Maximum Sub-array Πρόβλημα

- **Ορισμός του προβλήματος:**

Πρόκειται για ένα πρόβλημα όπου έχουμε μια σειρά (πίνακα) από αριθμούς (θετικούς, αρνητικούς ή μηδενικούς) και ψάχνουμε τη συνεχή υποσειρά (υποπίνακα) αριθμών που δίνει το μεγαλύτερο άθροισμα.

Το παραπάνω πρόβλημα, κλήθηκε να λυθεί με 4 αλγορίθμους διαφορετικής πολυπλοκότητας. Συγκεκριμένα, χρονικής πολυπλοκότητας $O(n^3)$, $O(n^2)$, $O(n \log n)$ και $O(n)$. Αναλυτικότερα:

✓ **Αλγόριθμος πολυπλοκότητας $O(n^3)$:**

```
def maxSubArray_On3(ls):
    lsLen = len(ls)
    maxSum = 0
    (start, end) = (-1, -1)
    for i in range(lsLen):
        for j in range(i, lsLen):
            tempSum = 0
            for o in range(i, j + 1):
                tempSum += ls[o]
                if tempSum > maxSum:
                    maxSum = tempSum
                    (start, end) = (i, j)

    return maxSum, start, end;
```

Ο παραπάνω αλγόριθμος **εξετάζει όλους τους δυνατούς συνδυασμούς υποπινάκων της λίστας εισόδου {brute force}** και επιστρέφει τον μέγιστο άθροισμα και τις αντίστοιχες θέσεις. Παράλληλα, πρόκειται για **πολυπλοκότητα $O(n^3)$ λόγω των τριών εμφωλισμένων βρόχων**.

✓ **Αλγόριθμος πολυπλοκότητας $O(n^2)$:**

```
def maxSubArray_On2(ls):
    lsLen = len(ls)
    maxSum = 0
    (start, end) = (-1, -1)
    for i in range(lsLen):
        tempSum = 0
        for j in range(i, lsLen):
            tempSum += ls[j]
            if tempSum > maxSum:
                maxSum = tempSum
                (start, end) = (i, j)

    return maxSum, start, end;
```

Ο παραπάνω αλγόριθμος **εξετάζει όλους τους δυνατούς συνδυασμούς υποπινάκων του δοσμένου πίνακα**. Για κάθε υποπίνακα, υπολογίζει το άθροισμα των στοιχείων του και αν αυτό είναι μεγαλύτερο από το τρέχον μέγιστο άθροισμα, τότε ενημερώνει το μέγιστο άθροισμα και τις αντίστοιχες θέσεις. Στο τέλος, επιστρέφει το μέγιστο άθροισμα και τις θέσεις του μέγιστου υποπίνακα. Παράλληλα, πρόκειται για **πολυπλοκότητα $O(n^2)$ λόγω των 2 εμφωλισμένων βρόχων**.

✓ **Αλγόριθμος πολυπλοκότητας O(n log n):**

```
def maxSubArray_Onlogn(ls):
    return maxSubArray_Onlogn_innerWorkingFunction1(ls, 0, len(ls) - 1)

def maxSubArray_Onlogn_innerWorkingFunction1(ls, lsStartPoint, lsEndPoint):
    #<<Διαίρει και βασίλευε>>
    if lsStartPoint == lsEndPoint:
        return ls[lsStartPoint], lsStartPoint, lsEndPoint;

    lsMidPoint = (lsStartPoint + lsEndPoint)//2
    resultFirstHalf = maxSubArray_Onlogn_innerWorkingFunction1(ls, lsStartPoint, lsMidPoint)
    resultSecondHalf = maxSubArray_Onlogn_innerWorkingFunction1(ls, lsMidPoint + 1, lsEndPoint)
    resultCombo = maxSubArray_Onlogn_innerWorkingFunction2(ls, lsStartPoint, lsMidPoint,
lsEndPoint)

    if resultFirstHalf[0] >= resultSecondHalf[0] and resultFirstHalf[0] >= resultCombo[0]:
        result = resultFirstHalf
    elif resultSecondHalf[0] >= resultFirstHalf[0] and resultSecondHalf[0] >= resultCombo[0]:
        result = resultSecondHalf
    else:
        result = resultCombo

    return result;

def maxSubArray_Onlogn_innerWorkingFunction2(ls, lsStartPoint, lsMidPoint, lsEndPoint):
    tempSum = 0
    (maxSumLeft, maxSumRight) = (0, 0)
    (startIndex, endIndex) = (-1, -1)
    for i in range(lsMidPoint, lsStartPoint - 1, -1):
        tempSum += ls[i]
        if tempSum > maxSumLeft:
            maxSumLeft = tempSum
            startIndex = i

    tempSum = 0
    for i in range(lsMidPoint + 1, lsEndPoint + 1):
        tempSum += ls[i]
        if tempSum > maxSumRight:
            maxSumRight = tempSum
            endIndex = i

    maxComboSum = (maxSumLeft + maxSumRight, startIndex, endIndex)
    maxSumLeft = (maxSumLeft, startIndex, lsMidPoint)
    maxSumRight = (maxSumRight, lsMidPoint + 1, endIndex)
    if maxSumLeft[0] >= maxComboSum[0] and maxSumLeft[0] >= maxSumRight[0]:
        result = maxSumLeft
    elif maxComboSum[0] >= maxSumLeft[0] and maxComboSum[0] >= maxSumRight[0]:
        result = maxComboSum
    else:
        result = maxSumRight

    return result;
```

Ο παραπάνω κώδικας κάνει **χρήση της τεχνικής "Διαίρει και Βασίλευε" (Divide and Conquer)** προκυμμένου να λύσει το πρόβλημα. Αυτή η τεχνική χωρίζει τον αρχικό πίνακα σε 2 υποπίνακες και επιλύει το πρόβλημα για κάθε υποπίνακα ξεχωριστά. Στη συνέχεια, συνδυάζει τα αποτελέσματα των 2 υποπινάκων για να βρει την τελική λύση! Παράλληλα, πρόκειται για **πολυπλοκότητα O(n log n)**. Συγκεκριμένα:

1. **Διαίρει**: Ο αλγόριθμος διαιρεί τον πίνακα στη μέση, δημιουργώντας 2 υποπίνακες. Αυτό γίνεται σε σταθερό χρόνο {O(1)}.
2. **Βασίλευε**: Ο αλγόριθμος καλείται αναδρομικά για κάθε υποπίνακα. Κάθε αναδρομική κλήση έχει μισό το μέγεθος του αρχικού πίνακα, οπότε ο συνολικός χρόνος είναι 2T(n/2).
   {T(n): ο χρόνος εκτέλεσης του αλγορίθμου}
3. **Συνδυασμός**: Ο αλγόριθμος υπολογίζει το μέγιστο άθροισμα που διασχίζει τη μέση του πίνακα. Αυτό γίνεται σε γραμμικό χρόνο {O(n)}.

Επομένως, **ο χρόνος εκτέλεσης του αλγορίθμου είναι T(n) = 2T(n/2) + O(n)**. Αυτή είναι μια αναδρομική σχέση που λύνεται σε **O(n log n) με τη χρήση του Θεωρήματος του Μαστερ**.

{σελ. 310 – Παράγραφος: 11.1.1 | Κεφάλαιο 11: Διαίρει και Βασίλευε}

✓ **Αλγόριθμος πολυπλοκότητας O(n):**

```python
def maxSubArray_On(ls):
    #Kadane's Algorithm
    lsLen = len(ls)
    maxSum = 0
    tempSum = 0
    tempStart = 0
    (start, end) = (-1, -1)
    for i in range(lsLen):
        tempSum += ls[i]
        if tempSum < 0:
            tempSum = 0
            if i + 1 < lsLen and ls[i + 1] >= 0:
                tempStart = i + 1
        elif tempSum > maxSum:
            maxSum = tempSum
            (start, end) = (tempStart, i)

    return maxSum, start, end;
```

Ο παραπάνω αλγόριθμος **διατρέχει τον πίνακα από την αρχή ως το τέλος, αθροίζοντας τα στοιχεία**. Αν το τρέχον άθροισμα γίνει αρνητικό, τότε το μηδενίζει και ξεκινάει να υπολογίζει ένα νέο υποπίνακα. Αν το τρέχον άθροισμα είναι μεγαλύτερο από το μέγιστο άθροισμα που έχει βρει μέχρι τώρα, τότε ενημερώνει το μέγιστο άθροισμα και τις θέσεις του αντίστοιχου υποπίνακα. Στο τέλος, ο αλγόριθμος επιστρέφει το μέγιστο άθροισμα και τις θέσεις του μέγιστου υποπίνακα. Παράλληλα, πρόκειται για **πολυπλοκότητα O(n) λόγω του ενός βρόχου**!

- **Δημιουργία των δεδομένων δοκιμής:**

```python
from random import seed, randint as randomInt

def generateRandomArrayN(N):
    bias = 10
    tempList = [randomInt(-bias, bias) for i in range(N)]

    return tempList;
```

Τα μεγέθη των πινάκων που χρησιμοποιήθηκαν για την αξιολόγηση της απόδοσης κάθε αλγορίθμου, **επιλέχθηκαν βάση δοκιμών**!

✍ **Αποτελέσματα στα δεδομένα δοκιμής:**

Αλγόριθμος	Μέγεθος λίστας	Χρόνος εκτέλεσης (sec)
maxSubArray_On3	484	1.3737730979919434
	215	0.12013936042785645
	273	0.20717477798461914
maxSubArray_On2	2349	0.20168828964233398
	7764	2.137838363647461
	7673	2.0693042278289795
maxSubArray_Onlogn	532910	1.6360435485839844
	574114	1.774977207183838
	625470	1.930908203125
maxSubArray_On	10481150	1.04176926612854
	14048638	1.4035966396331787
	13310991	1.3568413257598877

Συμπεράσματα:

Παρατηρούμε ότι **τα πειραματικά δεδομένα συμφωνούν με τα θεωρητικά**! Αναλυτικότερα, ισχύουν οι παρακάτω τύποι:

O(n^3)

$$\frac{n_1^3}{t_1} = \frac{n_2^3}{t_2}$$

όπου n_i*: το μέγεθος της λίστας &*
t_i*: ο χρόνος εκτέλεσης/διεκπεραίωσης του αλγορίθμου*

✍ **Π.χ.:**

$$\frac{484^3}{1.3737730979919434} = \frac{273^3}{t_2} \Longrightarrow t_2 \approx 0.246528\text{sec}$$

Ομοίως:

O(n^2)

$$\frac{n_1^2}{t_1} = \frac{n_2^2}{t_2}$$

O(n)

$$\frac{n_1}{t_1} = \frac{n_2}{t_2}$$

Όσων αφορά τον **αλγόριθμο πολυπλοκότητας O(n log n)**, **δεν υπάρχει κάποιος ανάλογος τύπος** όπως οι παραπάνω. Ωστόσο, είναι γνωστό ότι **ο χρόνος εκτέλεσης αυτού του αλγορίθμου θα κυμαίνεται μεταξύ του χρόνου εκτέλεσης του αλγορίθμου O(n^2) και O(n), για προβλήματα ίδιου μεγέθους**!

- **Απάντηση στο 2ο ερώτημα/σκέλος της άσκησης:**

Βάση των παραπάνω τύπων, συμπεραίνουμε ότι ισχύει:

Αλγόριθμος	Μέγεθος λίστας	Χρόνος εκτέλεσης (sec)
maxSubArray_On3	627	2.864130735397339
maxSubArray_On2	9238	3.179847478866577
maxSubArray_Onlogn	949560	2.9377198219299316
maxSubArray_On	30182739	3.1576552391052246

Οι υπολογισμοί των μεγεθών της λίστας εισόδου για χρόνο εκτέλεσης κοντά στα 3sec έγιναν κατά την εκτέλεση του προγράμματος, βάση των παραπάνω τύπων και της ιδικής περίπτωσης της O(n log n)!