεύετε να βάλετε υποψηφιότητα στις επερχόμενες εκλογές και επιθυμείτε να οργανώσετε όσο το δυνατό πιο αποτελεσματικά τις περιοδείες σας. Έχετε στα χέρια σας δημοσκοπικά δεδομένα τα οποία σας δίνουν πληροφορία για κάθε χωριό και πόλη της περιφέρειας στην οποία θέτετε υποψηφιότητα. Συγκεκριμένα για κάθε ένα από τα μέρη αυτά, έχετε στα χέρια σας τί δήλωσε κάθε πολίτης, που πήρε μέρος σε δημοσκόπηση, οτι προτίθεται να ψηφίσει (το ονοματεπώνυμο των υποψηφίων, εσάς και των αντιπάλων σας). Θέλετε να γνωρίζετε αν εσείς ή κάποιος άλλος υποψήφιος συγκεντρώνει τουλάχιστον τις μισές ψήφους σε κάθε μέρος.

2.1.1 Εκφώνηση

Σχεδιάστε έναν αλγόριθμο ο οποίος θα εξετάζει αν εσείς ή κάποιος άλλος υποψήφιος συγκεντρώνει τουλάχιστον τις μισές ψήφους σε ένα μέρος σε χρόνο

$$O(n^2) \tag{2.1}$$

Θεωρήστε ότι δεν ξέρετε ούτε τον αριθμό αλλά ούτε και την ταυτότητα των υποψηφίων ώστε ο αλγόριθμος να είναι όσο πιο γενικός γίνεται.

2.1.2 Υλοποίηση αλγορίθμου σε ψευδογλώσσα

```

function checkIfPotentialWinner(candidate, arr):
    counter = 0
    for i = 0 to len(arr) - 1 do
        if arr[i] equals candidate then
            counter = counter + 1
        end if
        if counter is greater than or equal to len(arr) divided by 2 then
            return true
        end if
        else return false
        end else
    end for

end function

function findWinnerCandidate(votesArray):
    n = length of votesArray
    winner1 = 0
    winner2 = 0
```

```
21
22      for i = 0 to n - 1 do
23          if checkIfPotentialWinner(votesArray[i], votesArray) then
24              if winner1 is not equal to 0 and votesArray[i] is not equal to winner1 then
25                  winner2 = votesArray[i]
26              else:
27                  winner1 = votesArray[i]
28              end if
29          end if
30      end for
31      if winner1 is equal to 0 and winner2 is equal to 0 then
32          print "No candidate was voted by more than 50% of the voters"
33      else:
34          return winner1, winner2
35      end if
36  end function
37
38  findWinnerCandidate(givenArray)
39
40
```

2.1.3 Επεξήγηση αλγορίθμου και υπολογισμός πολυπλοκότητας

Ο αλγόριθμος αποτελείται από 2 συναρτήσεις:

- *checkIfPotentialWinner* : Η συνάρτηση αυτή δέχεται σαν όρισμα τον πίνακα με τα ονόματα των υποψηφίων και το όνομα ενός συγκεκριμένου υποψηφίου. Στην πρώτη δομή επανάληψης (γραμμές 6-8 στον ψευδοκώδικα) μετράει πόσες φορές εμφανίζεται το όνομα του υποψηφίου στον πίνακα. Επομένως η πολυπλοκότητα αυτής της συνάρτησης είναι
$$O(n) \tag{2.2}$$
. Τέλος, ελέγχεται αν ο υποψήφιος εμφανίζεται σε τουλάχιστον τις μισές θέσεις του πίνακα.
- *findWinnerCandidate* : Η συνάρτηση αυτή δέχεται σαν όρισμα τον πίνακα με τα ονόματα των υποψηφίων ενός χωριού. Αρχικά ορίζονται οι μεταβλητές winner1 και winner2 για τους 2 πιθανούς νικητές. Στην δομή επανάληψης (γραμμές 18-23) καλείται η checkIfPotentialWinner για κάθε υποψήφιο που εμφανίζεται στον πίνακα και σε περίπτωση που ικανοποιεί το κριτήριο των ψήφων γίνεται η ανάθεση του ονόματός του στον πρώτο νικητή. Εάν υπάρχει και 2ος νικητής, δηλαδή υπάρχουν μόνο 2 υποψήφιοι τοτε γίνεται η ανάθεση του ονόματός του στον winner2. Η συγκεκριμένη δομή επανάληψης καλεί την συνάρτηση checkIfPotentialWinner, η οποία έχει πολυπλοκότητα
$$O(n) \tag{2.3}$$
για κάθε επανάληψη, οπότε πρόκειται για μια εμφωλευμένη for. Άρα συνολικά ο αλγόριθμος έχει πολυπλοκότητα
$$O(n^2) \tag{2.4}$$
.

2.1.4 Υλοποίηση αλγορίθμου σε *python*

Παρακάτω φαίνεται η υλοποίηση του αλγορίθμου σε python:

```python

# CheckIfPotentialWinner function has one for-loop hence the O(n) Complexity
def checkIfPotentialWinner(candidate, arr):  # This function finds out if the candidate
                                             # with has more than 50% of the votes
    counter  = 0
    for i in range(len(arr)):
        if arr[i] == candidate:
            counter = counter + 1
    if counter >= len(arr)/2 :
        return True
    else:
        return False

def findWinnerCandidate(votesArray):    # This function iterates through the votes array
    n = len(votesArray)
    winner1 = 0        # Winner1 is the first candidate that has at least N/2 votes
    winner2 = 0        # Winner2 is the second candidate that has at least N/2 votes
                       # (it exists only if there are two candidates and have half votes)

    for i in range(n):   # Here is the first for-loop that goes through the whole array
        if checkIfPotentialWinner(votesArray[i], votesArray):
            if(winner1 != 0 and votesArray[i] != winner1):  # Check if  a winner exists
                winner2 = votesArray[i]
            else:
                winner1 = votesArray[i]
    if(winner1 == 0 and winner2 == 0):
        print("No candidate was voted by more than 50% of the voters")
    else:
        return winner1,winner2
# findWinnerCandidate function has a for-loop but inside calls a function with O(n)
# Complexity called 'checkIfPotentialWinner' so the Overall Complexity is O(n^2)


print(findWinnerCandidate(arr))

```

2.2 Δεύτερη Ερώτηση

2.2.1 Εκφώνηση

Χρησιμοποιήστε την τεχνική Διαίρει και Βασίλευε για να επιτύχετε το παραπάνω σε χρόνο

$$O(nlogn) \tag{2.5}$$

2.2.2 Υλοποίηση αλγορίθμου σε ψευδογλώσσα

```python

#This function finds the occurences of a potential vote in the given array
#by taking advantage of the recursion tree method of the Divide And Conquer
#algorithm design technique.

```

```
#Inputs: arr-> The array with the potential votes (the input array of the main
#               algorithm) or a subarray of it
#        high-> The highest index of the subarray to be used
#        low-> The lowest index of the subarray to be used
#        key-> The name/vote to be searched
#Output: An integer showing the number of a potential vote's (key) occurence
#        in the recursion tree

function findOccurences(arr, high, low, key):

    # base case (leaf of the tree)

    #if the element of the array matches the key return a base score of 1

    if low equals high and arr[low] equals key:
        return 1
    end if

    #if wrong inputs for high and low were given or
    #the element of the array does not match the key
    #return a base score of 0

    if low > high or (low equals high and arr[low] not equal to x):
        return 0
    end if

    # other cases (not a leaf)

    #calculate the variables left and right as the number of occurences of x
    #in the left and right subtrees (subarrays) respectively
    #then return their sum as the total number of key's occurences in the given subarray

    left = findOccurences(arr, (low+high)//2, low, key)
    right = findOccurences(arr, high, (low+high)//2 +1, key)
    return left + right
end function


#The following is the main function of the algorithm
#Its main goal is to find the candidate who has the majority of the potential votes.
#This is done with respect to three different cases,
#   1-There is one candidate who may gather the majority of votes
#   2-There are two candidates who gather 50% of the votes each
#   3-No candidate gathers the majority of votes

function findMajority(arr):

    if arr.length is not equal to 0:

        #initialize the variables temp1 and temp2 to 'Nan'
```

```
        #temp1 refers to the winner of case 1
        #or one of the winners of case 2
        #temp2 refers to the remaining winner of case 2

        temp1='Nan'
        temp2='Nan'

        #Iterate through the array to find the candidate with the 50%
        #of the votes

        for i=1 to arr.length:
            if findOccurences(arr,arr.length-1,0,arr[i])>=arr.length/2:
                temp1 = arr[i]
            end if
        end for
        #Iterate through the array again to check if another
        #candidate has also 50% of the votes

        for j=1 to arr.length:
            if findOccurences(arr,arr.length-1,0,arr[i]) equals arr.length/2:
                temp2 = arr[j]
            end if
        end for

        #If neither temp1 nor temp2 have changed, there is no
        #candidate that gathers the required number of votes
        #So, the respective message will be shown and the function
        #will return no value

        if temp1 equals 'Nan' and temp2 equals 'Nan':
            print("No candidate has the majority of Votes")
            return
        end if

        #In any other occasion, both temp1 and temp2 will be returned
        #For case 1 only temp1 will contain a name
        #For case 2 both variables will contain a name

        return temp1, temp2
    end if

    else:
        print("No Voters Found")
    end else
end function
findMajority(givenArrayWithVotes) #calling the function
```

2.2.3 Επεξήγηση αλγορίθμου και υπολογισμός πολυπλοκότητας

- ***findOccurencies*** : Η συνάρτηση αυτή βρίσκει το πλήθος των εμφανίσεων μίας πιθανής ψήφου στον πίνακα που δέχεται ως όρισμα εκμεταλλευόμενη την μέθοδο του δέντρου αναδρομής ως μέθοδο της τεχνικής σχεδίασης αλγορίθμων "Διαίρει και Κυρίευε". Ως εισόδους η συνάρτηση δέχεται:
  - έναν πίνακα **arr** με τις πιθανές ψήφους (= ο πίνακας που δίνεται στην είσοδο του βασικού αλγόριθμου) ή έναν υποπίνακα αυτού
  - έναν δείκτη **high** που δείχνει τον υψηλότερο δείκτη του πίνακα **arr** που θα αξιοποιηθεί
  - έναν δείκτη **low** που δείχνει τον χαμηλότερο δείκτη του πίνακα **arr** που θα αξιοποιηθεί
  - ένα κλειδί **key** που αντιστοιχεί στο όνομα για το οποίο θα γίνει η αναζήτηση.

  Ως έξοδο η συνάρτηση δίνει έναν ακέραιο αριθμό ο οποίος αντιστοιχεί στο πλήθος εμφανίσεων του ζητούμενου ονόματος εντός του δέντρου αναδρομής.
  Λειτουργία
  Η συνάρτηση προσομοιώνει ένα δέντρο αναδρομής. Για τον λόγο αυτό, πρώτα ορίζει τη βασική περίπτωση, δηλαδή την περίπτωση ενός φύλλου. Αν το όνομα που θα περιέχεται στο φύλλο είναι ίδιο με το όνομα για το οποίο γίνεται η αναζήτηση θα επιστρέφεται **1**, διαφορετικά **0**. Σε κάθε αλλη περίπτωση, η συνάρτηση χωρίζει τον πίνακα που έχει δεχθεί σε δύο επιμέρους υποπίνακες μισού μεγέθους σε σχέση με τον αρχικά δοσμένο (Διαίρει) και αναδρομικά εξετάζει το πλήθος εμφανίσεων του ονόματος στο αριστερό και δεξί πίνακα-υποδέντρο (Κυρίευε). Τότε, θα επιστρέψει αθροιστικά το αποτέλεσμα των εμφανίσεων στο αριστερό και το δεξί υποδέντρο, γεγονός το οποίο στο τέλος των αναδρομών θα δώσει και το συνολικό πλήθος εμφανίσεων ενός ονόματος στο δέντρο.
- ***findMajority*** : Η συνάρτηση αυτή είναι η βασική συνάρτηση του αλγορίθμου. Στόχος της είναι να εντοπίσει τον υποψήφιο που συγκεντρώνει την πλειοψηφία των ψήφων εντός του δοσμένου πίνακα. Για τον λόγο αυτό εξετάζει τρεις διαφορετικές περιπτώσεις:
  - την ύπαρξη ενός μόνο υποψήφιου που συγκεντρώνει τη ζητούμενη πλειοψηφία
  - την ύπαρξη δύο υποψηφίων που συγκεντρώνουν εξίσου το **50%** των πιθανών ψήφων
  - την περίπτωση στην οποία κανένας υποψήφιος δεν συγκεντρώνει τη ζητούμενη πλειοψηφία.

  Λειτουργία
  Μέσα από δύο προσωρινές μεταβλητές γίνεται η καταχώρηση των ονομάτων που ικανοποιούν το ζητούμενο. Αρχικά, γίνεται η αναζήτηση για έναν νικητή, μέσα από την αναζήτηση των εμφανίσεων κάθε ονόματος στο δέντρο αναδρομών που εξηγήθηκε παραπάνω. Αν υπάρχει υποψήφιος που πληροί τις προϋποθέσεις, αυτός καταχωρείται στην πρώτη μεταβλητή. Στη συνέχεια, ο ίδιος έλεγχος πραγματοποιείται και για έναν πιθανό δεύτερο υποψήφιο, ο οποίος ικανοποιεί το ζητούμενο στο ενδεχόμενο ισοψηφίας. Αυτός, αν υπάρχει, θα καταχωρηθεί στη δεύτερη μεταβλητή. Τελικά, θα επιστραφούν το/τα όνομα/ονόματα των υποψηφίων, όπως αυτά καταχωρήθηκαν στις μεταβλητές. Αν, ωστόσο, δεν έχει αλλάξει καμία μεταβλητή στην πορεία εκτέλεσης του αλγορίθμου, δηλαδή κανένας υποψήφιος δεν ικανοποιεί το ζητούμενο, θα εκτυπωθεί το αντίστοιχο μήνυμα και η συνάρτηση δε θα επιστρέψει τίποτα. Να σημειωθεί ότι στις συναρτήσεις έχουν συμπεριληφθεί και έλεγχοι για αποφυγή σφαλμάτων σε περίπτωση που δοθεί πίνακας μηδενικού μεγέθους ή κάποιοι δείκτες περαστούν εσφαλμένα.

Πολυπλοκότητα
Η συνάρτηση ***findMajority*** διαθέτει δύο ανεξάρτητες επαναληπτικές δομές, η καθεμία εκ των οποίων επαναλαμβάνει **n** φορές τη συνάρτηση ***findOccurencies***, μία φορά για κάθε στοιχείο του πίνακα. Ως μια αναδρομική μορφή δέντρου, η **findOccurencies** είναι πολυπλοκότητας **O(logn)**. Το γεγονός αυτό μπορεί να επαληθευτεί με τη χρήση του θεωρήματος κυρίαρχου όρου (**master theorem**). Συγκεκριμένα, έστω **n** το μέγεθος του δοσμένου πίνακα. Με τη χρήση της **findOccurencies** ο πίνακας χωρίζεται σε δύο επιμέρους υποπίνακες. Άρα, α=2. Κάθε υποπίνακας είναι μεγέθους **n/2**. Άρα **b=2**. Τέλος, ο συνδυασμός των επιμέρους αποτελεσμάτων αποτελείται από προσθέσεις, οι οποίες είναι διαδικασίες σταθερού χρόνου. Άρα, **d=0**.

Παρατηρούμε ότι

$$d = log_b\alpha \tag{2.6}$$

Άρα η findOccurrences είναι πολυπλοκότητας

$$O(logn) \tag{2.7}$$

Τελικά, οι n επαναλήψεις της findOccurencies υποδεικνύουν ότι η βασική συνάρτηση του αλγορίθμου είναι πολυπλοκότητας

$$O(nlogn) \tag{2.8}$$

.

2.2.4 Υλοποίηση αλγορίθμου σε *python*

```python

# findWinnerCandidate function uses Divide and Conquer Algorithm Design Technique and
# utilizes reccursion so we can find the reccurences in O(logn) time Complexity, We
# found The Reccurences and the subproblems using Master Theorem
def findOccurrences(arr, high, low, key):

    if(low == high and arr[low] == key):
        return 1

    if low > high or (low == high and arr[low]!= key):
        return 0


    left = findOccurrences(arr, (low+high)//2, low, key)
    right  = findOccurrences(arr, high, (high+low)//2 + 1, key)
    return left + right

def findMajority(arr):
    if len(arr)!=0:
        temp1 = 'Nan'
        temp2 = 'Nan'
        print(arr)
        #Iterate through the array to find the Candidate with the 50% of the votes
        for i in range(len(arr)):
            if findOccurrences(arr,len(arr)-1,0,arr[i]) >= len(arr)/2 :
                temp1 = arr[i]
        #Iterate through the array again to make sure there isn't
        #a second candidate that has 50% of the votes
        for j in range(len(arr)):
            if  findOccurrences(arr,len(arr)-1,0,arr[i])==len(arr)/2 and arr[j] != temp1:
                temp2 = arr[j]
        if temp1 == 'Nan' and temp2 == 'Nan':
            print("No Candidate has the majority of the Votes")
            return

        return temp1,temp2
    else :
```

```
        print("No Voters Found")

# The findMajority function has two independent for-loops that each call a function
# called'findOccurences' with O(logn) Complexity,so the call of an O(logn) function
# inside the loop, makes the overall complexity O(nlogn)

print(findMajority(arr))

```

2.3 Τρίτη Ερώτηση

2.3.1 Εκφώνηση

Υπάρχει αλγόριθμος που να επιτυγχάνει την ίδια εργασία σε χρόνο

$$O(n) \tag{2.9}$$

;

2.3.2 Υλοποίηση αλγορίθμου σε ψευδογλώσσα

```

# This function finds the candidate with the most votes
# It uses the Boyer-Moore Majority Vote Algorithm
# The time complexity of this function is O(n)

function findCandidate(arr):
    if length of arr is not 0
        candidate = -1
        votes = 0
        for i in range(length of arr)
            if votes is 0
                candidate = arr[i]
                votes = 1
            end if
            else
                if arr[i] is candidate
                    votes = votes + 1
                end if
                else
                    votes = votes - 1
                end else
            end else
        end for
        return candidate
    end if
    else
        print "The input array cannot be of zero length"
        return None
    end else
end function

```

```
# This function checks if the candidate with the most votes has a majority in the array
# It has a time complexity of O(n)

function checkForMajority(candidate, array)
    ctr = 0
    for j in range(length of array)
        if array[j] is candidate:
            ctr = ctr + 1
        end if
    end for
    if ctr >= length of array / 2
        return True
    end if
    else
        return False
    end else
end function

# This function checks if there is a tie between two candidates with the most votes
# It first finds the candidate with the most votes using the findCandidate function
# Then it checks if that candidate has a majority in the array using CheckForMajority
# It creates a new array without the first candidate and finds the candidate with the
# most votes in that array It then checks if the second candidate has a majority
# in the original array using CheckForMajority
# The time complexity of this function is O(n) in the worst case, when
# there are two candidates with equal votes

function announceWinners(arr)
    cnd = findCandidate(arr)
    if checkForMajority(cnd, arr) is True
        arr2 = []
        for i in range(length of arr)
            if arr[i] is not cnd
                arr2.append(arr[i])
            end if
        end for

        cnd2 = findCandidate(arr2)

        if checkForMajority(cnd2, arr) is True
            return cnd, cnd2
        end if
        else
            return cnd
        end else
    end if
    else
        return None
    end else
end function
```

```python
print(announceWinners(votesarray)
```

2.3.3 Επεξήγηση αλγορίθμου και υπολογισμός πολυπλοκότητας

Γενική Ιδέα:

Για το συγκεκριμένο πρόβλημα υλοποιήθηκε μια παραλλαγή του αλγορίθμου Boyer-Moore, τροποποιημένη, έτσι ώστε να λαμβάνει υπόψη και την περίπτωση που υπάρχουν δύο νικητές. Πιο συγκεκριμένα, υλοποιήθηκε μια συνάρτηση και 2 βοηθητικές συναρτήσεις. Η βασική συνάρτηση είναι η announeWinners και λαμβάνει ως input έναν πίνακα και έπειτα καλεί τις δύο βοηθητικές συναρτήσεις. Η πρώτη είναι η findCandidate, η οποία βρίσκει το υποψήφιο δημοφιλέστερο στοιχείο του πίνακα και η δεύτερη είναι η checkForMajority η οποία καλείται για να αποφανθεί για το αν το υποψήφιο δημοφιλέστερο στοιχείο του πίνακα, είναι και αυτό με την πλειοψηφία.

Ανάλυση:

- Η πρώτη συνάρτηση που ορίζεται είναι η findCandidate, η οποία έχει ως όρισμα έναν πίνακα και ως σκοπό να επιστρέψει το υποψήφιο συχνότερο στοιχείο. Η ιδέα είναι ότι τα στοιχεία που είναι διαφορετικά μεταξύ τους αλληλοαναιρούνται, μέχρι να μείνει μόνο ένα στοιχείο. Για την υλοποίηση, υπάρχουν δύο μεταβλητές, μια που θα αρχικοποιεί ένα στοιχείο του πίνακα (στο εξής υποψήφιος) και μία ακόμα η οποία θα λειτουργεί ως μετρητής. Επιπλέον, χρειάζεται και μια συνθήκη ελέγχου, η οποία θα ελέγχει αν ο μετρητής έχει φτάσει στο μηδέν, μέσα σε μια επαναληπτική δομή, η οποία κάνει μια γραμμική προσπέλαση του πίνακα. Αναλυτικότερα, όταν ο μετρητής είναι αρχικοποιημένος στην τιμή μηδέν, τότε ο υποψήφιος θα πρέπει να πάρει την τιμή που έχει το i-στο στοιχείο του πίνακα, ενώ ο μετρητής θα πρέπει να αυξηθεί κατά 1. Στην επόμενη επανάληψη, θα γίνει έλεγχος για το αν ο υποψήφιος είναι ίδιος με το [i+1] στοιχείο του πίνακα. Σε αυτή την περίπτωση θα αυξηθεί ο μετρητής, ενώ σε διαφορετική περίπτωση θα μειωθεί. Με αυτόν τον τρόπο, ο μετρητής λειτουργεί ως μια "μνήμη", ώστε να υπολογίζεται πόσες φορές εμφανίζεται ο υποψήφιος από τη στιγμή που αρχικοποιήθηκε μέχρι να χρειαστεί να οριστεί νέος υποψήφιος αν ο μετρητής φτάσει στο μηδέν. Έτσι όταν τελειώσει η επαναληπτική δομή, ο υποψήφιος θα είναι το μοναδικό στοιχείο που θα μπορεί εν δυνάμει να έχει την πλειοψηφία. Εδώ πρέπει να επισημανθεί ότι δεν είναι απαραίτητο ο υποψήφιος να εμφανίζεται στον πίνακα περισσότερες από N/2 φορές (όπου N το πλήθος των στοιχείων του πίνακα), παρόλα αυτά ο αλγόριθμος εγγυάται ότι αν υπάρχει κάποιο στοιχείο το οποίο εμφανίζεται για περισσότερες από N/2 φορές, αυτό το στοιχείο θα είναι υποχρεωτικά ο υποψήφιος.
- Η επόμενη συνάρτηση που χρειάζεται να κληθεί είναι η checkForMajority, η οποία έχει ως ορίσματα τον υποψήφιο και έναν πίνακα με τα ονόματα των υποψηφίων. Όπως προαναφέρθηκε, ο υποψήφιος που υπολογίστηκε προηγουμένως δεν είναι απαραίτητα αυτός με την πλειοψηφία, δηλαδή αυτός που εμφανίζεται τουλάχιστον N/2 φορές στον πίνακα. Για αυτόν τον λόγο, απαιτείται μια συνάρτηση, η οποία θα κάνει μια γραμμική προσπέλαση του πίνακα μέσα από μια επαναληπτική δομή που θα μετράει με έναν μετρητή πόσες φορές εμφανίζεται στον πίνακα ο υποψήφιος. Τέλος αν ο μετρητής είναι μεγαλύτερος ή ίσος με το (μήκος του πίνακα) / 2, η συνάρτηση επιστρέφει λογική κατάφαση, ειδάλλως επιστρέφει λογική άρνηση.
- Η τελευταία συνάρτηση για να ολοκληρωθεί η υλοποίηση του αλγορίθμου είναι η announceWinners. Η συγκεκριμένη παίρνει ως όρισμα πίνακα και καλεί τις δύο προηγούμενες συναρτήσεις. Αναλυτικότερα καλεί την findCandidate και αποθηκεύει τον candidate σε μια μεταβλητή. Σε μια δομή ελέγχου καλεί την CheckForMajority και αν υπάρχει κατάφαση, τότε δημιουργεί έναν δεύτερο πίνακα τον οποίο και αρχικοποιεί με όλα τα στοιχεία του πίνακα που δόθηκε ως όρισμα στην βασική συνάρτηση, εκτός από τον υποψήφιο που αποθηκεύτηκε προηγουμένως. Με αυτόν τον τρόπο μπορεί

να ξαναγίνει κλήση της findCandidate και της checkForMajority για τον καινούργιο πλέον πίνακα. Έτσι στην περίπτωση που υπάρχουν μόνο δύο υποψήφιοι και υπάρχει ισοψηφία θα υπάρχει και δεύτερος νικητής τον οποίο θα επιστρέψει η συνάρτηση μαζί με τον πρώτο. Ακολουθούν κατάλληλες εκτυπώσεις. Τέλος για να ολοκληρωθεί η άσκηση θα γίνει κλήση της συνάρτησης announceWinners με όρισμα τον πίνακα με τις ψήφους και θα ακολουθήσουν όλοι οι υπολογισμοί που περιγράφηκαν παραπάνω.

Ανάλυση Χρονικής Πολυπλοκότητας**:**
Για τον υπολογισμό της χρονικής πολυπλοκότητας, θα υπολογίσουμε για κάθε συνάρτηση του αλγορίθμου την πολυπλοκότητα ξεχωριστά και κάνοντας χρήση της αρχής της υπέρθεσης, θα αθροίσουμε τις επιμέρους πολυπλοκότητες. Αρχικά η συνάρτηση findCandidate έχει μια δομή ελέγχου η οποία χρειάζεται σταθερό χρόνο O(1) και μια δομή επανάληψης η οποία κάνει γραμμική προσπέλαση όλου του πίνακα. Επειδή μέσα στην επαναληπτική δομή υπάρχουν μόνο δομές ελέγχου σταθερού χρόνου, προκύπτει ότι η συνολική χρονική πολυπλοκότητα της δομής επανάληψης και κατά συνέπεια της ίδιας της συνάρτησης, είναι O(n). Συνεχίζοντας η συνάρτηση checkForMajority, διαθέτει μία επαναληπτική δομή που κάνει επίσης γραμμική προσπέλαση του πίνακα, επομένως έχει πολυπλοκότητα O(n). Οι υπόλοιπες δομές ελέγχου της συνάρτησης γίνονται σε σταθερό χρόνο οπότε συνολικά η πολυπλοκότητα της συνάρτησης είναι O(n). Τέλος, η μέθοδος announceWinners έχει μόνο μια δομή επανάληψης η οποία έχει χρονική πολυπλοκότητα O(n). Όλες οι άλλες δομές συμβαίνουν σε σταθερό χρόνο, άρα συνολικά η πολυπλοκότητα είναι και πάλι O(n). Καταληκτικά, η χρονική πολυπλοκότητα του αλγορίθμου, είναι O(n) + O(n) + O(n) = O(n).

2.3.4 Υλοποίηση αλγορίθμου σε *python*

Παρακάτω φαίνεται η υλοποίηση του αλγορίθμου σε python:

```python

#This Function is a variation of the Boyer-Moore Majority Algorithm that
#takes and array as input and finds the most popular element in O(n) Complexity
def findCandidate(arr):
    if len(arr) != 0:
        candidate = -1
        votes = 0

        for i in range(len(arr)):
            if votes == 0:
                candidate = arr[i]
                votes = 1
            else :
                if arr[i] == candidate:
                    votes = votes + 1
                else:
                    votes = votes - 1
        return candidate
    else:
        print("The input array cannot be of zero length")
        return None


#This Function Checks if The most popular
#Element of the Array also has the #Majority in O(n) complexity
def checkForMajority(candidate,array):
```

```python
    ctr = 0
    for j in range(len(array)):
        if array[j] == candidate:
            ctr = ctr + 1
    if ctr >= len(array) / 2:
        return True
    else :
        return False

#This Function is needed to cover the case that 2 winners exist It creates
#a new array without the candidate element
#the previous functions calculated, and checks #for a second Candidate, then
#Checks if The 2nd Candidate has 50% of the Votes
def announceWinners(arr):
    cnd = findCandidate(arr)
    print(cnd)
    print(arr)
    if checkForMajority(cnd,arr) == True:
        arr2 = []
        for i in range(len(arr)):
            if arr[i] != cnd:
                arr2.append(arr[i])

        cnd2 = findCandidate(arr2)
        if checkForMajority(cnd2,arr) == True:
            return cnd,cnd2
        else :
            return cnd
    else:
        return None



print(announceWinners(arr))

```

3 Πρόβλημα 2

Έστω πίνακας T με στοιχεία n θετικούς ακεραίους με εύρος [0, ..., k] (k ακέραιος). Δίνεται ο αλγόριθμος:

```
for i = 0, . . . , k do
    H[i] = 0
end for
for j = 1, . . . , n do
    H[T[j]] = H[T[j]] + 1
end for
for i = 1, . . . , k do
    H[i] = H[i] + H[i - 1]
end for
for j = n, . . . , 1 do
```

```
11        S[H[T[j]]] = T[j]
12        H[T[j]] = H[T[j]] - 1
13    end for
```

3.1 Πρώτη ερώτηση

3.1.1 Εκφώνηση

Περιγράψτε τον πίνακα S.

3.1.2 Απάντηση

Προκειμένου να περιγραφεί ο πίνακας S, είναι απαραίτητο να γίνει σαφής περιγραφή του αλγορίθμου και των εντολών-βημάτων του.

- Γραμμές **1-3**
  - Πρώτη επαναληπτική δομή
  - Αρχικοποίηση Πίνακα H

  Αρχικά, θεωρείται ότι έχει δημιουργηθεί ένας πίνακας H με τόσες θέσεις όσες και το πλήθος των ακέραιων αριθμών στο σύνολο [0,...,k], δηλαδή k+1. Με τη δομή αυτή αρχικοποιούνται όλα τα στοιχεία του πίνακα H στην τιμή 0.
- Γραμμές **4-6**
  - Δεύτερη επαναληπτική δομή
  - Καταγραφή συχνοτήτων εμφάνισης αριθμών του πίνακα T

  Εδώ, τα στοιχεία T[j] του πίνακα T χρησιμοποιούνται ως δείκτες για τον πίνακα H και με κάθε εκτέλεση της επανάληψης θα πραγματοποιείται επαύξηση του στοιχείου του πίνακα H στη θέση T[j] κατά 1. Ουσιαστικά, γίνεται μέσα στον πίνακα η καταγραφή της συχνότητας εμφάνισης κάθε ακεραίου μέσα στον δοσμένο πίνακα T και στο τέλος όλων των επαναλήψεων ο πίνακας H θα περιλαμβάνει στις θέσεις 1,2,3,...,k,k+1 τη συχνότητα εμφάνισης των ακεραίων 0,1,2,3,...,k αντίστοιχα.
- Γραμμές **7-9**
  - Τρίτη επαναληπτική δομή
  - Καταγραφή του αθροίσματος της συχνότητας ενός αριθμού και των προηγούμενών του

  Στο τέλος της λούπας κάθε στοιχείο H[i] του πίνακα H θα περιλαμβάνει το άθροισμα της συχνότητας εμφάνισης του αριθμού i με τη συνολική συχνότητα εμφάνισης όλων των αριθμών x τέτοιων ώστε 0<=x<i.
  Σε κάθε περίπτωση, τα στοιχεία του πίνακα θα είναι τοποθετημένα σε αύξουσα (όχι απαραίτητα γνησίως) σειρά και αναμένεται το τελευταίο στοιχείο του πίνακα (H[k]) να είναι ίσο με n, όσο δηλαδή και το πλήθος των ακεραίων στον πίνακα T.
- Γραμμές **10-13**
  - Τέταρτη επαναληπτική δομή
  - Ταξινόμηση του πίνακα T[]

  Κάθε επανάληψη της δομής αυτής πραγματοποιεί τις εξής ενέργειες:
  - Αναθέτει το στοιχείο T[j] του πίνακα T στη θέση του πίνακα S που υποδεικνύει ο δείκτης H[T[j]]. Όσο πιο μεγάλο είναι το T[j], τόσο πιο μεγάλο θα είναι το H[T[j]] (λόγω της αύξουσας σειράς) και άρα τόσο πιο «δεξιά» θα τοποθετείται αυτό στον πίνακα S.
  - Μειώνει το H[T[j]] κατά 1. Το γεγονός αυτό εξυπηρετεί την τοποθέτηση ίδιων αριθμών που πιθανώς να περιλαμβάνονται μέσα στον πίνακα T σε διαδοχικές θέσεις, ώστε να αποφευχθούν φαινόμενα επικάλυψης (και κατά συνέπεια ύπαρξης κενών θέσεων στον πίνακα). Με άλλα λόγια, αν μέσα στον πίνακα T ο αριθμός x εμφανίζεται 2 φορές, τότε την πρώτη φορά που θα

χρησιμοποιηθεί, θα καταχωρηθεί στη θέση του πίνακα που του αντιστοιχεί και τη δεύτερη φορά θα καταχωρηθεί μία θέση πίσω.

Τελικά, ο πίνακας S θα περιλαμβάνει τα στοιχεία του πίνακα T σε αύξουσα σειρά, δηλαδή ο αλγόριθμος 1 είναι ένας αλγόριθμος ταξινόμησης των στοιχείων ενός πίνακα.

Παρατήρηση 1

Ουσιαστικά, ο πίνακας H με την ολοκλήρωση της 3ης επαναληπτικής δομής είχε διαμορφωθεί με τέτοιο τρόπο ώστε κάθε στοιχείο του να δείχνει τη θέση του πίνακα S στην οποία αντιστοιχεί το T[j] στην τοποθέτηση σε αύξουσα σειρά.

Προκειμένου να γίνει καλύτερα αντιληπτή η παραπάνω ανάλυση, αξίζει να γίνει η χρήση ενός παραδείγματος.
Έστω ο πίνακας T=[3 1 2 2 0]. Για τον πίνακα αυτό, ο οποίος είναι 5 ακεραίων με μέγιστο των αριθμό 3, θα ισχύει n=5 και k=3.

- Γραμμές **1-3**
  Μετά την εκτέλεση αυτού του τμήματος κώδικα θα προκύψει ο πίνακας με k+1=4 στοιχεία:

  H = [0 0 0 0]

- Γραμμές **4-6**
  Ο πίνακας H στο τέλος θα είναι ο εξής:

  H = [1 1 2 1]

  Εύκολα φαίνεται ότι εμφανίζονται κατά σειρά οι συχνότητες εμφάνισης των στοιχείων 0,1,2,3 στον πίνακα T.
- Γραμμές **7-9**
  Μετά από τον βρόχο ο πίνακας H θα είναι ο εξής:

  H = [1 2 4 5]

  Σύμφωνα με την παραπάνω ανάλυση, αναμένουμε στη θέση 1 του πίνακα S να τοποθετηθεί το 0, στη δεύτερη το 1, στην τέταρτη το 2 (όπως και στην τρίτη θέση, αφού το 2 εμφανίζεται δύο φορές) και στην πέμπτη το 3.

j	T[j]	H[T[j]]	S[H[T[J]]]	S	H
5	0	1	0	[0 - - - -]	H[0] - -
4	2	4	2	[0 - - 2 -]	H[2] - -
3	2	3	2	[0 - 2 2 -]	H[2] - -
2	1	2	1	[0 1 2 2 -]	H[1] - -
1	3	5	3	[0 1 2 2 5]	H[3] - -

3.2 Δεύτερη ερώτηση

Αναλύστε το χρόνο εκτέλεσης του Αλγορίθμου 1.

3.2.1 Απάντηση

Για να γίνει η παρακάτω ανάλυση, έχει θεωρηθεί ότι τόσο οι αναθέσεις των στοιχείων στους πίνακες όσο και οι προσθέσεις είναι διαδικασίες σταθερού χρόνου. Επίσης, η ανάλυση της πολυπλοκότητας του αλγορίθμου, γενικά, αφορά τη χειρότερη περίπτωση. Απλώς, η φύση του αλγορίθμου είναι τέτοια που σε κάθε περίπτωση θα εκτελεστούν όλες οι επαναλήψεις κάθε βρόχου.

- Ο πρώτος βρόχος θα εκτελεί k+1 φορές μία διαδικασία σταθερού χρόνου. Επομένως, με βάση το BigO notation, θα έχει πολυπλοκότητα O(k+1)=O(k) (εκτελείται σε γραμμικό χρόνο).
- Ο δεύτερος βρόχος θα εκτελεί n φορές μία διαδικασία σταθερού χρόνου. Άρα, θα έχει πολυπλοκότητα O(n) (γραμμικός χρόνος).
- Ο τρίτος βρόχος θα εκτελεί k φορές μία διαδικασία σταθερού χρόνου. Άρα, θα έχει πολυπλοκότητα O(k) (γραμμικός χρόνος).
- Τέλος, ο τέταρτος βρόχος θα εκτελεί n φορές μία διαδικασία σταθερού χρόνου. Άρα, θα έχει πολυπλοκότητα O(n) (γραμμικός χρόνος).

Συνολικά, η χρονική πολυπλοκότητα του αλγορίθμου 1 θα είναι:

$$\mathbf{O}(\mathbf{k}) + \mathbf{O}(\mathbf{n}) + \mathbf{O}(\mathbf{k}) + \mathbf{O}(\mathbf{n}) = \mathbf{O}(\mathbf{2k} + \mathbf{2n}) = \mathbf{O}(\mathbf{k} + \mathbf{n}) \tag{3.1}$$

Παρατήρηση **2**

Παρατηρούμε δηλαδή ότι παρατηρείται γραμμικότητα στον χρόνο.

4 GitHub Repository

Ο κώδικας σε python, καθώς και test που χρησιμοποιήθηκαν για τον έλεγχο της λειτουργικότητας των αλγορίθμων υπάρχει στο GitHub Repository.