Αλέξανδρος Αριστόβουλος

1063199

Εργασία Running Median

Δημιουργία τοποθεσίας με θερμοκρασία

Για τη δημιουργία της τυχαίας θερμοκρασίας διαλέγουμε τυχαίο αριθμό (με 2 δεκαδικά) μεταξύ του -50 έως το +50. Επιπλέον διαλέγουμε δύο τυχαία ακέραια νούμερα μεταξύ του 0 και του 1000 σαν συντεταγμένες x και y τα οποία και τα ενώνουμε για τα χρησιμοποιήσουμε σαν αναγνωριστικό (key) της τοποθεσίας.

```
def createSpot():
    x = np.random.randint(0, MAX_POSITION)
    y = np.random.randint(0, MAX_POSITION)
    key = str(x)+str(y)
    temp = MAX_TEMP * np.random.random()
    temp = round(temp, TEMP_DECIMALS)
    return key, temp
```

Αυτά τα δεδομένα τα χρησιμοποιούμε στη κλάση Spot() για να τα έχουμε πιο οργανωμένα.

```
#class to store the spot info
class Spot:
    def __init__(self, key, temp):
        self.key = key
        self.temp = temp
```

Κλάση MinHeap

Για την αποθήκευση όλων των στοιχείων και για την εύρεση του median χρησιμοποιούμε 2 heaps. Ένα heap αποθηκεύουμε τις θερμοκρασίες που είναι κάτω από το median (με το max στη κορυφή του heap) και στο άλλο τις θερμοκρασίες που είναι πάνω από το median (με το min στη κορυφή του heap). Για αυτό το λόγο δημιουργούμε δική μας κλάση MinHeap την οποία χρησιμοποιούμε για τη δημιουργία των heaps.

Initialization της κλάσης MinHeap

Για το initialization της MinHeap δημιουργούμε κενό array στο οποίο θα αποθηκεύουμε σαν tuple το key και τη θερμοκρασία της τοποθεσίας. Επίσης,

φτιάχνουμε και ένα dictionary στο οποίο θα αποθηκεύουμε τα key μαζί με την τοποθεσία που μπορούμε να τα βρούμε στο array. Τέλος, χρησιμοποιούμε τη μεταβλητή size για να αποθηκεύουμε το μήκος του array.

Αφού έχουμε δημιουργήσει όλα αυτά η συνάρτηση χρησιμοποιεί το όρισμα που είναι ένα array κλάσεις Spot που θέλουμε να βάλουμε στο MinHeap, βάζει τα δεδομένα στις κατάλληλες μεταβλητές και τέλος (μέσω της συνάρτησης heapify) τα αποθηκεύει σε μορφή heap.

```
def __init__(self, spots):
        # tuple (key, value)
        self.array = []
        # position of key in array
        self.pos = {}
        # elements in min heap
        self.size = len(spots)

        #add all the elements of the list to heap
        for i, item in enumerate(spots):
            self.array.append((spots[i].key, spots[i].temp))
            self.pos[spots[i].key] = i

        #sort the elements into a heap
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
```

Συνάρτηση heapify της κλάσης MinHeap

Για να έχουμε δομή min heap πρέπει ο κάθε γονιός να είναι μικρότερος από τα παιδιά του. Για αυτό το λόγο βρίσκουμε τη μικρότερη τιμή μεταξύ του γονιού και των 2 παιδιών του και αν δεν έχει τη μικρότερη τιμή ο γονιός τότε τον αλλάζουμε θέση με το παιδί που την έχει. Έπειτα συνεχίζουμε και ελέγχουμε αναδρομικά γονιούς και παιδιά μέχρις ότου ο γονιός να είναι μικρότερος.

```
# modify array so that i roots a heap, down-heap
    def heapify(self, i):
        #assume that this elements is the smallest
        smallest = i
        #get the children
        le = 2 * i + 1
        ri = 2 * i + 2

        #check if the left child is smaller
        if le < self.size and self.array[le][1] < self.array[smallest][1]:
            smallest = le
        #check if the right child is smaller
        if ri < self.size and self.array[ri][1] < self.array[smallest][1]:
            smallest = ri

        #if the smallest element is not the initial change positions and call
recursively
        if smallest != i:
            # update pos
            self.pos[self.array[smallest][0]] = i
            self.pos[self.array[i][0]] = smallest
            # swap
```

```
            self.array[smallest], self.array[i] = self.array[i],
self.array[smallest]
            self.heapify(smallest)
```

Συνάρτηση isEmpty της κλάσης MinHeap

Αυτή η συνάρτηση ελέγχει αν το μέγεθος το heap είναι 0

```
    #check if heap is empty
    def isEmpty(self):
        return self.size == 0
```

Συνάρτηση getMin της κλάσης MinHeap

Αφού το ελάχιστο είναι πάντα στη κορυφή του heap, για να το βρούμε αρκεί να διαβάσουμε την τιμή της κορυφής

```
    # return the min element of the heap (it is always the top element)
    def getMin(self):
        if self.size == 0:
            return None
        return self.array[0]
```

Συνάρτηση extractMin της κλάσης MinHeap

Για να αφαιρέσουμε το min του heap επιστρέφουμε την τιμή της κορυφής και έπειτα βάζουμε το τελευταίο στοιχείο στη θέση του. Διαγράφουμε το στοιχείο και από το dictionary και ενημερώνουμε τη τιμή του size μειώνοντας την κατά ένα. Έπειτα καλούμε τη συνάρτηση heapify στη κορυφή για να ταξινομήσουμε πάλι τα στοιχεία με σωστό τρόπο.

```
# return and remove the min element of the heap
    def extractMin(self):
        #if heap is empty do nothing
        if self.size == 0:
            return None

        #if it has elements then we need to remove the top element
        root = self.array[0]
        lastNode = self.array[self.size - 1]
        self.array[0] = lastNode
        #update pos
        self.pos[lastNode[0]] = 0
        del self.pos[root[0]]
        self.size -= 1
        self.heapify(0)
        return root
```

Συνάρτηση decreaseTemp της κλάσης MinHeap

Για να μειώσουμε τη θερμοκρασία ενός σημείου πρέπει έπειτα να το βάλουμε στη σωστή θέση. Για αυτό το λόγο, αφού κάνουμε την αλλαγή θερμοκρασίας τσεκάρουμε το γονιό του και αν και ανταλλάσσουμε θέσεις με το γονιό αν έχει πιο μικρή τιμή. Η διαδικασία συνεχίζεται αναδρομικά μέχρι ο γονιός του να έχει μικρότερη τιμή από αυτό ή μέχρι να φτάσει στη κορυφή.

```
# decreace value of item (key, value)
    def decreaseTemp(self, key, new_temp):
        i = self.pos[key]

        # new value must be smaller than current
        if self.array[i][1] <= new_temp:
            return

        self.array[i] = (key, new_temp)
        # check if is smaller than parent
        p = (i - 1) // 2
        while i > 0 and self.array[i][1] < self.array[p][1]:
            # update pos
            self.pos[self.array[i][0]] = p
            self.pos[self.array[p][0]] = i
            # swap
            self.array[p], self.array[i] = self.array[i], self.array[p]
            i = p
            p = (i - 1) // 2
```

Συνάρτηση increaseTemp της κλάσης MinHeap

Αυξάνουμε την τιμή του σημείου και το βάζουμε στην κορυφή του heap. Έπειτα καλώντας τη συνάρτηση heapify το ταξινομούμε στη σωστή θέση.

```
# increace value of item (key, value)
    def increaseTemp(self, key, new_temp):
        i = self.pos[key]
        # new value must be greater than current
        if self.array[i][1] >= new_temp:
            return
        self.array[i] = (key, new_temp)
        # check children
        self.heapify(i)
```

Συνάρτηση insert της κλάσης MinHeap

Για να προσθέσουμε ένα στοιχείο του δίνουμε μία πολύ μεγάλη τιμή και το βλάζουμε στο τέλος του heap. Έπειτα, με τη συνάρτηση decreaseTemp το μετακινούμε στη σωστή θέση και δεν ξεχνάμε να αυξήσουμε το μέγεθος του heap.

```
    # insert item (key, value)
    def insert(self, spot):
        # insert an item at the end with big value
        if self.size < len(self.array):
```

```
            self.array[self.size] = (spot.key, 10**80)
        else:
            self.array.append((spot.key, 10**80))

        self.pos[spot.key] = self.size
        self.size += 1
        self.decreaseTemp(spot.key, spot.temp)
```

Συνάρτηση isInMinHeap της κλάσης MinHeap

Για να ελέγξουμε αν ένα στοιχείο βρίσκεται στο heap αρχικά ελέγχουμε αν είναι μέσα στο dictionary. Αν είναι μέσα στο dictionary ελέγχουμε αν η θέση που δείχνει βρίσκεται στα όρια του heap. Τότε και μόνο τότε το στοιχείο βρίσκεται στο heap.

```
    def isInMinHeap(self, key):
        key = str(key)
        if key in self.pos:
            if self.pos[key] <= self.size:
                return True

        return False
```

Λύση του προβλήματος

Χρησιμοποιούμε 2 δομές της κλάσης MinHeap που αναλύθηκε παραπάνω. Το ένα heap έχει τις τιμές που είναι πάνω από το μέσο όρο και έχει σαν κορυφή το μικρότερο στοιχείο. Το άλλο heap έχει τις τιμές που είναι κάτω από το μέσο όρο και στην κορυφή έχει το μέγιστο των τιμών αυτών. Για να το πετύχουμε αυτό βάζουμε τις τιμές με το ανάποδο πρόσημο και όταν τις εξάγουμε δεν ξεχνάμε να τις επαναφέρουμε στο σωστό πρόσημο.

Αρχικά, ξεκινάμε δίνοντας από μία θέση στα δύο heap.

```
#start by adding 1 spot for each heap
key, temp = createSpot()
spot = Spot(key, temp)
above_median_heap = MinHeap([spot])
#save to the array
spots = [spot]

#in the below_median_heap we store the temp with the opposite sign
#in order to use it as a max heap
key, temp = createSpot()
spot = Spot(key, -temp)
below_median_heap = MinHeap([spot])
#save to the array
spots.append(spot)
```

(Αποθηκεύουμε NUMBER_OF_SPOTS_IN_ARRAY θέσεις σε ένα ξεχωριστό array για μελλοντική χρήση)

Μετά φτιάχνουμε τα στοιχεία των heaps έτσι ώστε το below_median_heap να έχει μικρότερες τιμές από το above_median_heap με τη χρήση της συνάρτησης fixHeapValues

Συνάρτηση fixHeapValues

Χρησιμοποιούμε την συνάρτηση fixHeapValues για να φτιάξουμε τα δύο heap έτσι ώστε το above_median_heap να έχει μεγαλύτερες τιμές από το below_median_heap. Αυτό επιτυγχάνεται ελέγχοντας τις κορυφές των 2 heap και αν το below_median_heap έχει μεγαλύτερη τιμή από το above_median_heap τότε βγάζουμε τις 2 κορυφές και τις βάζουμε στο ανάποδο heap. Η διαδικασία επαναλαμβάνεται μέχρις ότου να ισχύει ότι η τιμή της κορυφής του above_median_heap είναι μεγαλύτερη της τιμής της κορυφής του below_median_heap.

```
#the below_median_heap should have a lower max value than the min of
above_median_heap
def fixHeapValues(below_median_heap, above_median_heap):
    while -below_median_heap.getMin()[1] > above_median_heap.getMin()[1]:
        #get the top spots from the heap
        old_below_key,old_below_temp = below_median_heap.extractMin()
        #in the below_median_heap we store the temp with the opposite sign
        #in order to use it as a max heap
        old_below_temp = -old_below_temp
        old_below_spot = Spot(old_below_key, old_below_temp)

        old_above_key,old_above_temp = above_median_heap.extractMin()
        #in the below_median_heap we store the temp with the opposite sign
        #in order to use it as a max heap
        old_above_temp = -old_above_temp
        old_above_spot = Spot(old_above_key, old_above_temp)

        #switch the spots
        below_median_heap.insert(old_above_spot)
        above_median_heap.insert(old_below_spot)
```

Έχοντας φτιάξει τα δύο heaps βρίσκουμε το median με της συνάρτησης calculateMedian.

Εύρεση του median με τη συνάρτηση calculateMedian

Αν έχουμε μονό αριθμό στοιχείων το median είναι στη κορυφή του heap που έχει το ένα παραπάνω στοιχείο. Αν έχουμε ζυγό αριθμό στοιχείων υπολογίζουμε το median σαν τον μέσο όρο των 2 κορυφών. (Προσέχουμε την τιμή του below_median_heap που είναι με αντίθετο πρόσημο)

```
def calculateMedian(below_median_heap, above_median_heap):
    #if both heaps are empty there is no median
    if below_median_heap.size == 0 and above_median_heap.size == 0:
        return None
```

```
    #if a heap has 1 more element then the median is at the top of the heap
    #be careful with values from the below_median_heap
    elif below_median_heap.size == above_median_heap.size + 1:
        return -below_median_heap.getMin()[1]

    elif above_median_heap.size == below_median_heap.size + 1:
        return above_median_heap.getMin()[1]

    #if both heaps have the same number of elements then calculate the average
    else:
        median = (-below_median_heap.getMin()[1] +
above_median_heap.getMin()[1])/2
        #round to 2 decimals
        median = round(median, TEMP_DECIMALS)
        return median
```

Μέσα σε for loop δημιουργούμε όσα στοιχεία θέλουμε και τα βάζουμε ένα-ένα στα heaps. Ο τρόπος που τα βάζουμε είναι να ελέγχοντας τη θερμοκρασία του καινούργιου σημείου, και ανάλογα με την τιμή της, το βάζουμε είτε στο below_median_heap είτε στο above_median_heap (με τη βοήθεια της προηγούμενης median τιμής). Αφού το εισάγουμε ισορροπούμε τα 2 heaps έτσι ώστε το size τους να διαφέρει το πολύ κατά 1.

Ιδιαίτερη προσοχή δίνεται έτσι ώστε αν ένα σημείο υπάρχει ήδη σε κάποιο heap να μην το ξανά-εισάγουμε αλλά να ενημερώσουμε τη θερμοκρασία του. Αυτό το επιτυγχάνουμε με τη συνάρτηση updateExistingSpots η οποία το βρίσκει, το ενημερώνει καταλλήλως και επιστρέφει True ή False για να ξέρουμε αν ενημέρωσε στοιχείο ή όχι.

Συνάρτηση updateExistingSpots

```
def updateExistingSpots(below_median_heap, above_median_heap):
    #check if we already have the spot in order to update it
    if above_median_heap.isInMinHeap(spot.key):
        index = above_median_heap.pos[str(spot.key)]
        if spot.temp > above_median_heap.array[index][1]:
            above_median_heap.increaseTemp(spot.key, spot.temp)
        else:
            above_median_heap.decreaseTemp(spot.key, spot.temp)

        fixHeapValues(below_median_heap, above_median_heap)
        return True

    elif below_median_heap.isInMinHeap(spot.key):
        #dont forget to adjust the temp
        spot.temp = - spot.temp
        index = below_median_heap.pos[str(spot.key)]
        if spot.temp > below_median_heap.array[index][1]:
            below_median_heap.increaseTemp(spot.key, spot.temp)
        else:
            below_median_heap.decreaseTemp(spot.key, spot.temp)

        fixHeapValues(below_median_heap, above_median_heap)
        return True

    return False
```

Αφού τελειώσει όλη η διαδικασία, χρησιμοποιούμε τα στοιχεία που αποθηκεύσαμε στον πίνακα spots για να ενημερώσουμε τις τιμές τους στα heaps και να βεβαιωθούμε για την λειτουργία της updateExistingSpots.

Αποτελέσματα κώδικα για 25 στοιχεία με 20 στοιχεία τα οποία αλλάζουμε στο τέλος

```
Above Median Heap
Array: (130237 : 38.44)
Below Median Heap
Array: (217558 : -21.25)
Total time for  25 number of passes is  0.0030024051666259766
Above Median Heap
Array: (586425 : 24.59) (590983 : 32.08) (695220 : 29.71) (716818 : 35.55) (236507 : 33.99) (356339 : 36.49) (2181
: 47.65) (331832 : 48.91) (130237 : 38.44) (509961 : 41.83) (891891 : 38.25) (836258 : 46.22) (321891 : 45.18)
Below Median Heap
Array: (663288 : -22.78) (217558 : -21.25) (237809 : -21.98) (903870 : -16.12) (730445 : -15.72) (129153 : -21.08)
(519925 : -14.58) (86284 : -9.78) (85169 : -10.48) (13168 : -7.08) (186402 : -7.22) (658618 : -19.24)
Median:  24.59

Total time for the whole program is  0.006005525588989258
Above Median Heap
Array: (663288 : 22.78) (590983 : 32.08) (586425 : 24.59) (716818 : 35.55) (236507 : 33.99) (695220 : 29.71) (2181
: 47.65) (331832 : 48.91) (130237 : 38.44) (509961 : 41.83) (891891 : 38.25) (836258 : 46.22) (321891 : 45.18)
Below Median Heap
Array: (237809 : -21.98) (217558 : -21.25) (356339 : -21.68) (903870 : -16.12) (730445 : -15.72) (129153 : -21.08)
(519925 : -14.58) (86284 : -9.78) (85169 : -10.48) (13168 : -7.08) (186402 : -7.22) (658618 : -19.24)
Median:  22.78
```

Χρόνος εκτέλεσης προγράμματος για 500.000 στοιχεία και 100 στοιχεία που αλλάζουμε στο τέλος

```
Total time for  500000 number of passes is  101.10304093360901
Total time for the whole program is  101.13107514381409
```

(Ο πρώτος χρόνος είναι για να δημιουργήσει και να αποθηκεύσει ένα-ένα όλα τα στοιχεία και ο δεύτερος χρόνος έχει τον επιπλέον χρόνο που χρειάζεται για να αλλάξει και τα 100 στοιχεία της λίστας)