ΑΛΓΟΡΙΘΜΟΙ ΚΑΙ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΠΡΩΤΗ ΑΤΟΜΙΚΗ ΕΡΓΑΣΙΑ

ΠΡΟΒΛΗΜΑ MAXIMUM SUB-ARRAY SUM

ΟΝΟΜΑΤΕΠΩΝΥΜΟ	ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ	ΗΜΕΡΟΜΗΝΙΑ	ΥΛΟΠΟΙΗΣΗ
Αλέξανδρος Τσακιρίδης	up1083879	Μάρτιος 2023	Python 3.9

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΓΡΑΦΗ ΠΡΟΒΛΗΜΑΤΟΣ

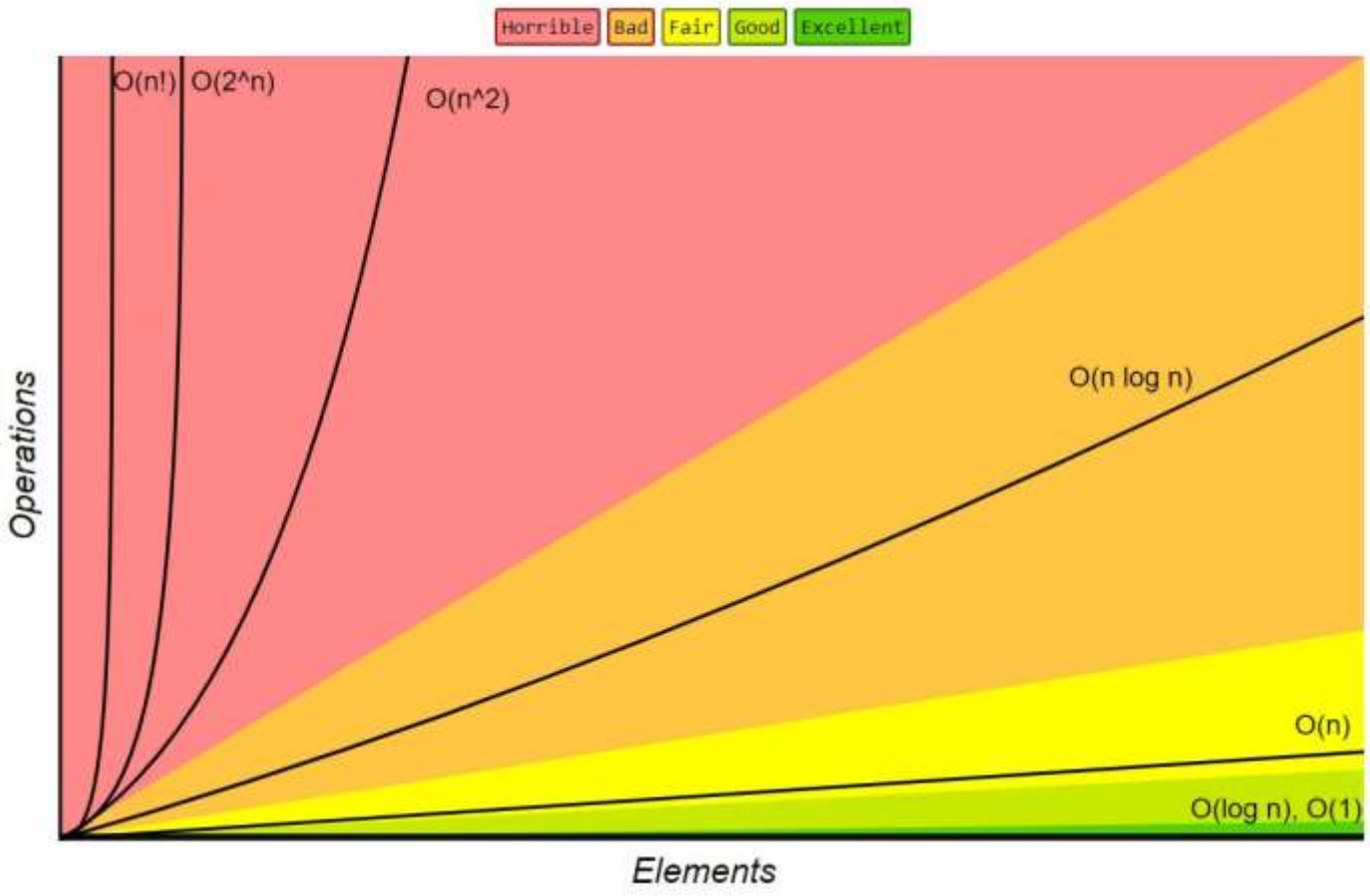
Δίνεται πίνακας θετικών και αρνητικών ακεραίων και ζητείται η εύρεση του μεγαλύτερου υποπίνακα συνεχόμενων στοιχείων με το μέγιστο άθροισμα.

ΠΕΡΙΓΡΑΦΗ ΛΥΣΗΣ

Θα γραφτούν τέσσερεις αλγόριθμοι για τη λύση, ο πρώτος πολυπλοκότητας $O(n^3)$, ο δεύτερος πολυπλοκότητας $O(n^2)$, ο τρίτος πολυπλοκότητας O(nlogn) και ο τέταρτος πολυπλοκότητας O(n). Θα δοκιμαστούν στα ίδια δεδομένα εισόδου και θα ελεγχθεί η αποδοτικότητα του καθενός.

Θα χρησιμοποιηθούν built-in βιβλιοθήκες της Python, η κλάση time της time για την χρονομέτρηση του κάθε αλγορίθμου και η συνάρτηση randint της random για την δημιουργία ψευδοτυχαίων ακεραίων. Η συνάρτηση seed της τελευταίας θα καλείται με παράμετρο έναν αριθμό μητρώου, για την αρχικοποίηση της γεννήτριας ψευδοτυχαίων αριθμών.

ΣΥΓΚΡΙΣΗ ΠΟΛΥΠΛΟΚΟΤΗΤΑΣ ΑΛΓΟΡΙΘΜΩΝ



ΟΡΓΑΝΩΣΗ ΠΡΟΒΛΗΜΑΤΟΣ

Περνάμε στην υλοποίηση του προγράμματος. Πρώτο βήμα θα είναι το να αποφασίσουμε ποιες κλάσεις απαιτούνται. Αφού γίνει αυτό, θα προχωρήσουμε στην δημιουργία scripts που θα δημιουργούν σετ δεδομένων και θα ελέγχουν την λειτουργία των διαφορετικών μεθόδων.

Η ΚΛΑΣΗ DATA

```
class Data:
```

Για την αποθήκευση των αποτελεσμάτων από τους αλγορίθμους, την χρονομέτρηση και τον έλεγχο της συμπεριφοράς τους, θα δημιουργήσουμε μια κλάση Data, σε αντικείμενο της οποίας αρχικά θα δημιουργείται ο πίνακας των αριθμών με επιθυμητό πλήθος. Ο κάθε αλγόριθμος θα παίρνει ως παράμετρο ένα τέτοιο αντικείμενο και στα attributes του θα γράφει πληροφορίες όπως η περιγραφή της μεθόδου που ακολουθείται, το μέγιστο άθροισμα, τα δύο indices που ορίζουν το sub-array (αριστερό και δεξί άκρο), ο συνολικός αριθμός loop iterations και το χρονικό διάστημα που απαιτήθηκε για την εκτέλεση. Με αυτόν τον τρόπο, οι συναρτήσεις που θα υλοποιούν τους ζητούμενους αλγορίθμους, θα δέχονται για είσοδο ένα αντικείμενο Data και δεν θα χρειάζεται να επιστρέφουν κάτι.

> Καθώς η εκφώνηση ορίζει οι αριθμοί να είναι στο διάστημα [-100, 100], ορίζουμε ένα integer class attribute με το όνομα MAX_ABS και του δίνουμε την τιμή 100. Σύντομα, θα φροντίσουμε η μέγιστη απόλυτη τιμή των τυχαίων ακεραίων που θα δημιουργούνται να είναι ίση με MAX_ABS.
>
> Υπάρχει περίπτωση κάποια από τις μεθόδους να μην υπολογίζει σωστά τα indices του διαστήματος, ακόμα κι αν βρίσκει το σωστό άθροισμα. Σε αυτήν την περίπτωση θα χρησιμοποιείται η σταθερά INVALID_INDEX, με τιμή -1.

Προχωράμε στην υλοποίηση της κλάσης Data, γράφοντας την dunder μέθοδο *__init__*. Μετά τα απαραίτητα import statements από το built-in module typing, ορίζουμε τον τύπο των παραμέτρων της μεθόδου και τους τύπους του κάθε attribute που θα δημιουργεί. Ακόμα και αν στην εκτέλεση αυτές οι πληροφορίες δεν θα έχουν σημασία, θα βοηθήσουν πολύ στην αλληλεπίδραση με αντικείμενα Data και γενικότερα στην ανάπτυξη του υπόλοιπου προγράμματος.

```
def __init__(self, amount: int = 1000) -> NoReturn:
    self.amount: int = amount
    self.numbers: List[int] = []
    self.left: int = 0
    self.right: int = 0
    self.summation: int = 0
    self.iterations: int = 0
    self.description: str = str()
    self.start_time: Optional[time] = None
    self.end_time: Optional[time] = None
    for element in range(amount): self.numbers.append(randint(-Data.MAX_ABS, Data.MAX_ABS))
    return
```

Πριν από οτιδήποτε άλλο, πάνω από την *__init__* ορίζουμε τα *__slots__* της Data, για να γίνει πιο αποδοτική στην διαχείριση μνήμης και την ταχύτητα:

```
__slots__: Tuple[str] = ("amount", "numbers", "left", "right", "summation", "iterations",
                         "description", "start_time", "end_time")
```

Προχωράμε με την προσθήκη άλλων instance methods και properties στην Data, απαραίτητων για την συνέχεια.

```
@property
def duration(self) -> int:
    try: assert self.end_time is not None and self.start_time is not None
    except AssertionError: print("start_time and end_time are not set!")
    return round(self.end_time - self.start_time, 4)

def __repr__(self) -> str: return f"DATA SET: {self.amount} numbers"

def __str__(self) -> str:
    out: str = f"FINAL ANSWER:\t Subarray of {self.right - self.left + 1} numbers from position "
    out += f"{self.left} until position {self.right} with sum: {self.summation}"
    out += " " * (100 - len(out)) + f"Duration: {self.duration} seconds"
    out += " " * (130 - len(out)) + f"Total iterations: {self.iterations}"
    return out + " " * (165 - len(out)) + "Description: " + self.description

def __eq__(self, other) -> bool: return self.numbers == other.numbers

@property
def subarray(self) -> str:
    return "\nSUBARRAY: " + str().join([str(self.numbers[i]) + "   " for i in range(self.left, self.right + 1)])

def reset(self) -> NoReturn:
    self.summation, self.left, self.right = 0, 0, 0
    self.start_time, self.end_time, self.description = None, None, str()
    return
```

Το property method duration θα επιστρέφει το χρονικό διάστημα που απαιτείται για την εκτέλεση του κάθε αλγορίθμου, με ακρίβεια τεσσάρων δεκαδικών ψηφίων. Το assert statement σιγουρεύει ότι οι αλγόριθμοι θα αποθηκεύουν σωστά τους δύο χρόνους, οι οποίοι είναι δυνατόν να αφαιρεθούν έτσι γιατί είναι floats. Καθένας από αυτούς είναι το πλήθος των δευτερολέπτων που παρήλθαν από την 1η Ιαν του 1970 (system dependent), όπως μας πληροφορεί η κλήση της time.gmtime με παράμετρο 0.

> Μια άλλη προσέγγιση δεν θα αποθήκευε στο αντικείμενο Data τους χρόνους έναρξης και τερματισμού αλλά θα δημιουργούσε ένα custom higher-order function (decorator) το οποίο θα μετρούσε το χρονικό διάστημα της εκτέλεσης και θα το τύπωνε στο τέλος. Στην δική μας περίπτωση, αυτή η λύση θα δημιουργούσε προβλήματα, όπως το ότι δεν θα έδινε την ευχέρεια να μετράμε χρονικά διαστήματα ακριβώς στο σημείο που θέλουμε.
>
> ```
> @timedelta_counter
> def algorithm_name(*args, **kwargs): ...
> ```

Το dunder method *__repr__* θα καλείται για να τυπώνεται ο κάθε πίνακας αριθμών που θα δημιουργείται, μαζί με το μήκος του.

Το dunder method *__str__* θα καλείται μετά από την εκτέλεση του κάθε αλγορίθμου, για να βλέπουμε τα αποτελέσματά του. Θα τυπώνει το μέγιστο άθροισμα, τα indices στα όριά του, την περιγραφή του αλγορίθμου, τον χρόνο εκτέλεσης και τα συνολικά loop iterations που απαιτήθηκαν. Αποφεύγουμε την χρήση χαρακτήρων tab, καθώς δεν εξασφαλίζουν την καλή εκτύπωση του αντικειμένου για μικρούς και μεγάλους αριθμούς.

Το dunder method *__eq__* υλοποιείται σε περίπτωση που ο χρήστης της κλάσης θέλει να συγκρίνει με τον τελεστή ισότητας (==) δύο αντικείμενα Data. Δεν θα χρησιμοποιηθεί για την ώρα.

Το property method subarray θα καλείται για λόγους debugging, για να επιβεβαιώνουμε ότι ο υποπίνακας που επέστρεψαν οι αλγόριθμοι είναι ο σωστός, τυπώνοντας τα στοιχεία αυτά.

Το method reset θα καλείται μετά την εκτέλεση του κάθε αλγορίθμου, για να αρχικοποιεί εκ νέου τα attributes που θα επεξεργαστεί ο επόμενος αλγόριθμος. Ίσως κάτι τέτοιο να μην είναι απαραίτητο.

Η ΚΛΑΣΗ ALGORITHMS

```
class Algorithms:
```

Οι διαφορετικοί αλγόριθμοι που θα αναπτυχθούν, θα οριστούν ως στατικές μέθοδοι μιας κλάσης, της Algorithms. Η κάθε τέτοια μέθοδος θα μπορεί να καλείται είτε από αντικείμενα Algorithm είτε από την ίδια την κλάση.

Κάποιος κακοπροαίρετος θα μπορούσε να κάνει την εξής διακριτική προσθήκη στην κλάση:

```
def __new__(cls, *args, **kwargs) -> NoReturn: return None
```

Μετά από ένα τέτοιο overriding της *__new__*, οι χρήστες της κλάσης δεν θα μπορούν να δημιουργήσουν αντικείμενά της όσο και να θέλουν. Είναι σίγουρα μια καλή ευκαιρία να αναλογιστούμε την ευρηματικότητα των σχεδιαστών της γλώσσας και την ελευθερία που δίνουν στους χρήστες της. Εν πάση περιπτώσει, δεν κάνουμε κάτι τέτοιο καθώς θέλουμε την δημιουργία αντικειμένου Algorithms, μόνο και μόνο για να μπορεί να χρησιμοποιείται σαν iterator στα loops. Κάνουμε override την *__iter__* της κλάσης για να επιστρέφει ένα list από Callable, αφού πρώτα το μετατρέπει σε iterator. Με αυτόν τον τρόπο, αργότερα το αντικείμενο Algorithms θα μπαίνει στο for loop όπως θα έμπαινε ένα οποιαδήποτε collection.

```
def __iter__(self) -> Iterable: return iter([Algorithms.my_method, Algorithms.max_subarray_on,
                                             Algorithms.max_subarray_on2, Algorithms.max_subarray_on3,
                                             Algorithms.max_subarray_recursive])
```

Καθώς δεν επιθυμούμε κάποιο instance attribute, παραλείπουμε την *__init__* και υλοποιούμε τον κάθε αλγόριθμο σύμφωνα με το ακόλουθο πρότυπο:

```
@staticmethod
def algorithm_name(data: Data) -> NoReturn: ...
```

Φροντίζουμε όλοι οι αλγόριθμοι να δέχονται την ίδια είσοδο (αντικείμενο Data) και να επιστρέφουν την ίδια έξοδο (None). Αυτή η πρακτική θα μας επιτρέψει αργότερα να τους καλούμε με τη σειρά χωρίς προβλήματα, βάζοντας τα ονόματά τους σε λίστα.

Επίσης, ο κάθε αλγόριθμος θα μετρά τα loop iterations αυξάνοντας την τιμή της data.iterations και θα αποθηκεύει τον χρόνο έναρξης και τερματισμού για να υπολογιστεί αργότερα το χρονικό διάστημα της εκτέλεσης. Οι απαντήσεις είναι κάθε φορά τρεις και θα αποθηκεύονται στα attributes data.left, data.right και data.summation.

ΑΛΓΟΡΙΘΜΟΣ ΜΕ ΠΟΛΥΠΛΟΚΟΤΗΤΑ $O(n^3)$

```
@staticmethod
def max_subarray_on3(data: Data) -> NoReturn:  # ---------------------------------------------- O(n^3) COMPLEXITY
    data.description = "Method with O(n^3)"
    data.start_time = time()  # ------------------------------------------------------------------ START TIME
    for i in range(len(data.numbers)):
        for j in range(i, len(data.numbers)):
            temporary = 0
            for k in range(i, j + 1):
                data.iterations += 1
                temporary += data.numbers[k]
            if temporary > data.summation:
                data.summation = temporary
                data.left = i
                data.right = j
    data.end_time = time()  # -------------------------------------------------------------------- END TIME
    return
```

ΑΛΓΟΡΙΘΜΟΣ ΜΕ ΠΟΛΥΠΛΟΚΟΤΗΤΑ O(n²)

```
@staticmethod
def max_subarray_on2(data: Data) -> NoReturn:
    data.description = "Method with O(n^2)"
    temporary_left, temporary_right, temporary_sum = 0, 0, 0
    data.start_time = time()  # ------------------------------------------------------------------------ START TIME
    for i in range(len(data.numbers)):
        if not temporary_sum and data.numbers[i] < 0:
            continue  # ------------------------------- WITHOUT THIS, WE WOULD HAVE (1+2+3+...+n) TOTAL ITERATIONS
        temporary_left = i
        for j in range(i, len(data.numbers)):
            data.iterations += 1
            temporary_sum += data.numbers[j]
            temporary_right = j
            if temporary_sum > data.summation:
                data.summation, data.left, data.right = temporary_sum, temporary_left, temporary_right
        temporary_sum = 0
    data.end_time = time()  # -------------------------------------------------------------------------- END TIME
    return
```

ΝΕΟΣ ΑΛΓΟΡΙΘΜΟΣ ΜΕ ΠΟΛΥΠΛΟΚΟΤΗΤΑ O(n²)

```
@staticmethod
def my_method(data: Data) -> NoReturn:
    data.description = "Another method with O(n^2)"
    data.start_time = time()  # ------------------------------------------------------------------------ START TIME
    only_negatives: bool = True
    for number in data.numbers:
        if number > 0: only_negatives = False

    for left in range(len(data.numbers)):  # ------------------------------------------------------------ OUTER LOOP
        if not only_negatives and data.numbers[left] < 0: continue
        subarray: list[int] = [data.numbers[left]]
        for right in range(left, len(data.numbers)):  # ---------------------------------------------- INNER LOOP
            data.iterations += 1
            if left != right: subarray.append(data.numbers[right])
            if sum(subarray) > data.summation: data.left, data.right, data.summation = left, right, sum(subarray)
    data.end_time = time()  # -------------------------------------------------------------------------- END TIME
    return
```

Ο συγκεκριμένος δεν έχει ακριβώς πολυπλοκότητα O(n²), καθώς στην αρχή εκτελείται ένα ακόμα loop για τα στοιχεία του πίνακα.

ΑΛΓΟΡΙΘΜΟΣ ΜΕ ΠΟΛΥΠΛΟΚΟΤΗΤΑ O(n)

```
@staticmethod
def max_subarray_on(data: Data) -> NoReturn:  # ------------------------------------------------ O(n) COMPLEXITY
    data.description = "Method with O(n) [Kadane]"
    current_max: int = data.numbers[0]
    temporary_left: int = 0
    data.start_time = time()  # ------------------------------------------------------------------------ START TIME

    for index in range(1, len(data.numbers)):
        if current_max < 0:
            current_max = data.numbers[index]
            temporary_left = index
        else:
            current_max += data.numbers[index]
        temporary_right = index
        if current_max > data.summation:
            data.summation = current_max
            data.left = temporary_left
            data.right = temporary_right

    data.end_time = time()  # -------------------------------------------------------------------------- END TIME
    return
```

ΑΛΓΟΡΙΘΜΟΣ ΜΕ ΠΟΛΥΠΛΟΚΟΤΗΤΑ O(n*log(n))

```
@staticmethod
def max_subarray_recursive(data: Data) -> NoReturn:
    data.description = "Method with O(n^3)"
    data.start_time = time()  # ------------------------------------------------------------------------------ START TIME

    def findMaximumSum(numbers: list[int], left: Optional[int] = None, right: Optional[int] = None) -> int:
        if not numbers: return 0
        if left is None and right is None: left, right = 0, len(numbers) - 1
        if right == left: return numbers[left]
        middle = (left + right) // 2

        left_maximum, total = -Data.MAX_ABS_VALUE, 0
        for index in range(middle, left - 1, -1):
            total += numbers[index]
            if total > left_maximum: left_maximum = total

        right_maximum, total = -Data.MAX_ABS_VALUE, 0
        for index in range(middle + 1, right + 1):
            total += numbers[index]
            if total > right_maximum: right_maximum = total

        maxLeftRight = max(findMaximumSum(numbers, left, middle), findMaximumSum(numbers, middle + 1, right))
        return max(maxLeftRight, left_maximum + right_maximum)
    data.summation = findMaximumSum(data.numbers)
    data.left, data.right = Data.INVALID_INDEX, Data.INVALID_INDEX
    data.end_time = time()  # -------------------------------------------------------------------------------- END TIME
    return
```

ΔΟΚΙΜΗ ΑΛΓΟΡΙΘΜΩΝ ΣΤΟ ΙΔΙΟ ΣΕΤ ΔΕΔΟΜΕΝΩΝ

```
from random import seed

from assets import Algorithms, Data

seed(1083879)

if __name__ == "__main__":

    for data in (Data((amount + 1) * 100) for amount in range(5)):
        print(repr(data))
        for algorithm in iter(Algorithms()):
            algorithm(data)
            print(data)
            data.reset()
```

Δοκιμάζουμε όλους τους αλγόριθμους που φτιάξαμε με τη σειρά, στο ίδιο σετ δεδομένων, με 100, 200, 300, 400 και 500 ακεραίους. Πρώτη παρατήρηση είναι η διαφορά στον χρόνο εκτέλεσης, κάτι που φαίνεται έντονα όσο μεγαλώνει το πλήθος των ακεραίων. Δεύτερη παρατήρηση είναι η αδυναμία του αναδρομικού O(n log(n)) αλγορίθμου να υπολογίσει τα indices του ζητούμενου subarray και τα συνολικά iterations.

Ήταν αναμενόμενο η πολυπλοκότητα του καθενός και τα συνολικά loop iterations που απαιτούνται για την εκτέλεσή του, να έχουν σαν αποτέλεσμα τους διαφορετικούς χρόνους. Είναι προφανές ότι ο πλέον αποδοτικός είναι αυτός της μικρότερης πολυπλοκότητας, δηλαδή η μέθοδος του Kadane.

ΟΡΙΑ ΑΛΓΟΡΙΘΜΩΝ ΣΤΟ ΔΕΥΤΕΡΟΛΕΠΤΟ

Θα προσπαθήσουμε να προσδιορίσουμε στο περίπου τι μεγέθους πίνακα δεδομένων μπορεί να χειριστεί η κάθε μέθοδος σε χρόνο ενός δευτερολέπτου. Σε ένα νέο script, κάνουμε import τις δύο κλάσεις και δημιουργούμε ένα νέο for loop που θα δημιουργεί κάθε φορά μεγαλύτερα σετ δεδομένων και θα δοκιμάζει σε αυτά κάποια μέθοδο. Στο παρακάτω παράδειγμα, θα δοκιμαστεί η μέθοδος πολυπλοκότητας O(n). Όπως φαίνεται, ο πρώτος πίνακας δεδομένων θα έχει 5 500 000 ακεραίους, ενώ το μέγεθός του θα αυξάνεται με σταθερό βήμα. Όταν ο χρόνος εκτέλεσης υπερβεί το ένα δευτερόλεπτο, το loop θα διακοπεί και θα ξέρουμε ποιο είναι το όριό του.

```python
from assets import Algorithms, Data


if __name__ == "__main__":

    for data in (Data(amount * 250 + 5_500_000) for amount in range(51)):
        print(repr(data))
        Algorithms.max_subarray_on(data)
        print(data)
        if data.duration > 1: break
        data.reset()
```

Επαναλαμβάνοντας την ίδια διαδικασία για κάθε μέθοδο, βγάζουμε τα παρακάτω (χονδρικά) αποτελέσματα: Η μέθοδος O(n) ξεπέρασε το ένα δευτερόλεπτο με πίνακα 5 500 750 ακεραίων, ενώ έγιναν τουλάχιστον 5000 iterations ανά millisecond. Η μέθοδος $O(n^2)$ ξεπέρασε το ένα δευτερόλεπτο με πίνακα 5250 ακεραίων, ενώ η $O(n^3)$ το ξεπέρασε με πίνακα 380 ακεραίων.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Python documentation: https://docs.python.org/3.9/

Python random module: https://docs.python.org/3.9/library/random.html

Python time module: https://docs.python.org/3.9/library/time.html

Python typing module: https://docs.python.org/3.9/library/typing.html

ΣΗΜΕΙΩΣΕΙΣ

Τα code snippets επικολλήθηκαν επίτηδες με διατήρηση της αρχικής μορφοποίησής τους για να αναδειχθεί η καλαισθησία των Theme Plugins του PyCharm, καθώς και η monospace γραμματοσειρά-στολίδι της JetBrains.