ASSIGNMENT 2 Complex Data Management on Spatial Data

ΙΩΑΝΝΑ ΠΑΠΑΓΙΑΝΝΗ 2790

Η γλώσσα προγραμματισμού που χρησιμοποιήθηκε και για τα τρια μέρη της άσκησης είναι python3.
Επιπλέον το αρχείο “Beijing_restaurants.txt” είναι κατεβασμένο ήδη και έχει τοποθετηθεί στον ίδιο φάκελο με τα αρχεία part1.py, part2.py και part3.py.

PART 1

Στο μέρος 1 ο κώδικας θέλουμε να υλοποιεί ένα απλό χωρικό ευρετήριο βασισμένο σε grid.

if __name__ == "__main__":

```
coordList=[]
boundaries=findLimits()
theGrid(coordList, boundaries)
```

Όπου coordList η λίστα από λίστες που θα κρατάμε τα “φτιαγμένα” δεδομένα από το αρχείο *Beijing_restaurants.txt.*
Η main αρχικά καλεί την *findLimits():*

def findLimits():

```
with open('Beijing_restaurants.txt', 'r', encoding='UTF-8') as df1:

        lineNum=0
        maxX=0
        maxY=0
        minX=200
        minY=200

        try:
                df1.__next__()

        except StopIteration:
                print('StopIteration')
                sys.exit(1)

        for row in df1:

                lineNum+=1
                fixedData=fixData(lineNum, row)
                coordList.append(fixedData)

                if fixedData[1]>maxX:
                        maxX=fixedData[1]
                elif fixedData[1]<minX:
                        minX=fixedData[1]
                if fixedData[2]>maxY:
                        maxY=fixedData[2]
                elif fixedData[2]<minY:
                        minY=fixedData[2]

        return [minX, maxX, minY, maxY]
```

Ανοίγει το αρχείο *“Beijing_restaurants.txt”* για ανάγνωση “r” και αρχικοποιεί ένα αναγνωριστικό αριθμό *lineNum* που αντιστοιχεί στη γραμμή στην οποία βρίσκεται το σημείο στο “*Beijing_restaurants.txt*” στο 0.

Τα άνωτερα όρια των τιμών των συντεταγμένων x, y στο 0 (*maxX=0 maxY=0*), ξεκινάμε με πολύ λιγότερο δηλαδή και από το κατώτερο όριο των δωθέντων σημείων, ώστε σίγουρα οι τιμές των x και αντίστοιχα y, στο αρχείο να είναι μεγαλύτερες για να μπορέσει να “μπεί” στις παρακάτω συνθήκες (if) και να βρει το πραγματικό maxX maxY.

Όμοια και για τα κατώτερα όρια των τιμών των συντεταγμένων x, y στο 200 (*minX=200 minY=200*), δηλαδή πολύ μεγαλύτερες και από το μεγαλύτερο όριο των δωθέντων σημείων, ώστε όλες οι τιμές των x και αντίστοιχα y, στο αρχείο να είναι μικρότερες.

Στη συνέχεια, δοκιμάζουμε να πάμε στην επόμενη γραμμή (*try: ...except StopIteration:)* γιατί η πρώτη απλά περιέχει το πλήθος των σημείων που ακολουθούν στις επόμενες γραμμές.

Για κάθε γραμμή στο αρχείο, αυξάνουμε το *lineNum* κατά 1, και καλούμε την *fixedData()* για να φέρουμε τα δεδομένα μας σε μορφή κατάλληλη για επεξεργασία.

---

def fixData(lineNum, line):

```
rawData=line.split(' ')
splitXY=[i.split(',') for i in rawData]
```

floatData= [float(j) for i in splitXY for j in i]
floatData.insert(0,lineNum)

return floatData

Η συγκεκριμένη συνάρτηση παίρνει ως όρισμα τον αναγνωριστικό αριθμό γραμμής του αρχείου "*Beijing_restaurants.txt"* και τα δεδομένα της αντίστοιχης γραμμής.

Πρώτα τα κάνει split με βάση το space *rawData=line.split(' ')*
πχ: ['39.865999', '116.26745\n']

Στη συνέχεια τα κάνει split με βάση το comma *splitXY=[i.split(',') for i in rawData]*
πχ: [['39.865999'], ['116.26745\n']]

Έπειτα τα μετατρέπει από string σε float *floatData= [float(j) for i in splitXY for j in I]*
πχ: [39.865999, 116.26745]

Και τέλος πριν τα επιστρέψει, προσθέτει και στην αρχή της λίστας τον αναγνωριστικό αριθμό γραμμής του αρχείου
floatData.insert(0,lineNum)
πχ: [51959, 39.865999, 116.26745]

Γυρίζουμε το αποτέλεσμα με το *return floatData* στην *findLimits(),* και κάνουμε append στην coordList (*coordList.append(fixedData)*).

Αν το δεύτερο στοιχείο της λίστας *fixedData* (οι συντεταγμένη του x) είναι μεγαλύτερο από το τωρινό *maxX* κάνουμε αυτό να είναι το *maxX* αλλίως αν είναι μικρότερο από το τωρινό *minX* κάνουμε αυτό να είναι το *minX.*

```
if fixedData[1]>maxX:
        maxX=fixedData[1]
    elif fixedData[1]<minX:
        minX=fixedData[1]
```

Αντίστοιχα δουλεύουμε και για τα y coordinates παίρνοντας τώρα το τρίτο στοιχείο της λίστας *fixedData* (οι συντεταγμένη του y)

```
if fixedData[2]>maxY:
        maxY=fixedData[2]
    elif fixedData[2]<minY:
        minY=fixedData[2]
```

Αφού ελέγξουμε όλες τις γραμμές του αρχείου και έχουμε πλέον βρει τα *minX, maxX, minY, maxY* τα επιστρέφουμε στην *main* με την μορφή μιας λίστας στο *boundaries*.

Η *main* καλεί μετά την συνάρτηση *theGrid():*

def theGrid(cList, boundaries):

```
    dividedRangeX=(boundaries[1]-boundaries[0])/10
    dividedRangeY=(boundaries[3]-boundaries[2])/10

    cellWithElements=0
    firstTimeInCell=1
    belongsToCell=[]
    placeInGrd=0
    numOfRestaurants=0
    ....
```

Παίρνει ως όρισμα την λίστα με τις λίστες από τα fixedData του αρχείου "*Beijing_restaurants.txt"* και τα όρια,

Πρώτα βρίσκουμε το εύρος τιμών των συντεταγμένων μας αφαιρώντας από το max το min και το διαίρουμε με 10 ώστε να βρούμε τα ίσα διαστήματα τιμών για το grid τόσο του x όσο και του y αντίστοιχα.

Αρχικοποιούμε στο 0 τη μεταβλήτη για το αν υπάρχουν σημεία στο κελί (*cellWithElements* σαν "flag" παίρνει τιμές 0 ή 1*)*, το πλήθος-αριθμό των σημείων στο κελί (*numOfRestaurants)* και τη θέση στο αρχείο "*grid.grd"* η οποία περιέχει το πρώτο σημείο στο κελί (δηλαδή πόσους χαρακτήρες πρέπει να διαβάσει από την αρχή του αρχείου "*grid.grd"* για να φτάσει στο πρώτο εστιατόριο του κάθε κελιού).

Αρχικοποιούμε στο 1 το "flag" για το αν είναι το πρώτο στοιχείο που ανήκει στο συγκεκριμένο κελί (*firstTimeInCell).*

Ορίζουμε ως λίστα *belongsToCell[]* αυτήν, που θα κρατάει τις γραμμές με τα σημεία που ανήκουν στο συγκεκριμένο κελί.

...
with open('grid.grd', 'w+', encoding='UTF-8') as dfgrd, open('grid.dir', 'w+', encoding='UTF-8') as dfdir:

dfdir.write('%s %s %s %s\n' % ('{0:.6f}'.format(boundaries[0]), '{0:.6f}'.format(boundaries[1]), '{0:.6f}'.format(boundaries[2]),

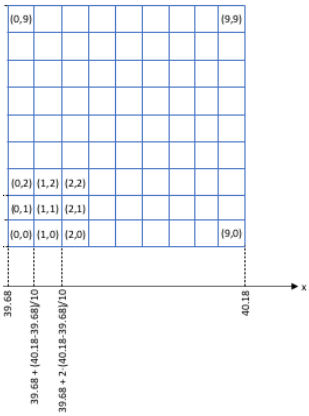
```
                                                                    '{0:.6f}'.format(boundaries[3])))
for x in range(10):
  for y in range(10):
    for sublist in cList:
      if ((sublist[1]>=boundaries[0]+(x*dividedRangeX) and sublist[1]<boundaries[0]+((x+1)*dividedRangeX)) or
            (x==9 and sublist[1]==boundaries[1])) and ((sublist[2]>=boundaries[2]+(y*dividedRangeY) and
            sublist[2]<boundaries[2]+((y+1)*dividedRangeY)) or (y==9 and sublist[2]==boundaries[3])):

        cellWithElements=1
        numOfRestaurants+=1
        belongsToCell.insert(len(belongsToCell), sublist)

        if firstTimeInCell==1:

            firstRestaurant=[x,y,placeInGrd]
            firstTimeInCell=0

        with6decx='{0:.6f}'.format(sublist[1])
        with6decy='{0:.6f}'.format(sublist[2])
        placeInGrd+=len(str(sublist[0]))+len(with6decx)+len(with6decy)+3

    dfgrd.writelines('%s %s %s\n' % (str(i[0]), '{0:.6f}'.format(i[1]), '{0:.6f}'.format(i[2]))  for i in belongsToCell)

    if cellWithElements==1:
        firstRestaurant.insert(len(firstRestaurant), numOfRestaurants)
        dfdir.write('%s %s %s %s\n' % (str(firstRestaurant[0]), str(firstRestaurant[1]), str(firstRestaurant[2]), str(firstRestaurant[3])))

    belongsToCell=[]
    numOfRestaurants=0
    firstTimeInCell=1
    cellWithElements=0
```

Ανοίγουμε τα αρχεία “*grid.grd*” , “*grid.dir*” για γράψιμο και διάβασμα (w+).

Στο “*grid.dir*” γράφουμε την πρώτη γραμμή που είναι ήδη γνωστή είναι τα boundaries (την ελάχιστη και τη μέγιστη τιμή σε κάθε άξονα). Χρησιμοποιούμε το *'{0:.6f}'.format* που μετατρέπει τα floats σε strings με 6 δεκαδικά ψηφία. Βάζουμε και *\n* ώστε να γράψουμε στην επόμενη γραμμή στη συνέχεια.

Ξεκινάμε να φτιάχνουμε το grid.
Έχουμε μια for loop για τον x άξονα και μέσα της μια for loop για τον y άξονα. Εκεί εσωτερικά, αρχίζουμε να ψάχνουμε (for loop) για κάθε υπολίστα της λίστας που έχουμε από το αρχείο “*Beijing_restaurants.txt*” αν ισχύει η απαραίτητη συνθήκη ώστε να ανήκει στο κελί.

Κοιτάμε τα φράγματα για το x_coordinate :



Θέλουμε το συγκεκριμένο x_coordinate της υπολίστας να είναι **μεγαλύτερο ή ίσο** από το minX, αυξημένο κατά τον αριθμό του συγκεκριμένου x κελιού (x) και πολλαπλασιασμένο με το εύρος του κελιού (dividedRangeX)

ΚΑΙ

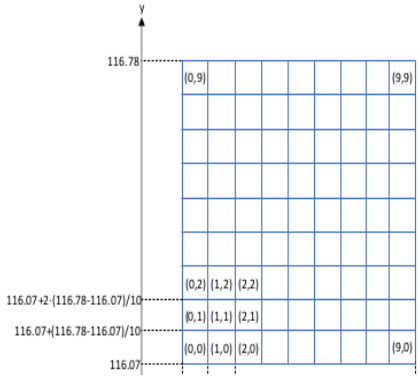
το συγκεκριμένο x_coordinate της υπολίστας να είναι αυστηρά **μικρότερο** (ώστε αν ένα σημείο πέφτει ακριβώς σε μια διαχωριστική γραμμή να αντιστοιχίζεται στο κελί που ακολουθεί) από το minX, αυξημένο κατά τον αριθμό του επόμενου x κελιού(x+1) και πολλαπλασιασμένο με την το εύρος του κελιού(dividedRangeX) .

ΚΑΛΥΠΤΕΙ ΟΛΕΣ ΤΙΣ ΠΕΡΙΠΤΩΣΕΙΣ ΓΙΑ ΤΑ X_COORDINATES ΕΚΤΟΣ ΤΗΣ ΠΕΡΙΠΤΩΣΗΣ ΠΟΥ ΒΡΙΣΚΟΜΑΣΤΕ ΣΕ X_COORDINATE ΠΟΥ ΠΕΦΤΕΙ ΣΤΗΝ ΑΚΡΙΑΝΗ ΔΙΑΧΩΡΙΣΤΙΚΗ ΓΡΑΜΜΗ ΤΟΥ ΑΞΟΝΑ X (ΓΙΑΤΙ ΕΚΕΙ ΔΕΝ ΘΑ ΕΧΕΙ ΚΕΛΙ ΝΑ ΑΚΟΛΟΥΘΗΣΕΙ ΩΣΤΕ ΝΑ ΜΠΕΙ ΣΤΟ ΕΠΟΜΕΝΟ).

Γι’αυτο, έχουμε και το **OR** ώστε είτε να ισχύει το παραπάνω είτε να **βρισκόμαστε στο ακριανό x_cell** του grid (*x==9*) και το συγκεκριμένο x_coordinate της υπολίστας να είναι **ίσο με το maxX.**

ΚΑΙ...

Όμοια κοιτάμε τα φράγματα για το y_coordinate :



Θέλουμε το συγκεκριμένο y_coordinate της υπολίστας να είναι **μεγαλύτερο ή ίσο** από το minY, αυξημένο κατά τον αριθμό του συγκεκριμένου y κελιού (y) και πολλαπλασιασμένο με το εύρος του κελιού (dividedRangeY)

ΚΑΙ

το συγκεκριμένο y_coordinate της υπολίστας να είναι αυστηρά **μικρότερο** (ώστε αν ένα σημείο πέφτει ακριβώς σε μια διαχωριστική γραμμή να αντιστοιχίζεται στο κελί που ακολουθεί) από το minY, αυξημένο κατά τον αριθμό του επόμενου y κελιού(y+1) και πολλαπλασιασμένο με την το εύρος του κελιού(dividedRangeY) .

ΚΑΛΥΠΤΕΙ ΟΛΕΣ ΤΙΣ ΠΕΡΙΠΤΩΣΕΙΣ ΓΙΑ ΤΑ Y_COORDINATES ΕΚΤΟΣ ΤΗΣ ΠΕΡΙΠΤΩΣΗΣ ΠΟΥ ΒΡΙΣΚΟΜΑΣΤΕ ΣΕ Y_COORDINATE ΠΟΥ ΠΕΦΤΕΙ ΣΤΗΝ ΑΚΡΙΑΝΗ ΔΙΑΧΩΡΙΣΤΙΚΗ ΓΡΑΜΜΗ ΤΟΥ ΑΞΟΝΑ Y (ΓΙΑΤΙ ΕΚΕΙ ΔΕΝ ΘΑ ΕΧΕΙ ΚΕΛΙ ΝΑ ΑΚΟΛΟΥΘΗΣΕΙ ΩΣΤΕ ΝΑ ΜΠΕΙ ΣΤΟ ΕΠΟΜΕΝΟ).

Γι'αυτο, έχουμε και το **OR** ώστε είτε να ισχύει το παραπάνω είτε να **βρισκόμαστε στο ακριανό y_cell** του grid (y==*9*) και το συγκεκριμένο y_coordinate της υπολίστας να είναι **ίσο με το maxY.**

Πρέπει να ισχύουν ταυτόχρονα τα x_coordinate φράγματα και τα y_coordinate φράγματα για να θεωρήσουμε ότι ένα σημείο ανήκει στο συγκεκριμένο κελί.

Από την στιγμή που αυτή η συνθήκη ισχύει κάνουμε cellWithElements=1, αυξάνουμε τον αριθμό των σημείων(εστιατορίων) κατά 1 και προσθέτουμε στο τέλος της λίστας *belongsToCell* την υπολίστα για την οποία ίσχυει η συνθήκη.

Αν είναι η πρώτη φορά που βρήκαμε σημείο για το οποίο ισχύει η συνθήκη για το συγκεκριμένο κελί, βάζουμε σε μια λίστα *firstRestaurant* το x το y και τη θέση στην οποία θα γραφτεί στο αρχείο "*grid.grd*" το πρώτο σημείο στο κελί και μαρκάρουμε ότι πλέον δεν είναι η πρώτη φορά στο συγκεκριμένο κελί (firstTimeInCell=0) .

Για να μετράμε τους χαρακτήρες που θα γράφονται στο "*grid.grd*" για τα x_coordinates και y_coordinates σωστά, πρέπει να τα μετατρέπουμε από float σε string αλλά με 6 δεκαδικά ψηφία:

with6decx='{0:.6f}'.format(sublist[1])
with6decy='{0:.6f}'.format(sublist[2])

Οπότε το *placeInGrd* θα είναι κάθε φορά το προηγούμενο αυξημένο κατά το μήκος του αναγνωριστικού αριθμού που έχουμε βάλει(το οποίο είναι int οπότε δεν μας δημιουργεί πρόβλημα στην μετατροπή) συν το μήκος του *with6decx* συν το μήκος του *with6decy,* συν 3 γιατί στο αρχείο θα χρειαστούμε να έχουμε δύο κενά μεταξύ των εγγραφών και ένα '\n' κάθε φορά για αλλαγή γραμμής: *placeInGrd+=len(str(sublist[0]))+len(with6decx)+len(with6decy)+3*

Με το που βγούμε από την for loop για κάθε sublist στην λίστα μας, σημαίνει ότι για το συγκεκριμένο κελί έχουμε ελέγξει κάθε γραμμή του αρχείου "*Beijing_restaurants.txt*" οπότε θα πρέπει να δούμε αν το κελί είχε στοιχεία. Αν ναι, στο τέλος της λίστας του *firstRestaurant* βάζουμε και τον αριθμό των συνολικών σημείων στο κελί και έπειτα το γράφουμε στο αρχείο *"grid.dir"*. Υπάρχουν κελιά που δεν έχουν στοιχεία, το 0 8 και το 0 9.

Πριν προχωρήσουμε για το επόμενο κελί, αδειάζουμε το περιεχόμενο της λίστας *belongsToCell,* κάνουμε τον αριθμό των συνολικών σημείων πάλι 0, το "flag" για το αν είναι η πρώτη φορά στο κελί στο 1, και το "flag" για το αν υπάρχουν σημεία στο κελί στο 0.

PART 2

Χρησιμοποιούμε τώρα το grid που φτιάξαμε στο Part1, για την αποτίμηση ερωτημάτων επιλογής παραθύρου(window selection queries).

if __name__ == '__main__':

```
    dirList=[]
    bounds=dirData()
    dimensions=getWindow(bounds)
    windowEvaluation(dimensions, bounds)
```

Στην main() έχουμε δημιουργήσει μια άδεια λίστα *dirList* στην οποία θα προσθέσουμε-αποθηκεύσουμε έπειτα τα δεδομένα του αρχείου *grid.dir* που έχουμε φτιάξει στο Part1.
Καλούμε λοιπόν, την *dirData()* η οποία θα κρατήσει αυτά τα δεδομένα*:*

def dirData():

```
    firstRow=1

    with open('grid.dir', 'r', encoding='UTF-8') as dfdir:
        for row in dfdir:
            if firstRow==1:
                boundaries=row.split(' ')
                firstRow=0
            else:
                row=row.split(' ')
                intRow=[int(i) for i in row]
                dirList.insert(len(dirList), intRow)

    return boundaries
```

Η μεταβλήτη *firstRow* λειτουργεί ως "flag" ώστε να κρατήσουμε την πρώτη γραμμή του αρχείου *grid.dir* σε μια μεταβλήτη *boundaries* γιατί περιέχει την μικρότερη και την μεγαλύτερη τιμή σε κάθε συντεταγμένη και να μην την προσθέσουμε-αποθηκεύσουμε στην λίστα μας *dirList.*

Κάνουμε γενικά τη συνήθη διαδικασία, “σπάμε” την γραμμή όπου βρούμε κενό.

Για όλες τις άλλες γραμμές του αρχείου εκτός της πρώτης μετατρέπουμε κιόλας σε int τα data της κάθε γραμμής ώστε η λίστα *dirList* να περιέχει λίστες από int. Κάθε φορά τα προσθέτουμε στο τέλος της λίστας μας.

Επιστρέφουμε τα *boundaries* που βρήκαμε πίσω στην main γιατί θα μας χρειαστούν στη συνέχεια (στη main: *bounds=dirData()*)

Στη συνέχεια, στη main() καλείται η συνάρτηση *getWindow* η οποία παίρνει ως όρισμα τα bounds που μόλις βρήκαμε. Αυτή η συνάρτηση θα χρησιμοποιηθεί για να πάρουμε τα command line arguments.

def getWindow(b):

```
print('----------WINDOW SELECTION QUERY----------')
checked=0
while checked==0:
        dimensions=input('Give LowerX, UpperX (%s-%s) and LowerY, UpperY (%s-%s): ' % (b[0], b[1], b[2], b[3]) ).split(' ')
        try:
                dimensions=[float(i) for i in dimensions]
                checked=checkDimensions(dimensions, b)
        except ValueError:
                checked=0

return dimensions
```

Θέλουμε να ορίζει ο χρήστης το window που θέλει με συντεταγμένες της προτίμησής του, ‘ομως οι συντεταγμένες θα πρέπει να είναι μεταξύ των ορίων που έχουμε (διότι διαφορετικά το αποτέλεσμα θα είναι σίγουρα μηδενικό μιας και οι συντεταγμένες δεν θα απευθύνονται στα δεδομένα που διαθέτουμε).
Γι’αυτο χρησιμοποιούμε μια μεταβλήτη *checked* για να γνωρίζουμε πότε τα δεδομένα μας δεν είναι checked (==0 σημαίνει πρέπει να τα τσεκάρουμε πάλι) και μια while-loop που θα επαναλαμβάνει το input όσο δεν έχουμε αποδεκτά dimensions ως input.

Παίρνουμε input από το terminal και δοκιμάζουμε να δούμε αν μετατρέπονται απο string σε float.

Αν ναι, καλείται η *checkDimensions()* με ορίσματα τα dimensions που είναι σίγουρα αριθμοί (και πλέον είναι και float μετά την επιτυχή μετατροπή) και τα b (δηλαδή τα bounds που πήραμε ως όρισμα και στην *getWindow*).
Αν όχι αριθμοί, “πετάει” exception ValueError οπότε η μεταβλητή *checked* παραμένει 0 και ξαναζητάμε από τον χρήστη να μας δώσει input. Το input πρέπει αυστηρά να αποτελείται από 4 ορίσματα για να τρέχει ο κώδικας.

def checkDimensions(d, b):

```
minX=float(b[0])
maxX=float(b[1])
minY=float(b[2])
maxY=float(b[3])

if d[0]<=d[1] and d[2]<=d[3] and d[0]>=minX and d[0]<=maxX and d[1]>=minX and d[1]<=maxX and d[2]>=minY and d[2]<=maxY
                                                                                and  d[3]>=minY and d[3]<=maxY:
        return 1
else:
        return 0
```

Στην περίπτωση που έχουμε περάσει το *try* η συνάρτηση *checkDimensions()* μας εξασφαλίζει αν τα όρια που δίνει είναι ορθά με βάση τη λογική που θέλουμε, δηλαδή οι τιμές που δίνει ο χρήστης να είναι με την σειρά <x_low> <x_high> <y_low> <y_high>.
Συγκεκριμένα, θα επιστρέφει (θα κάνει δηλαδή την μεταβλήτη *checked*) 1 αν όντως η τιμή που ‘εδωσε για το <x_low> είναι μικρότερη ή ίση από την <x_high> και αντίστοιχα η τιμή για την <y_low> από την <y_high> καθώς και <x_low> και αντίστοιχα <x_high> ,να είναι μεγαλύτερη ή ίση από την ολικά μικρότερη τιμή που μπορεί να πάρει (το κάτω όριο του x) και μικρότερη ή ίση από την ολικά μεγαλύτερη τιμή που μπορεί να πάρει (το πάνω όριο του x), ΌΜΟΙΑ για <y_low> και <y_high> με τα πάνω κάτω άκρα του y.
Διαφορετικά, δεν είναι σωστά τα dimensions οπότε επιστρέφει 0.

Όταν το *checked* τελικά είναι διάφορο του 0 τότε μόνο θα “σπάσει η while και θα επιστρέψουμε τα dimensions στην main.

Τέλος, στη main() καλείται η *windowEvaluation()* με ορίσματα τα dimensions που έχει δώσει ο χρήστης, και τα bounds.

def windowEvaluation(d,b):

```
founddl=0
foundur=0
dividedRangeX=(float(b[1])-float(b[0]))/10
dividedRangeY=(float(b[3])-float(b[2]))/10

for x in range(10):
```

```
for y in range(10):

    lowerXCellBound=float(b[0])+(x*dividedRangeX)
    upperXCellBound=float(b[0])+((x+1)*dividedRangeX)
    lowerYCellBound=float(b[2])+(y*dividedRangeY)
    upperYCellBound=float(b[2])+((y+1)*dividedRangeY)

    if d[0]>=lowerXCellBound and d[0]<upperXCellBound and d[2]>=lowerYCellBound and d[2]<upperYCellBound:

        founddl=1
        xdl=x
        ydl=y

    if d[1]<=upperXCellBound and d[1]>lowerXCellBound and d[3]<=upperYCellBound and d[3]>lowerYCellBound:

        foundur=1
        xur=x
        yur=y

    if founddl==1 and foundur==1:
        with open('grid.grd', 'r', encoding='UTF-8') as dfgrd, open('results_part2.txt', 'w', encoding='UTF-8') as rp2:
                for l in dirList:
                    numspots=0

                    if l[0]>xdl and l[0]<xur and l[1]>ydl and l[1]<yur and numspots==0:

                        dfgrd.seek(l[2])
                        for row in dfgrd:
                            numspots+=1
                            row=row.split(' ')
                            if numspots<=l[3]:
                                rp2.write('%s %s %s' % (row[0], row[1], row[2]))
                            else:
                                break

                    elif ((l[0]==xdl or l[0]==xur) and l[1]>=ydl and l[1]<=yur) or ((l[1]==ydl or l[1]==yur) and
                                                                    l[0]>=xdl and l[0]<=xur) and numspots==0:

                        dfgrd.seek(l[2])
                        for row in dfgrd:
                            numspots+=1
                            row=row.split(' ')
                            if float(row[1])>=d[0] and float(row[1])<=d[1] and float(row[2])>=d[2] and
                                                                    float(row[2])<=d[3] and numspots<=l[3]:

                                rp2.write('%s %s %s' % (row[0], row[1], row[2]))
                            elif numspots>l[3]:
                                break
        break

if founddl==1 and foundur==1:
        break
```

Αρχικοποιούμε τις μεταβλήτες “flag” founddl (found down left) και foundur (found upper right) στο 0.
Αυτές θα γίνουν 1 όταν βρούμε το κάτω αριστερά κελί (founddl) που τέμνει το window που έδωσε ο χρήστης και αντίστοιχα το πάνω δεξιά(foundur).

Όπως και στο Part1, βρίσκουμε το εύρος τιμών των συντεταγμένων μας αφαιρώντας από το max (float(b[1])) το min (float(b[0])) και το διαίρουμε με 10 ώστε να βρούμε τα ίσα διαστήματα τιμών για το grid τόσο του x όσο και του y αντίστοιχα:

```
dividedRangeX=(float(b[1])-float(b[0]))/10
dividedRangeY=(float(b[3])-float(b[2]))/10
```

Ξεκινάμε να τσεκάρουμε τώρα τα κουτάκια του grid :

```
for x in range(10):
    for y in range(10):
```

και κάθε φορά υπολογίζουμε το lowerXCellBound upperXCellBound lowerYCellBound και upperYCellBound του κάθε κελιού.

Τσεκάρουμε αν το <x_low> του χρήστη (d[0]), είναι μεγαλύτερο ή ίσο από το κάτω bound x του συγκεκριμένου κελιού και ταυτόχρονα είναι μικρότερο από το άνω bound x του συγκεκριμένου κελιού και συνάμα το <y_low> του χρήστη (d[2]) είναι μεγαλύτερο ή ίσο από το κάτω bound y του συγκεκριμένου κελιού και ταυτόχρονα είναι μικρότερερο από το άνω bound y του συγκεκριμένου κελιού ΤΟΤΕ έχουμε βρεί “την κάτω αριστερή γωνία του window” founddl=1 και κρατάμε στις μεταβλήτες xdl και ydl τα x, y ώστε να ξέρουμε πιο κελί ήταν.

Τσεκάρουμε μετά αν το <x_ high> του χρήστη (d[1]), είναι μικρότερο ή ίσο από το πάνω bound x του συγκεκριμένου κελιού και ταυτόχρονα είναι μεγαλύτερο από το κάτω bound x του συγκεκριμένου κελιού και συνάμα το <y_high> του χρήστη (d[3]) είναι μικρότερο ή ίσο από το πάνω bound y του συγκεκριμένου κελιού και ταυτόχρονα είναι μεγαλύτερο από το κάτω bound y του συγκεκριμένου κελιού ΤΟΤΕ έχουμε βρεί "την πάνω δεξιά γωνία του window" foundur=1 και κρατάμε στις μεταβλήτες xur και yur τα x, y ώστε να ξέρουμε πιο κελί ήταν.

Με το που έχουμε βρει και τα δυο αυτά (*if founddl==1 and foundur==1:)* δεν χρειαζόμαστε περαιτέρω ψάξιμο για κελιά x, y οπότε*:*

Ανοίγουμε το αρχείο *grid.grd* για διάβασμα και το αρχείο '*results_part2.txt*' για γράψιμο .
Για κάθε υπολίστα *l* στην *dirList* αρχικοποιούμε τα σημεία που σκοπεύουμε να διαβάσουμε στο αρχείο *dir.grd* στο 0 (numspots=0) .
Έπειτα, θα προκύψουν δυο περιπτώσεις στις οποίες καλούμαστε να γράψουμε τα αντίστοιχα αποδεκτά σημεία από το αρχείο *grid.grd* με βάση τα δωθέντα dimensions του χρήστη :

-Τα σημεία ('ολα) των κελιών που καλύπτονται πλήρως από το window :

if l[0]>xdl and l[0]<xur and l[1]>ydl and l[1]<yur and numspots==0:

Πρέπει να είναι η συντεταγμένη του x αυστηρά μεγαλύτερη από την τιμή του xdl που δηλώνει το κελί που τέμνει το window κάτω αριστερά, αυστηρά μικρότερη από την τιμή του xur που δηλώνει το κελί που τέμνει το window πάνω δεξιά, η συντεταγμένη του y αυστηρά μεγαλύτερη από την τιμή του ydl που δηλώνει το κελί που τέμνει το window κάτω αριστερά, αυστηρά μικρότερη από την τιμή του yur που δηλώνει το κελί που τέμνει το window πάνω δεξιά και να έχουμε νέα περίπτωση για x, y (τα numspots μηδενίζονται όταν τελείωνουμε από ένα κελί με συγκεκριμένα x, y).

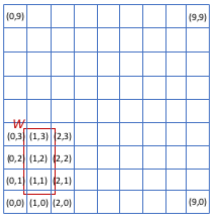
ΤΟΤΕ και μόνο τότε βρισκόμαστε σίγουρα σε εσωτερικά κελιά οπότε μετακινούμαστε όσους χαρακτήρες χρειάζεται στο αρχείο *grid.grd* (dfgrd.seek(l[2])) και για κάθε γραμμή στο αρχείο αυξάνουμε τα numspots κατά 1, "σπάμε" τα data της γραμμής και αν το numspots δεν έχει υπερβεί τα καταγεγραμμένα στο αρχείο *grid.grd* σημεία που ανήκουν στο κελί, τα γράφουμε χωρίς να χρειάζεται να εξετάσουμε τις συντεταγμένες τους. Κάνουμε break όταν ξεπεράσουμε τα numspots του συγκεκριμένου κελιού και πάμε να κοιτάξουμε για νέα x, y.

-Τα σημεία (μόνο όσα είναι αποδεκτά) των κελιών που καλύπτονται μερικώς από το window :

elif ((l[0]==xdl or l[0]==xur) ***and*** *l[1]>=ydl and l[1]<=yur)* ***or*** *((l[1]==ydl or l[1]==yur)* ***and*** *l[0]>=xdl and l[0]<=xur) and numspots==0:*

Εδώ, θέλουμε να πιάσουμε όλα τα κελιά τα οποία "τέμνει" ουσιαστικά απο κάθε πλευρά το window του χρήστη.

Γι'αυτό, αν η συντεταγμένη του x ισούται με το xdl ή το xur και η συντεταγμένη του y κυμαίνεται από ydl ώς και yur (πχ στο παράδειγμα της εκφώνησης αυτά είναι τα : (0,0) , (0,1) , (0,2), (0,3) , (2,0), (2,1), (2,2), (2,3))
Ή αν η συντεταγμένη του y ισούται με το ydl ή το yur και η συντεταγμένη του x κυμαίνεται από xdl ως και xur (πχ στο παράδειγμα της εκφώνησης αυτά είναι τα : (0,0) , (1,0) , (2,0) , (0,3), (1,3), (2,3))
και να έχουμε νέα περίπτωση για x, y (numspots==0)



ΤΟΤΕ και μόνο τότε βρισκόμαστε σίγουρα σε κελιά που τα "τέμνουν" οι γραμμές του window και θα έτσι μετακινούμαστε όσους χαρακτήρες χρειάζεται στο αρχείο *grid.grd* (dfgrd.seek(l[2])) και για κάθε γραμμή στο αρχείο αυξάνουμε τα numspots κατά 1, "σπάμε" τα data της γραμμής και αν το x_coordinate της γραμμής του αρχείου είναι μεγαλύτερο ή ίσο από το <x_low> του χρήστη και το x_coordinate της γραμμής του αρχείου είναι μικρότερο ή ίσο από το <x_high> του χρήστη και το y_coordinate της γραμμής του αρχείου είναι μεγαλύτερο ή ίσο από το <y_low> του χρήστη και το y_coordinate της γραμμής του αρχείου είναι μικρότερο ή ίσο από το <y_high> του χρήστη και τα numspots δεν έχουν υπερβεί τον αριθμό για το συγκεκριμένο κελί γράφουμε τις συντεταγμένες τους.
Αλλίως αν έχουμε υπερβεί το numspots του συγκεκριμένου κελιού κάνουμε break και πάμε να κοιτάξουμε για νέα x, y.

Όταν κλείσει η *with open* σημαίνει ότι έχουμε τελειώσει και κάνουμε break για να βγούμε από την for πρώτα του y και έπειτα με την συνθήκη *if founddl==1 and foundur==1:* κάνουμε break και για το x.

PART 3

Σε αυτό το μέρος, θα βρίσκουμε σταδιακά τους πλησιέστερους k γείτονες- τα πλησιέστερα εστιατόρια- ενός σημείου q, όπου k,q θα δίνονται από τον χρήστη στην αρχή του προγράμματος.
Κάνουμε ***import math*** για τις ρίζες στις ευκλείδιες αποστάσεις που θα χρειαστούμε παρακάτω.

if __name__ == '__main__':

```
priorityQueue=[]
nCList=[]
dirList=[]
allVisitedCells=[]

bounds=dirData()
arguments=getkq(bounds)

k=int(arguments[0])
q=[float(arguments[1]), float(arguments[2])]
cell=findqCell(q, bounds)
print('q cell:', cell)

with open('results_part3.txt', 'w', encoding='UTF-8') as rp3:
```

```
        counter=0
        for nn in knnGenerator(q, k, bounds, cell):
                if counter<k:
                        rp3.write('%s %s\n' % ('{0:.6f}'.format(nn[0]), '{0:.6f}'.format(nn[1])))
                        counter+=1
                else:
                        for i in allVisitedCells:
                                rp3.write(str(i))
                        break
```

Στην main, αρχικοποιούμε να είναι κενή:

- την λίστα *priorityQueue[]* η οποία θα θα περιέχει κελιά και σημεία με προτεραιότητα βασισμένη στην απόσταση τους από το q.
- την λίστα *nClist[]* η οποία θα περιέχει κάθε κελί που επισκεπτόμαστε και όσα γειτονικά του υπάρχουν (‘ολα από μία μόνο φορά όμως).
- την λίστα *dirList[]* η οποία θα έχει τα περιεχόμενα του αρχείου “grid.dir”.
- την λίστα *allVisitedCells[]* η οποία θα περιέχει όσα κελιά έχουν βγεί από την ουρά (τα κελιά δηλαδή που διαβάστηκαν από το πρόγραμμα για τα σημεία που έχουν “μέσα”.

Πρώτα, καλούμε την *dirData()* η οποία βάζει τα περιεχόμενα του αρχείου “grid.dir” στην *dirList[]* και μας επιστρέφει τα bounds που βρίσκονται στην πρώτη γραμμή του αρχείου (όπως ακριβώς και στο part2) .

def dirData():

```
        firstRow=1

        with open('grid.dir', 'r', encoding='UTF-8') as dfdir:
                for row in dfdir:
                        if firstRow==1:
                                boundaries=row.split(' ')
                                firstRow=0
                        else:
                                row=row.split(' ')
                                intRow=[int(i) for i in row]
                                dirList.insert(len(dirList), intRow)
        return boundaries
```

Έπειτα, καλούμε την συνάρτηση *getkq()* με όρισμα τα bounds που βρήκαμε από την *dirData()* η οποία θα ζητά και θα καλεί την συνάρτηση *checkArgs()* για να ελέγξει αν είναι επιτρεπτά τα inputs και θα μας επιστρέψει το k και q που έδωσε ο χρήστης.

def getkq(b):

```
        print('----------INCREMENTAL NEAREST NEIGHBOR SELECTION----------')
        checked=0
        while checked==0:
                args=input('Give k (1-51969), x_coordinate (%s-%s) and y_coordinate (%s-%s): ' % (b[0], b[1], b[2], b[3]) ).split(' ')
                try:
                        checked=checkArgs(args,b)
                except ValueError:
                        checked=0
        return args
```

Για την *checkArgs()* θα πρέπει να πάρουμε ως όρισμα το input του χρήστη και τα bounds, και στην συνέχεια να ελέγξουμε:
- το k να είναι απο 1 ως 51969 (δεν έχει νόημα να είναι 0 και συνολικά τα σημεία ειναι γνωστό από την εκφώνηση ότι είναι 51970 άρα οι γείτονες του ενός από αυτών μπορούν να φτάσουν ως ένα πλην)
- και τα x,y coordinates που έδωσε ο χρήστης αντίστοιχα να είναι για το καθένα ανάμεσα στα επιτρεπτά του bounds.

Το input πρέπει αυστηρά να αποτελείται από 3 ορίσματα για να τρέχει ο κώδικας.

def checkArgs(inpt,b):

```
        minX=float(b[0])
        maxX=float(b[1])
        minY=float(b[2])
        maxY=float(b[3])

        if int(inpt[0])>=1 and int(inpt[0])<51970 and float(inpt[1])>=minX and float(inpt[1])<=maxX and float(inpt[2])>=minY and
                                                                                                        float(inpt[2])<=maxY :
                return 1
        else:
                return 0
```

Εφόσον έχουμε στην *main()* βρεί k. q μας μένει να βρούμε σε ποιο κελί βρίσκεται το q το οποίο το αναλαμβάνει η συνάρτηση *findqCell()* με ορίσματα τα q, bounds .

def findqCell(q, b):

```
dividedRangeX=(float(b[1])-float(b[0]))/10
dividedRangeY=(float(b[3])-float(b[2]))/10
for x in range(10):
        for y in range(10):

            lowerXCellBound=float(b[0])+(x*dividedRangeX)
            upperXCellBound=float(b[0])+((x+1)*dividedRangeX)
            lowerYCellBound=float(b[2])+(y*dividedRangeY)
            upperYCellBound=float(b[2])+((y+1)*dividedRangeY)

            if q[0]>=lowerXCellBound and q[0]<=upperXCellBound and q[1]>=lowerYCellBound and
                                                                        q[1]<=upperYCellBound:

                return [x,y]
```

Βρίσκει το εύρος για τα κελιά και για τον άξονα y και για τον x και μετά, ψάχνει να βρεί σε ποιο κελί τα x,y του q περιέχονται και επιστρέφει το κελί.

Στην main() έχουμε συγκεντρώσει πλέον τις βασικές πληροφορίες και θα προχωρήσουμε με ανοίγμα του αρχείου "results_part3.txt" για γράψιμο .
Καλούμε σε μια for loop την generator συνάρτηση μας *knnGenerator()* με ορίσματα το q, k, bounds και cell.
Εσωτερικά έχουμε μία συνθήκη για να σταματήσουμε όταν φτάσουμε σε k φορές. Όσο το counter είναι μικρότερο του k γράφουμε στο αρχείο ότι μας έχει επιστρέψει ο generator αλλιώς γράφουμε όλα τα στοιχεία της λίστας *allVisitedCells[]* και τερματίζουμε με break.

def knnGenerator(q, k, b, cell):

```
ordCells=[[cell[0], cell[1], 0]]
ordSpots=[]
firstCell=[]
lastSpot=[]
firstSpotNeighborCells=[]

haveCell=1
firstTime=1
canYield=0
noCells=0
noFC=0

countSpots=0
wheresLastSpot=-1

priorityQueue.insert(len(priorityQueue), [cell[0], cell[1], 0])
```

...
Αρχικοποιούμε τις λίστες που θα χρησιμοποιήσουμε στην συνέχεια.
- Η λίστα *ordCells[]* θα κρατάει συνολικά όλα τα κελιά (αρχικά το κελί στο οποίο βρίσκεται το q και έπειτα τα γειτονικά κελιά οποιουδήποτε κελιού ανοίγουμε για εισαγωγή στοιχείων) μαζί με την κοντινότερη απόσταση του κελιού από το σημείο q.
- Η λίστα *ordSpots[]* θα έχει κάθε φορά τα νέα spots του κελιού που ανοίγουμε.
- Η λίστα *firstCell[]* θα έχει το πρώτο κελί και την κοντινότερη απόσταση του απο το σημείο q, που συναντάμε μέσα στην priorityQueue όταν πλέον θα έχουμε εξαντλήσει όλα όσα είναι να εισαχθούν.
- Η λίστα l*astSpot[]* θα έχει τις συντεταγμένες του τελευταίου spot και την απόσταση του από το q που συναντάμε στην priorityQueue πριν το πρώτο κελί.
- Η λίστα *firstSpotNeighborCells[]* κάθε φορά προσθέτει τα γειτονικά κελιά και την απόσταση από το q του κοντινότερου όμως σημείου spot του κελιού.

Flags:
- Το haveCell όταν 1 σημαίνει ότι έχουμε βρει κελί αλλιώς 0 για σημείο.
- Το firstTime όταν 1 σημαίνει ότι είναι η εισαγωγή του κελιού που βρίσκεται το q.
- Το canYield όταν 0 σημαίνει ότι δεν μπορούμε να κάνουμε yield γιατί δεν είναι η σειρά του σημείου ακόμα να επιστραφεί αλλίως 1 όταν μπορεί.
- Το noCells όταν 0 σημαίνει ότι έχουμε ακόμα κελιά στην *firstSpotNeighborCells[]* αλλιώς 1.
-Το noFC όταν 1 σημαίνει ότι δεν βρήκαμε κανένα κελί μέσα στην priorityQueue όταν την σαρώσαμε και είμαστε στην περίπτωση που έχουμε βάλει όλα τα υπάρχοντα κελιά του grid ήδη.

Μεταβλητές:
- Η countSpots μετράει πόσα σημεία έχουμε κάνει insert στην priorityQueue.
-Η wheresLastSpot μετράει σε ποια θέση στην priorityQueue είναι το lastSpot. Την αρχικοποιούμε στο -1 ώστε αν μπεί ως πρώτο στην priorityQueue το lastSpot να βρισκόμαστε στη θέση 0 (το αυξάνουμε εκεί).

Ξεκινάμε κάνοντας insert (στο τέλος της priorityQueue) το κελί που βρίσκεται το q και την απόσταση απο το σημείο q, που είναι 0.

Έχουμε μια loop η οποία θα τρέχει μέχρι να τερματίσει το k από την main.
...
```
while True:
```

```
        if haveCell==1:

                inserting(priorityQueue[0][0], priorityQueue[0][1], firstTime, countSpots, ordCells, ordSpots, firstSpotNeighborCells,
                                                                                                                allVisitedCells)

                if firstTime==1:
                        firstTime=0

                wheresLastSpot=-1
                for element in priorityQueue:
                        if element[0]>=10:
                                lastSpot=element
                                wheresLastSpot+=1
                                haveCell=0
                        else:
                                break
```

...

Στην περίπτωση που έχουμε ως πρώτο στοιχείο στην priorityQueue μας, κελί, καλούμε την συνάρτηση *insert()* η οποία θα αναλάβει να κάνει την απαραίτητη διαδικασία για να αλλάξει κατάλληλα σε κάθε περίπτωση αυτή η δυνανική ουρά προτεραιότητας.

Έπειτα κάνουμε το firstTime 0 αν είμαστε στην πρώτη φορά.

Ξεκινάμε να ξάχνουμε το lastSpot της πλέον τροποποιημένης priorityQueue. Αν έστω μπεί μία φορά στην *if element[0]>=10:* κάνουμε το haveCell 0 γιατί σίγουρα έχουμε βρει ένα στοιχείο όπως σαρώνουμε την priorityQueue, αν βρούμε κελί κάνουμε break (οπότε το haveCell παραμένει 1 αν πρώτο στην priorityQueue είναι κελί).

....

```
elif not firstSpotNeighborCells:

        noCells=1

        for element in priorityQueue:
                if element[0]>=0 and element[0]<=9:
                        firstCell=element
                        noFC=0
                        break
                else:
                        noFC=1

        if noFC==1:

                for element in priorityQueue:
                        nearestNeighbor=element
                        priorityQueue.pop(0)
                        yield nearestNeighbor
                        break

        else:
                leftCellSpots=orderedNSpots(q, [firstCell[0], firstCell[1]])

                if not leftCellSpots:
                        allVisitedCells.insert(len(allVisitedCells), [firstCell[0], firstCell[1]])
                        priorityQueue.remove(firstCell)

                else:

                        firstSpotDist=leftCellSpots[0][2]
                        for element in priorityQueue:
                                eldist=element[2]

                                if eldist<=firstSpotDist and element[0]>=10:

                                        nearestNeighbor=element
                                        priorityQueue.pop(0)
                                        yield nearestNeighbor
                                        break
                                else:
                                        allVisitedCells.insert(len(allVisitedCells), [firstCell[0], firstCell[1]])
                                        priorityQueue.remove(firstCell)

                                        for spot in leftCellSpots:
                                                place=0
                                                spotdist=spot[2]
                                                for element in priorityQueue:
                                                        eldist=element[2]
                                                        if spotdist<=eldist and spot not in priorityQueue:
                                                                priorityQueue.insert(place, spot)

                                                                break
                                                        else:
                                                                place+=1

                                        for spot in leftCellSpots:
                                                if spot not in priorityQueue:
                                                        priorityQueue.insert(len(priorityQueue), spot)
                                        break
```

.....

Αυτή είναι μια περίπτωση που ‘φθάνει’ μόνο όταν εξαντλήσουμε όλα τα γειτονικά κελιά (εισάγουμε όλο το grid, και η

firstSpotNeighborCells[] είναι δηλαδή κενή).
Κάνουμε το noCells 1 και σαρώνουμε την priorityQueue μας μέχρι να βρούμε το πρώτο κελί.

Άμα δεν υπάρχει, σημαίνει ότι στην ουρά έχουν μείνει μόνο σημεία οπότε κάνουμε yield nearestNeighbors μέχρι να μας σταματήσει η main, διαφορετικά, φορτώνουμε 'προσωρινά' στην *leftCellSpots[]* τα σημεία του *firstCell* και αν δεν υπάρχουν απλά σημειώνουμε ότι το επισκεφθήκαμε και το αφαιρούμε από την priorityQueue αλλιώς ως *firstSpotDist* κρατάμε την απόσταση του πρώτου σημείου του συγκεκριμένου κελιού από την ουρά και σαρώνουμε την priorityQueue.

Όσο η απόσταση των σημείων της priorityQueue είναι μικρότερη ή ίση απο την απόσταση του πρώτου σημείου στο πρώτο επόμενο κελί και δεν συναντάμε κάποιο κελί ο nearestNeighbor είναι το πρώτο στοιχείο στην ουρά, το αφαιρούμε, το κάνουμε yield και κάνουμε break ώστε να ξαναελέγξουμε.

Διαφορετικά, πρέπει να εισάγουμε το κελί, οπότε το σημειώνουμε ως *allVisitedCells[]* το αφαιρούμε από την priorityQueue και στην συνέχεια τοποθετούμε τα σημεία του *firstCell* κατάλληλα.

Για κάθε spot στην *leftCellSpots[]* και για κάθε στοίχειο της priorityQueue (εφόσον το στοιχείο δεν είναι ήδη στην priorityQueue) **εάν η απόσταση του spot από το q είναι μικρότερη ή ίση απ'ότι το στοιχείο της priorityQueue, πρόσθέτουμε το spot πριν από το συγκεκριμένο στοιχείο στην priorityQueue,** αλλιώς αύξάνουμε το *place* κατά 1.
Για όσα δεν μπήκαν λόγω μεγαλύτερης απόστασης από τα ήδη περασμένα στην priorityQueue, έχουμε μια ακόμα σάρωση, η οποία για κάθε spot της *leftCellSpots[]* που δεν έχει εισαχθεί, το βάζει στο τέλος της priorityQueue (αφού είναι ταξηνομημένα τα spots, αν κάποιο ήταν να εισχώρήσει κάπου ανάμεσα των στοιχείων θα είχε ήδη μπει).

Κάνουμε break ώστε να ελέγξει ξανά.

```
...
        elif haveCell==0 and lastSpot[2]>firstSpotNeighborCells[0][2] and k>=wheresLastSpot+1 and noCells==0:

                inserting(firstSpotNeighborCells[0][0], firstSpotNeighborCells[0][1], firstTime, countSpots, ordCells, ordSpots,
                                                                                        firstSpotNeighborCells, allVisitedCells)

                wheresLastSpot=-1
                for element in priorityQueue:
                        if element[0]>=10:
                                lastSpot=element
                                wheresLastSpot+=1
                        else:
                                if priorityQueue[0][0]>=0 and priorityQueue[0][0]<=9:
                                        haveCell=1
                                break

...
```

Είναι μία περίπτωση στην οποία έχουμε σίγουρα σημείο-α στην κορυφή της ουράς μας αλλά **το τελευταίο σημείο πριν το πρώτο κελί έχει απόσταση απο το σημείο q μεγαλύτερη από ότι το πρώτο σημείο του πρώτου κελιού που ακολουθεί, το k να είναι μεγαλύτερο ή ίσο με τη θέση που βρήκαμε το τελευταίο σημείο πριν το πρώτο κελί** και φυσικά θέλουμε το flag του noCells να είναι 0.

Τότε, καλούμε την συνάρτηση *inserting()* με το κελί που έχουμε με τα μικρότερα σημεία μέσα και μετά βρίσκουμε το επόμενο lastSpot της τροποποιημένης priorityQueue.

```
...
        elif haveCell==0 and ((lastSpot[2]<=firstSpotNeighborCells[0][2] and k>=wheresLastSpot+1) or (lastSpot[2]>firstSpotNeighborCells[0][2]
                                                                                and k<wheresLastSpot+1)) and noCells==0:

                for element in priorityQueue:

                        canYield=0
                        checked=0
                        if element[0]>=10 and checked==0:
                                for ncell in firstSpotNeighborCells:
                                        if lastSpot[2]<=ncell[2]:

                                                canYield=1
                                        else:

                                                canYield=0
                                                checked=1
                                                inserting(ncell[0], ncell[1], firstTime, countSpots, ordCells, ordSpots,
                                                                                        firstSpotNeighborCells, allVisitedCells)

                                                wheresLastSpot=-1
                                                for element in priorityQueue:
                                                        if element[0]>=10:
                                                                lastSpot=element
                                                                wheresLastSpot+=1
                                                        else:
                                                                if priorityQueue[0][0]>=0 and priorityQueue[0][0]<=9:
                                                                        haveCell=1
                                                                break
                                                break
                                if canYield==1:
                                        checked=1
                                        nearestNeighbor=element
                                        priorityQueue.remove(element)
                                        yield nearestNeighbor
```

```
                                        break
                else:
                        if element[0]>=0 and element[0]<=9:
                                haveCell=1
                                break
```

...
Άλλη μια περίπτωση που σίγουρα έχουμε σημείο-α στην κορυφή της ουράς μας αλλά τώρα είτε
-**το τελευταίο σημείο πριν το πρώτο κελί έχει απόσταση από το σημείο q μικρότερη ή ίση απ' ότι το πρώτο σημείο του πρώτου κελιού που ακολουθεί και το k είναι μεγαλύτερο ή ίσο με την θέση που βρήκαμε το τελευταίο σημείο πριν το πρώτο κελί** - είτε - **το πρώτο σημείο πριν το πρώτο κελί είναι μεγαλύτερο απ' ότι το πρώτο σημείο του πρώτου κελιού που ακολουθεί και το k είναι μικρότερο όμως απ' τη θέση του τελευταίου σημείου στην priorityQueue** (που σημαίνει ότι αναγκαστικά θα πρέπει να ανοίξουμε και άλλο-α κελι-ά γιατί δεν μας φτάνουν τα σημεία) και φυσικά το flag του noCells να είναι 0.

Εδώ θα κάνουμε "μερικώς yield", θα μας χρειαστούν και δύο flags, canYield που σημαίνει ότι μπορεί όντως να είναι ο επόμενος nearestNeighbor όταν 1, αλλιώς 0 και checked που όταν είναι 1 σημαίνει ότι για το συγκεκριμένο στοιχείο στην priorityQueue έχω ελέγξει αν μπορεί να είναι nearestNeighbor αλλίως 0.

Πιο συγκεκριμένα, θα σαρώνουμε την priorityQueue και αν το στοίχειο είναι σημείο που δεν το έχουμε ελέγξει (*element[0]>=10)* τότε για κάθε ένα στοιχείο της firstSpotNeighborCells[] θα πρέπει η απόσταση να είναι μικρότερη ή ίση απ'ότο του πρώτου σημείου των επόμενων κελιών που έχουν ήδη προστεθεί κάπου μέσα στην priorityQueue.

Αν έστω και ένα από αυτά τα κελιά έχει πρώτο σημείο μικρότερο η canYield γίνεται 0 και τσεκάρετε ότι ελέγθηκε, και συνεπώς καλούμε την *inserting()* με όρισμα το συγκεκριμένο κελί που βρήκαμε ότι έχει μικρότερο πρώτο στοιχείο. Στη συνέχεια, βρίσκουμε το επόμενο lastSpot (αν το priorityQueue[0][0]>=0 and priorityQueue[0][1]<=9 σημαίνει ότι έχουμε πρώτο στην priorityQueue κελί) και κάνουμε τα break που χρειάζονται για να ξαναελέγξουμε.

Αν όλα τα κελιά της nearestNeighborCells[] έχουν μικρότερα ή ίσα πρώτα στοιχεία το canYield παραμένει 1 και πάμε στην συνθήκη όπου σημειώνουμε ότι ελεγθηκε το σημείο αυτό και το γυρνάμε ως nearestNeighbor το βγάζουμε από την priorityQueue και κάνουμε break για να ελέγξουμε για το επόμενο (έχουμε και εδώ έλεγχο για το αν το πλέον νέο στοιχείο της priorityQueue είναι κελί).

```
...
        elif lastSpot[2]<=firstSpotNeighborCells[0][2] and k<wheresLastSpot+1 and noCells==0:

                for element in priorityQueue:
                        if element[0]>=10:
                                nearestNeighbor=element
                                yield nearestNeighbor
```

Τέλος, 'εχουμε την περίπτωση να έχουμε **το τελευταίο σημείο πριν το πρώτο κελί να έχει απόσταση από το σημείο q μικρότερη ή ίση απ' ότι το πρώτο σημείο του πρώτου κελιού που ακολουθεί και το k είναι μικρότερο από την θέση που βρήκαμε το τελευταίο σημείο πριν το πρώτο κελί** και noCells 0.

Εδώ, έχουμε το πλεονέκτημα να κάνουμε yield όλα τα σημεία πρίν το πρώτο κελί χωρίς κάποιον άλλον έλεγχο. Είναι περίπτωση που ξέρουμε ότι θα οδηγήσει και σε τερματισμό.

Είδαμε ότι χρησιμποιήσαμε την συνάρτηση *inserting()* για να εισάγουμε κάθε φορά το κελί που χρειαζόταν στην ουρά μας. Τα ορίσματα της είναι οι συντεταγμένες x, y, το "flag" firstTime, το countSpots, οι λίστες ordCells, ordSpots, firstSpotNeighborCells και allVisitedCells.

***def inserting** (xcoord, ycoord, firstTime, countSpots, ordCells, ordSpots, firstSpotNeighborCells, allVisitedCells):*

```
        if firstTime==0:
                for f in firstSpotNeighborCells:
                        if f[0]==xcoord and f[1]==ycoord:
                                firstSpotNeighborCells.remove(f)
                                break

        allVisitedCells.insert(len(allVisitedCells), [xcoord, ycoord])
        newNCells=mindist(q, bounds,ordCells, [xcoord, ycoord])
....
```

Αν δεν είναι η πρώτη φορά, πρέπει να αφαιρούμε το κελί και από την *firstSpotNeighborCells[]* , σε κάθε περίπτωση εισάγουμε το κελί που πήραμε ως όρισμα στην λίστα με αυτά που έχουμε επισκεπτεί (*allVisitedCells[])* και καλούμε την συνάρτηση *mindist()* ώστε να μας επιστρέψει στην *newNCells[]* με αύξουσα σειρά όλα τα γειτονικά κελιά του κελιού που επισκεφθήκαμε.

```
....
        for i in newNCells:
                if i not in ordCells:
                        #**************************************
                        newiSpots=orderedNSpots(q, [i[0], i[1]])
                        if  newiSpots:
```

```
                                    firstSpotDist=newiSpots[0][2]
                                    firstSpotNeighborCells=putOrder(firstSpotNeighborCells, i[0], i[1], firstSpotDist)
                            #*****************************************
                            ordCells.insert(len(ordCells), I)

        ordSpots=orderedNSpots(q, [xcoord, ycoord])
        for cells in ordCells:
                if cells[0]==xcoord and cells[1]==ycoord:
                        xdl=cells[2]
                        break

        priorityQueue.remove([xcoord, ycoord, xdl])
....
```

Για κάθε στοιχείο στην *newNCells[]* αν δεν υπάρχει στο *ordCells[]* ήδη πρώτον φορτώνουμε “προσωρινά” στην *newiSpots[]* τα σημεία του μέσω της κλήσης στην συνάρτηση *orderedNSpots()* και αν έχει (γιατί υπάρχουν και κελιά που δεν έχουν όπως είδαμε (0,8) (0,9)) κρατάμε στο *firstSpotDist* την απόσταση που έχει το πρώτο σημείο του συγκεκριμένου γειτονικού κελιού και καλούμε την συνάρτηση *putOrder()* για να το βάλει στη σωστή θέση μέσα στην *firstSpotNeighborCells[].*
Για καθένα επίσης, το προσθέτουμε και στο τέλος της ordCells.

Στη συνέχεια φορτώνουμε στην *ordSpots[]* τα νέα σημεία του κελιού που πήραμε ως όρισμα.

Βρίσκουμε από την *ordCells[]* ποιο είναι το κελί αυτό (γιατί αν έχει εισαχθεί από αυτά που εισάγουμε με την χρήση του *firstSpotNeighborCells []* δεν έχουμε την απόσταση του κελιού απο το σημείο q αλλά την απόσταση του πρώτου σημείου του κελιού από το σημειο q) και το αφαιρούμε από την priorityQueue.

```
....
        for spot in ordSpots:
                place=0
                spotdist=spot[2]
                for element in priorityQueue:
                        eldist=element[2]
                        if spotdist<=eldist and spot not in priorityQueue:
                                priorityQueue.insert(place, spot)
                                countSpots+=1
                                break
                        else:
                                place+=1
        for spot in ordSpots:
                if spot not in priorityQueue:
                        priorityQueue.insert(len(priorityQueue), spot)
                        countSpots+=1

        for c in newNCells:
                place=0
                celldist=c[2]
                for element in priorityQueue:
                        eldist=element[2]
                        if celldist<=eldist:
                                priorityQueue.insert(place, c)
                                break
                        else:
                                place+=1
        for c in newNCells:
                if c not in priorityQueue:
                        priorityQueue.insert(len(priorityQueue), c)
```

Τέλος μας απομένει το κομμάτι της εισαγωγής των στοιχείων αυτών στην priorityQueue.
Θα μας βοηθήσει η χρήση μιας μεταβλητής *place* ώστε να ξέρουμε που να τα τοποθετούμε στην priorityQueue.

Για κάθε spot στην *ordSpots[]* και για κάθε στοίχειο της priorityQueue (εφόσον το στοιχείο δεν είναι ήδη στην priorityQueue) **εάν η απόσταση του spot από το q είναι μικρότερη ή ίση απ’ότι το στοιχείο της priorityQueue, πρόσθέτουμε το spot πριν από το συγκεκριμένο στοιχείο στην priorityQueue,** αλλιώς αύξάνουμε το *place* κατά 1.
Για όσα δεν μπήκαν λόγω μεγαλύτερης απόστασης από τα ήδη περασμένα στην priorityQueue, έχουμε μια ακόμα σάρωση, η οποία για κάθε spot της ordSpots που δεν έχει εισαχθεί, το βάζει στο τέλος της priorityQueue (αφού είναι ταξηνομημένα τα spots, αν κάποιο ήταν να εισχώρήσει κάπου ανάμεσα των στοιχείων θα είχε ήδη μπει).
Για κάθε εισαγωγή στο priorityQueue αυξάνουμε κατά 1 το *countSpots* ώστε να ξέρουμε πόσα σημεία έχουμε φτάσει ήδη μέσα στην priorityQueue.

Όμοια, ακριβώς στην συνέχεια, τσεκάρουμε και για τα νέα γειτονικά κελιά που πρέπει να βάλουμε στην priorityQueue .

def nearestCells(c):

```
        x=c[0]
        y=c[1]

        if [x,y] not in nCList:
                nCList.insert(len(nCList), [x,y])
```

Η συνάρτηση *nearestCells()* παίρνει ως όρισμα ένα κελί κοιτάει αν το κελί βρίσκεται στην *nCList[]* αν όχι το βάζει στο τέλος της και ελέγχει έπειτα σε πιο σημείο στο grid βρίσκεται.

Στα ακριανά γωνιακά σημεία, μπορούν να μπούν έως 3 γείτονες αν δεν βρίσκονται ήδη στην *nCList[].*

```
if x==0 and y==0:
        if [x, y+1] not in nCList:
                nCList.insert(len(nCList), [x, y+1])
        if [x+1, y+1] not in nCList:
                nCList.insert(len(nCList), [x+1, y+1])
        if [x+1, y] not in nCList:
                nCList.insert(len(nCList), [x+1, y])
elif x==0 and y==9:
        if [x+1, y] not in nCList:
                nCList.insert(len(nCList), [x+1, y])
        if [x+1, y-1] not in nCList:
                nCList.insert(len(nCList), [x+1, y-1])
        if [x, y-1] not in nCList:
                nCList.insert(len(nCList), [x, y-1])
elif x==9 and y==0:
        if [x-1, y] not in nCList:
                nCList.insert(len(nCList), [x-1, y])
        if [x-1, y+1] not in nCList:
                nCList.insert(len(nCList), [x-1, y+1])
        if [x, y+1] not in nCList:
                nCList.insert(len(nCList), [x, y+1])
elif x==9 and y==9:
        if [x-1, y] not in nCList:
                nCList.insert(len(nCList), [x-1, y])
        if [x-1, y-1] not in nCList:
                nCList.insert(len(nCList), [x-1, y-1])
        if [x, y-1] not in nCList:
                nCList.insert(len(nCList), [x, y-1])
```

x==0 and y==9:	[x+1, y]							[x-1, y]	x==9 and y==9:
[x, y-1]	[x+1, y-1]							[x-1, y-1]	[x, y-1]
[x, y+1]	[x+1, y+1]							[x-1, y+1]	[x, y+1]
x==0 and y==0:	[x+1, y]							[x-1, y]	x==9 and y==0:

Μετά έχουμε περιπτώσεις που βρίσκονται ακριανά αλλά δεν βρίσκονται στις κορυφές:

```
elif x==0 and y>=1 and y<=8:
        if [x, y+1] not in nCList:
                nCList.insert(len(nCList), [x, y+1])
        if [x+1, y+1] not in nCList:
                nCList.insert(len(nCList), [x+1, y+1])
        if [x+1, y] not in nCList:
                nCList.insert(len(nCList), [x+1, y])
        if [x+1, y-1] not in nCList:
                nCList.insert(len(nCList), [x+1, y-1])
        if [x, y-1] not in nCList:
                nCList.insert(len(nCList), [x, y-1])
```

[x, y+1]	[x+1, y+1]								
qcell	[x+1, y]								

[x, y-1]	[x+1, y-1]								

elif x>=1 and x<=8 and y==9:

```
if [x-1, y] not in nCList:
        nCList.insert(len(nCList), [x-1, y])
if [x-1, y-1] not in nCList:
        nCList.insert(len(nCList), [x-1, y-1])
if [x, y-1] not in nCList:
        nCList.insert(len(nCList), [x, y-1])
if [x+1, y-1] not in nCList:
        nCList.insert(len(nCList), [x+1, y-1])
if [x+1, y] not in nCList:
        nCList.insert(len(nCList), [x+1, y])
```

	[x-1, y]	qcell	[x+1, y]						
	[x-1, y-1]	[x, y-1]	[x+1, y-1]						

elif x==9 and y>=1 and y<=8:

```
if [x, y+1] not in nCList:
        nCList.insert(len(nCList), [x, y+1])
if [x-1, y+1] not in nCList:
        nCList.insert(len(nCList), [x-1, y+1])
if [x-1, y] not in nCList:
        nCList.insert(len(nCList), [x-1, y])
if [x-1, y-1] not in nCList:
        nCList.insert(len(nCList), [x-1, y-1])
if [x, y-1] not in nCList:
        nCList.insert(len(nCList), [x, y-1])
```

								[x-1, y+1]	[x, y+1]
								[x-1, y]	qcell
								[x-1, y-1]	[x, y-1]

elif x>=1 and x<=8 and y==0:

```
if [x-1, y] not in nCList:
        nCList.insert(len(nCList), [x-1, y])
if [x-1, y+1] not in nCList:
        nCList.insert(len(nCList), [x-1, y+1])
if [x, y+1] not in nCList:
        nCList.insert(len(nCList), [x, y+1])
if [x+1, y+1] not in nCList:
        nCList.insert(len(nCList), [x+1, y+1])
if [x+1, y] not in nCList:
        nCList.insert(len(nCList), [x+1, y])
```


						[x-1, y+1]	[x, y+1]	[x+1, y+1]	
						[x-1, y]	qcell	[x+1, y]	

Τέλος έχουμε και την περίπτωση που ανήκει σε οποιοδήποτε εσωτερικό κελί :
else:

```
if [x-1, y+1] not in nCList:
        nCList.insert(len(nCList), [x-1, y+1])
if [x, y+1] not in nCList:
        nCList.insert(len(nCList), [x, y+1])
if [x+1, y+1] not in nCList:
        nCList.insert(len(nCList), [x+1, y+1])
if [x-1, y] not in nCList:
        nCList.insert(len(nCList), [x-1, y])
if [x+1, y] not in nCList:
        nCList.insert(len(nCList), [x+1, y])
if [x-1, y-1] not in nCList:
        nCList.insert(len(nCList), [x-1, y-1])
if [x, y-1] not in nCList:
        nCList.insert(len(nCList), [x, y-1])
if [x+1, y-1] not in nCList:
        nCList.insert(len(nCList), [x+1, y-1])
```

			[x-1, y+1]	[x, y+1]	[x+1, y+1]				
			[x-1, y]	qcell	[x+1, y]				
			[x-1, y-1]	[x, y-1]	[x+1, y-1]				

Επιστρέφουμε λοιπόν την nCList[] κάθε φορά ενημερωμένη.

def putOrder(aList, x, y, distance):

```
if len(aList)==0:
        aList.insert(len(aList), [x, y, distance])
        return aList

place=0
for a in aList:

        if distance<=a[2]:
                aList.insert(place, [x, y, distance])
                return aList
        else:
                place+=1


if [x, y, distance] not in a:
        aList.insert(len(aList), [x, y, distance])
        return aList
```

Η συγκεκριμένη συνάρτηση μας τοποθετεί τα στοιχεία σε αύξουσα σειρά ταξινόμησης. Θα χρειαστεί για *firstSpotNeighborCells[], cellList[], spotList[].*

Αν η λίστα που παίρνουμε ως όρισμα είναι κενή τότε απλά κάνουμε εισαγωγή στο τέλος της λίστας και γυρίζουμε την λίστα.

Αλλιώς, σαρώνουμε τα στοιχεία της λίστας και αν η distance είναι μικρότερη ή ίση τότε με την βοήθεια όπως και προηγουμένως μιας μεταβλητής place ξέρουμε που είναι η θέση που πρέπει να τοποθετηθεί το στοιχείο.

Αν το στοιχείο ήταν μεγαλύτερο από ‘όλα όσα είχε μέσα η λίστα ήδη δεν έχει προστεθεί, οπότε το τοποθετούμε στο τέλος της λίστας.

def orderedNSpots(q, cell):

```
spotList=[]
for l in dirList:
    numspots=0
    if l[0]==cell[0] and l[1]==cell[1]:

        with open('grid.grd', 'r', encoding='UTF-8') as dfgrd:
            dfgrd.seek(l[2])
            for row in dfgrd:
                numspots+=1
                row=row.split(' ')
                if numspots<=l[3]:
                    euclideanDist=math.sqrt((float(row[1])-q[0])**2+(float(row[2])-q[1])**2)
                    spotList=putOrder(spotList, float(row[1]), float(row[2]), euclideanDist)
        break

return spotList
```

Είναι η συνάρτηση που αναλαμβάνει κάθε φορά να επιστρέφει μια λίστα από ταξηνομημένα σημεία του κελιού που την κάλεσε.

Συγκεκριμένα, αρχικοποιούμε μια λίστα *spotList=[]* και κάθε φορά ψάχνουμε την λίστα *dirList[]* να δούμε ποιο είναι το κελί στο οποίο απευθυνόμαστε.

Όταν το βρούμε ανοίγουμε το αρχείο *'grid.grd'* για διάβασμα και ψάχνουμε με την βοήθεια των περιεχομένων της *dirList[]* την γραμμή στο οποίο ξεκινάει.

"Σπάμε" τα στοιχεία της γραμμής και χρησιμοποιούμε τον τύπο της ευκλείδιας απόστασης για να υπολογίσουμε την απόσταση των x, y συντεταγμένων από το σημείο q. Το αποτέλεσμα αναλαμβάνει να το εισάγει η συνάρτηση *putOrder().*

Κάνουμε break όταν τελειώσουμε με όλες τις γραμμές (πρέπει να μην ξεπεράσουμε τις *l[3])*

def mindist(q, b, ordCells, cell):

```
cellList=[]

for x in range(10):
    for y in range(10):

        if x==cell[0] and y==cell[1]:
            pass

        if [x,y] in nearestCells(cell) and [x,y] not in allVisitedCells:

            dividedRangeX=(float(b[1])-float(b[0]))/10
            dividedRangeY=(float(b[3])-float(b[2]))/10

            lowerXCellBound=float(b[0])+(x*dividedRangeX)
            upperXCellBound=float(b[0])+((x+1)*dividedRangeX)
            lowerYCellBound=float(b[2])+(y*dividedRangeY)
            upperYCellBound=float(b[2])+((y+1)*dividedRangeY)

            if q[0]>=upperXCellBound and q[1]>=lowerYCellBound and q[1]<=upperYCellBound:

                    minCellDist=q[0]-upperXCellBound

            elif q[0]<=lowerXCellBound and q[1]>=lowerYCellBound and q[1]<=upperYCellBound:

                    minCellDist=lowerXCellBound-q[0]

            elif q[0]>=lowerXCellBound and q[0]<=upperXCellBound and q[1]<=lowerYCellBound:

                    minCellDist=lowerYCellBound-q[1]

            elif q[0]>=lowerXCellBound and q[0]<=upperXCellBound and q[1]>=upperYCellBound:

                    minCellDist=q[1]-upperYCellBound

            elif q[0]>upperXCellBound and q[1]<lowerYCellBound:

                    minCellDist=math.sqrt((q[0]-upperXCellBound)**2+(lowerYCellBound-q[1])**2)

            elif q[0]<lowerXCellBound and q[1]<lowerYCellBound:
```

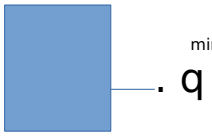
```
                    minCellDist=math.sqrt((lowerXCellBound-q[0])**2+(lowerYCellBound-q[1])**2)
            elif q[0]>upperXCellBound and q[1]>upperYCellBound:
                    minCellDist=math.sqrt((q[0]-upperXCellBound)**2+(q[1]-upperYCellBound)**2)
            elif q[0]<lowerXCellBound and q[1]>upperYCellBound:
                    minCellDist=math.sqrt((lowerXCellBound-q[0])**2+(q[1]-upperYCellBound)**2)
            if [x, y, minCellDist] not in priorityQueue and [x, y, minCellDist] not in ordCells:
                    cellList=putOrder(cellList, x, y, minCellDist)
    return cellList
```

Η συνάρτηση αυτή υπολογίζει την ελάχιστη ευκλείδια απόσταση του σημείου q από τα κοντινότερα πάντα γειτονικά του κελιά (τα τοποθετεί όλα σε μια λίστα *cellList[]* μαζί με τις αποστάσεις τους ταξηνομημένα με αύξουσα σειρά). Χρησιμοποιούμε μια nested loop για το x, y ώστε να τσεκάρουμε κάθε φορά σε σχέση με τα αντίστοιχα *lowerXCellBound, upperXCellBound, lowerYCellBound, upperYCellBound* που βρισκόμαστε ώστε να εντοπίσουμε ποια είναι η ελάχιστη απόσταση του x,y από το q.

Για τις συντεταγμένες του κελιού που περιλαμβάνει το q απλά κάνουμε pass.

Εαν το x,y ανήκει στα *nearestCells() (*καλούμε την συνάρτηση στο σημείο αυτό) και το x,y δεν το έχουμε ακόμα επισκεπτεί (όχι στην *allVisitedCells[]*) τότε έχουμε 8 πιθανές περιπτώσεις για την απόσταση του συγκεκριμένου κελιού από το q.

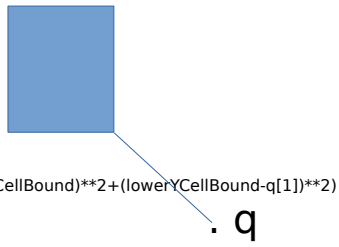
CASE 1
q[0]>=upperXCellBound and q[1]>=lowerYCellBound and q[1]<=upperYCellBound:



minCellDist=q[0]-upperXCellBound

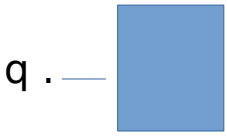
Εδώ εφόσον το σημείο είναι φραγμένο για y,
η διαφορά των y στον τύπο είναι 0.

CASE 5
q[0]>upperXCellBound and q[1]<lowerYCellBound:



minCellDist=math.sqrt((q[0]-upperXCellBound)**2+(lowerYCellBound-q[1])**2)

CASE 2
q[0]<=lowerXCellBound and q[1]>=lowerYCellBound and q[1]<=upperYCellBound:

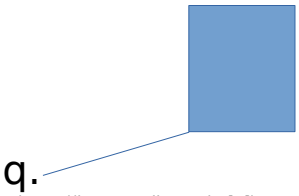


minCellDist=lowerXCellBound-q[0]

Όμοια, είναι φραγμένο για y,

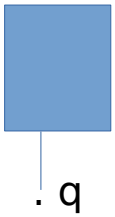
η διαφορά των y στον τύπο είναι 0.

CASE 6
q[0]<lowerXCellBound and q[1]<lowerYCellBound:



minCellDist=math.sqrt((lowerXCellBound-q[0])**2+(lowerYCellBound-q[1])**2)

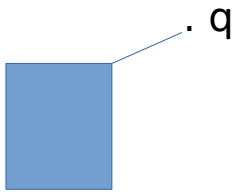
CASE 3
q[0]>=lowerXCellBound and q[0]<=upperXCellBound and q[1]<=lowerYCellBound:



minCellDist=lowerYCellBound-q[1]

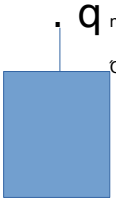
Εδώ εφόσον το σημείο είναι φραγμένο για x,
η διαφορά των x στον τύπο είναι 0.

CASE 7
q[0]>upperXCellBound and q[1]>upperYCellBound:



minCellDist=math.sqrt((q[0]-upperXCellBound)**2+(q[1]-upperYCellBound)**2)

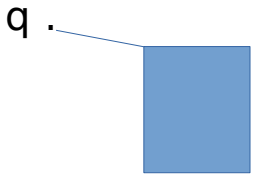
CASE4
q[0]>=lowerXCellBound and q[0]<=upperXCellBound and q[1]>=upperYCellBound:



minCellDist=q[1]-upperYCellBound

Όμοια, είναι φραγμένο για x,
η διαφορά των x στον τύπο είναι 0.

CASE 8
q[0]<lowerXCellBound and q[1]>upperYCellBound:

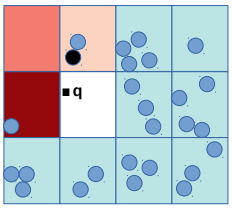


minCellDist=math.sqrt((lowerXCellBound-q[0])**2+(q[1]-upperYCellBound)**2)

Αφού καταλήξουμε σε κάποια από αυτές, ελέγχουμε αν [x, y, minCellDist] που βρήκαμε δεν υπάρχει ήδη στην priorityQueue και στην *ordCells[].* Καλούμε την *putOrder()* για την σωστή τοποθέτηση στην λίστα *cellList[]* και τέλος την επιστρέφουμε.

Γενικά, τα κελιά στο grid δεν είναι τετράγωνα (dividedRangeX=0.049982099999999972 και dividedRangeY=0.06495100000000065) οπότε μοιάζουν στην πραγματικότητα κάπως έτσι:

Γι'αυτο και κάθε φορά σορτάρουμε και την firstSpotNeighborCells[] διότι μπορεί ένα κελί να βρίσκεται από άποψη απόστασης πιο κοντά στο q αλλά να μην έχει σημεία πιο κοντά από οτι ένα λίγο πιο μακρινό κελί.



Εδώ το κοντινότερο κελί στο q μπορεί να είναι το "κόκκινο" αλλά παρατηρούμε ότι το σημείο του κόκκινου κελιού δεν είναι το πλησιέστερο.

Το επόμενο πλησιέστερο κελί είναι το ακριβώς από πάνω, αλλά τυχαίνει να μην έχει καν σημεία. Το κελί που τελικά περιέχει το κοντινότερο σημείο **είναι το 3ο από άποψη απόστασης κελιών από το q.**

Συνεπώς, χρειάζεται να έχουμε όλα τα γειτονικά κελιά κάθε φορά και να γνωρίζουμε το πρώτο τους σημείο μαζί με την απόσταση του από το q για να επιστρέφουμε τον σωστό κοντινότερο γείτονα.