Εργασία 1 Αλγορίθμων

ΓΕΩΡΓΙΟΥ ΑΛΕΞΙΟΣ – ΛΑΖΑΡΟΣ

Ρ3180027

ΙΟΥΝΙΟΣ 2020

1.1)

Με βάση το θεώρημα των ορίων μπορούμε να βρούμε ποιος συμβολισμός εκφράζει τη σχέση των δυο συναρτήσεων f και g, f = O(g) δηλαδή τι φράγμα αποτελεί η g για την f ασυμπτωτικά.

α. $\lim_{n\to\infty} \frac{f}{g} = \lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{n^{\frac{3}{4}}} = \lim_{n\to\infty} \frac{n^{\frac{2}{4}}}{n^{\frac{3}{4}}} = \lim_{n\to\infty} n^{-\frac{1}{4}} = \lim_{n\to\infty} \frac{1}{n^{\frac{1}{4}}} = 0$

Άρα $f \in O(g)$

β. $\lim_{n\to\infty} \frac{f}{g} = \lim_{n\to\infty} \frac{100n+\log(n)}{n+(\log(n))^2} = \lim_{n\to\infty} \frac{100n+\ln(n)}{n+(\ln(n))^2} \underset{DLH}{\Longrightarrow} \lim_{n\to\infty} \frac{100+\frac{1}{n}}{1+\frac{2\ln(n)}{n}} =$

$\lim_{n\to\infty} \frac{\frac{100n+1}{n}}{\frac{n+2\ln(n)}{n}} = \lim_{n\to\infty} \frac{100n+1}{n+2\ln(n)} \underset{DLH}{\Longrightarrow} \lim_{n\to\infty} \frac{100}{1+\frac{2}{n}} = 100$

Άρα $f \in \Theta(g)$

γ. $\lim_{n\to\infty} \frac{f}{g} = \lim_{n\to\infty} \frac{n\cdot 2^n}{3^n} = \lim_{n\to\infty} n \cdot \left(\frac{2}{3}\right)^n = \lim_{n\to\infty} \frac{n}{\left(\frac{3}{2}\right)^n} \underset{DLH}{\Longrightarrow} \lim_{n\to\infty} \frac{1}{\frac{(ln(3)-ln(2))\cdot 3^n}{2^n}} = 0$

Άρα $f \in O(g)$

δ. Προσεγγιστικά τα αθροίσματα με την χρήση του θεωρήματος με ολοκληρώματα είναι:

Για την $f(n) = \sum_{i=1}^{n} i^k$ και $k \geq 1$ έχουμε $f \uparrow$

$$\int_0^n i^k di \leq f(n) \leq \int_1^{n+1} i^k di \Rightarrow \frac{n^{k+1}}{k+1} \leq f(n) \leq \frac{(n+1)^{k+1}}{k+1} - \frac{1}{k+1}$$

Άρα $f(n) \in \Theta(n^{k+1})$

Το ίδιο βγάζουμε και για $k < 1$ ($f \downarrow$)

Για την $g(n) = \sum_{i=1}^{n} i2^i$ έχουμε $g \uparrow$

$$\int_0^n i2^i di \leq g(n) \leq \int_1^{n+1} i2^i di \Rightarrow \textit{ομοιώς με τη λύση του ολοκληρώματος βγαίνει}$$

$$g(n) \in \Theta(2^{n+1} \cdot (n+1))$$

Άρα $f \in \Theta(g)$ αφού $f \in O(g)$ $\textit{και}$ $f \in \Omega(g)$

ε. Παρατηρούμε ότι η $f(n) = 2^{n-1}$ δηλαδή $f = \Theta(2^n)$ και $g = \Theta(2^n)$ άρα $f \in \Theta(g)$

Από διωνυμικό θεώρημα έχουμε $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$

που αποδεικνύει την f $\textit{για}$ $x, y = 1$.

στ. Με βάση τον ορισμό $f \in \Omega(g)$ αφού για

c = 1 και $n_0 = 5, f(n) \geq c \cdot g(n) \geq 0$ $\textit{για κάθε}$ $n \geq n_0$

δηλαδή $n! \geq 2^n \geq 0$ $\textit{για κάθε}$ $n \geq 5$ άρα η g είναι κάτω ασυμπτωτικό φράγμα της f.

Η εξίσωση αποδεικνύεται επαγωγικά,

για $n = 5$ $\textit{ισχύει}$ $5! \geq 2^5 \Leftrightarrow 120 \geq 32$

Έστω ότι ισχύει για n, θα αποδείξουμε ότι ισχύει για $n+1$ και άρα θα ισχύει για όλα τα $n \geq 5$

$(n+1)! \geq 2^{n+1} \Leftrightarrow (n+1) \cdot n! \geq 2 \cdot 2^n \Leftrightarrow$

Που ισχύει γιατί $n + 1 > 2$ $\textit{για}$ $n \geq 5$ $\textit{και}$ $n! \geq 2^n$ όπως υποθέσαμε άρα ισχύει για όλα τα n.

ζ. Με βάση τον ορισμό $f \in O(g)$ αφού για

c = 1 και $n_0 = 1, 0 \leq f(n) \leq c \cdot g(n)$ $\textit{για κάθε}$ $n \geq n_0$

δηλαδή $0 \leq n! \leq n^n$ $\textit{για κάθε}$ $n \geq 1$ άρα η g είναι άνω ασυμπτωτικό φράγμα της f.

Η εξίσωση αποδεικνύεται επαγωγικά,

για $n = 1$ $\textit{ισχύει}$ $1! \leq 1^1 \Leftrightarrow 1 \leq 1$

Έστω ότι η ανίσωση ισχύει $0 \leq n! \leq n^n$ για $n \geq 1$, θα αποδείξουμε ότι ισχύει για $n+1$ και άρα θα ισχύει για όλα τα $n \geq 1$

$(n+1)! \leq (n+1)^{n+1} \Leftrightarrow (n+1) \cdot n! \leq (n+1) \cdot (n+1)^n \Leftrightarrow n! \leq (n+1)^n$

Που ισχύει γιατί από υπόθεση ισχύει $n! \leq n^n$ και $n^n \leq (n+1)^n$ για $n \geq 1$

1.2)

α. Υποθέτω ότι έχω έτοιμους του συνδυασμούς και μπορώ να τους πάρω σε $\Theta(1)$. Το σύνολο των συνδυασμών με k στοιχεία στο σύνολο S με n στοιχεία είναι $\binom{n}{k}$ και συνδυασμό πρέπει να προσθέσω k αριθμούς και να ελέγξω αν είναι ίσοι με το M. Άρα συνολικά θα κάνω περίπου $\binom{n}{k} * \boldsymbol{k}$ προσθέσεις (αν εξαιρέσουμε και τις συγκρίσεις με το M.

Άρα έχουμε πολυπλοκότητα: $\mathbf{O}(\binom{n}{k} * \boldsymbol{k})$

β. Πρέπει να κατασκευάσουμε όλους τους συνδυασμούς του συνόλου, δηλαδή όλους του συνδυασμούς με κ στοιχεία για κ από 1 μέχρι n (τον αριθμό στοιχείων του συνόλου). Γνωστό ως και δυναμοσύνολο το οποίο έχει 2^{n} στοιχεία. Άρα ο αλγόριθμος του προβλήματος είναι $\mathbf{O}(\boldsymbol{2^n})$.

1.3)

α. Έχουμε $\alpha = 4, \beta = 4, f(n) = 7n$. Από Master Theorem $T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 7n$

$$n^{\log_b a} = n^{\log_4 4} = n$$

$$Ισχύει\ f(n) = 7n = \Theta(n)\ για\ \varepsilon > 0\ άρα\ \boldsymbol{T(n) = \Theta(nlogn)}$$

β. Έχουμε $\alpha = 4, \beta = 16, f(n) = n^{\frac{5}{4}}$. Από Master Theorem $T(n) = 4 \cdot T\left(\frac{n}{16}\right) + n^{\frac{5}{4}}$

$$n^{\log_b a} = n^{\log_{16} 4} = n^{0.5}$$

$$Ισχύει\ f(n) = n^{\frac{5}{4}} = \Omega(n^{0.5-\varepsilon})\ για\ \varepsilon > 0\ και$$

$$af\left(\frac{n}{b}\right) \leq c \cdot f(n)\ για\ c < 1 \Longrightarrow 4f\left(\frac{n}{16}\right) \leq cf(n) \Longrightarrow 4\left(\frac{n}{16}\right)^{\frac{5}{4}} \leq cn^{\frac{5}{4}}$$

$$άρα\ \boldsymbol{T(n) = \Theta(n^{\frac{5}{4}})}$$

γ. Έχουμε $\alpha = 8, \beta = 4, f(n) = 6\sqrt{n}$. Από Master Theorem $T(n) = 8 \cdot T\left(\frac{n}{4}\right) + 6\sqrt{n}$

$$n^{\log_b a} = n^{\log_4 8} = n^{1.5}$$

$$Ισχύει\ f(n) = 6\sqrt{n} = O(n^{1.5-\varepsilon})\ για\ \varepsilon > 0\ άρα\ \boldsymbol{T(n) = \Theta(n^{1.5})}$$

δ. Έχουμε μια αναδρομική εξίσωση $T(n) = 2T(n-2) + c, όπου\ c\ σταθερός\ χρόνος\ \Theta(1)$

Υποθέτουμε ότι χωρίς βλάβη της γενικότητας το n είναι άρτιος και γνωρίζουμε ότι $T(0) = 1$.

Με μέθοδο επανάληψης:

Τα πρώτα βήματα της αναδρομής:

$T(n) = 2T(n-2) + c \ [1] \quad T(n-2) = 2T(n-4) + c \ [2] \quad T(n-4) = 2T(n-6) + c \ [3]$

Άρα με χρήση των παραπάνω εξισώσεων και αντικατάσταση στα εκάστοτε βήματα βγάζουμε

$T(n) = 2T(n-2) + c = 2(2T(n-4) + c) + c = 4T(n-4) + 2c + c = 4(2T(n-6) + c) + 2c + c = 8T(n-6) + 4c + 2c + c = \cdots$

Στο βήμα κ θα έχουμε:

$$2^{\kappa}T(n-2k) + 2^{\kappa-1} \cdot c + 2^{\kappa-2} \cdot c + \cdots + c$$

Σε ποιο βήμα τελειώνει ο αλγόριθμος;

Όταν το $T(n-2k) = T(0) = 1$ δηλαδή για $n - 2k = 0 \implies k = \frac{n}{2}$

Άρα έχουμε για $k = \frac{n}{2}$:

$$T(n) = 2^{\frac{n}{2}}T(0) + 2^{\frac{n}{2}-1} \cdot c + 2^{\frac{n}{2}-2} \cdot c + \cdots + c = 2^{\frac{n}{2}} + c \cdot (2^{\frac{n}{2}-1} + 2^{\frac{n}{2}-2} + \cdots + 2^0)$$

$$= 2^{\frac{n}{2}} + c \cdot \sum_{i=0}^{i=\frac{n}{2}-1} 2^i = 2^{\frac{n}{2}} + c \cdot \frac{2^{\frac{n}{2}-1+1} - 1}{1} = 2^{\frac{n}{2}} + c \cdot \left(2^{\frac{n}{2}} - 1\right)$$

$$= 2^{\frac{n}{2}}(1 + c) - \ c = \ \boldsymbol{\Theta}(\mathbf{2}^{\frac{\boldsymbol{n}}{\mathbf{2}}})$$

Για να επιλέξουμε αλγόριθμο για την επίλυση του προβλήματος επιλέγουμε τον αλγόριθμο με την μικρότερη πολυπλοκότητα, δηλαδή τον αλγόριθμο A_1 με πολυπλοκότητα $\Theta(nlogn)$

Η σύγκριση έγινε με “εμπειρικό” κανόνα αφού έχουν αποδειχτεί όμοιες συγκρίσεις στην πρώτη άσκηση.

1.4) Ο αλγόριθμος είναι παρόμοιος με τον αλγόριθμο της Mergesort, μόνο που τώρα πριν ταξινομήσουμε κάθε υποπίνακα θα προσαρμόσουμε τον αλγόριθμο να μετράει και τις αντιστροφές.

Έχουμε δυο συναρτήσεις:

Η Mergesort η οποία καλεί τον εαυτό της για το αριστερό και δεξί υποπίνακα και στο τέλος την Merge η οποία τους ενώνει ταξινομημένα. Παράλληλα όμως οι συναρτήσεις εκτός από την ταξινόμηση μετράνε και τις αντιστροφές που βρήκανε και στο τέλος έχουμε το συνολικό αριθμό αντιστροφών.

Η Merge δέχεται τον πίνακα ο οποίος περιέχει και τους δυο υποπίνακες ταξινομημένους ως προς τον εαυτό τους, δημιουργεί τους δυο υποπίνακες και κάποια position counters για να τα χρησιμοποιήσει όταν συμπληρώνει τον τελικό πίνακα ταξινομημένο. Οπότε ξεκινάει η διαδικασία που ελέγχουμε τα στοιχεία των δυο υποπινάκων και βάζουμε το κατάλληλο στον τελικό πίνακα. Αν βρεθεί κάποιο από τον δεξιά υποπίνακα σημαίνει ότι υπάρχουν αντιστροφές

στην διαδικασία και είναι ίσες με το position του πρώτου αριθμού του δεξιού υποπίνακα πλην τον αριθμό του μετρητή του αριστερού υποπίνακα (δηλαδή υπάρχουνε αντιστροφές ίσες για τα αντικείμενα του αριστερού πίνακα πλην από αυτά που έχουμε βάλει ήδη στον τελικό πίνακα)

Αυτό συμβαίνει εκμεταλλευόμενοι το γεγονός ότι οι δυο υποπίνακες είναι ταξινομημένοι.

Η πολυπλοκότητα είναι η ίδια με της Mergesort αφού οι αλλαγές που έχουμε εφαρμόσει στον κώδικα είναι όλες Θ(1) (συγκρίσεις, μετρητές, αριθμητικές πράξεις). Οπότε η τελική πολυπλοκότητα είναι ίση με $\boldsymbol{O(nlogn)}$**.** Πιο αναλυτικά, η συνάρτηση Merge τρέχει σε $\Theta(n)$ πολυπλοκότητα αφού πρέπει να ενώσει n αντικείμενα συνολικά, επίσης κάθε πρόβλημα αναλύεται σε 2 ίδιου μεγέθους υποπροβλήματα και ενώνονται σε $\Theta(n)$ $όπως\ είδαμε$.

Άρα $T(n) = 2T\left(\frac{n}{2}\right) + n$, έχουμε $n^{\log_b a} = n^{\log_2 2} = n$ άρα από Master Theorem έχουμε

$f(n) = n = \Theta\left(n^{\log_b a}\right) = \Theta(n)$ άρα $T(n) = \ \Theta(nlogn) = \boldsymbol{O(nlogn)}$

Σχεδίαση Αλγόριθμου (ψευδοκώδικας):

```
Mergesort function(array, l, r){
    count = 0
    count += MergeSort(count of inversion in left half of the array)
// this will also sort the array after counting
    count += MergeSort(count of inversion right half of the array)
// this will also sort the array after counting
    count += Merge(count of inversion that occurs in merging the two halves)
    stop when position bounds are wrong
    return count
    }

Merge function(array, l, r){

//we have one array which the left and right subarray is sorted independently
so far
    make a leftarray and rightarray subarray
    leftpos, rightpos, mainpos, count = 0
    while leftpos < leftarray.length and rightpos < rightarray.length){
        if leftarray[leftpos] <= rightarray[rightpos]{
            put leftarray item in main array and increase the suitable pos
counters
        }else{
            put rightarray item in main array and increase the suitable pos
counters
            //If we are here it means we found reverse pairs
            //If an item from right array needs to be added in the main array
at this point,
            //it means that there are items from leftarray that are bigger than
the item.
            //So we will have inversions equal to middle index (or first item
from right array index) minus the last leftpos we had
            counter += middlepos - leftpos
        }
    }
    complete the main array with potential remaining items from either left or
right subarray
    return counter
}
```

1.5)

```
//Προεπεξεργασία
Ο αλγόριθμος δέχεται n διάστημα της μορφής [Si, Fi]
Ταξινομώ τα διαστήματα έτσι ώστε να ισχύει για f1< f2 < ... <fi
X = ταξινομημένη λίστα διαστημάτων // O(nlogn)

//Άπληστο βήμα
Επιλεγώ για κάθε διάστημα i το fi και το βάζω
στην λίστα των τελικών σημείων μόνο αν το διάστημα δεν έχει καλυφθεί
από τα προηγούμενα f που έχει η τελική λίστα

Επιστροφή του μήκους της λίστας τελικών σημείων


C = []
for i in X: // O(n) θα τρέξει για όλα τα διαστήματα
    Fineeded = true
    for acceptedF in C: // O(n) χειρότερη περίπτωση ου έχουμε ξένα
μεταξύ τους διαστήματα και θα έχουμε n AcceptedF
        if acceptedF >= Si AND acceptedF <= Fi //Αν βρεις F που να
καλύπτει το διάστημα
            Fineeded = false //δεν θα βάλουμε το καινούργιο F
    if Fineeded == true
        C += fi // Άμα δεν βρήκαμε κάτι που να καλύπτει το διάστημα,
τότε βάζουμε το F του διαστήματος
return C.length
```

Πολυπλοκότητα:

Έχουμε Προεπεξεργασία (ταξινόμηση των διαστημάτων με Mergesort) πολυπλοκότητα $O(nlogn)$.

Και ο άπληστος αλγόριθμος τρέχει σε πολυπλοκότητα $O(n^2)$ αφού έχουμε n διαστήματα και πρέπει να ελέγχουμε το πολύ i αριθμούς της λίστας αποδοχής, όπου i ο αριθμός του διαστήματος (το πολύ n).

Άρα θα έχουμε το πολύ $\sum_{i=0}^{n} i$ βήματα συνολικά το οποίο είναι $O(n^2)$

Συνολικά ο αλγόριθμος τρέχει σε $O(n^2)$ αφού $O(nlogn) + O(n^2) = O(n^2)$

Απόδειξη ορθότητας με επιχείρημα ανταλλαγής:

Έστω ότι η greedy λύση δεν είναι βέλτιστη, και έστω ότι υπάρχει μια άλλη βέλτιστη λύση OPT. Η λύση greedy έχει επιλογές $t_1, t_2 \cdots, t_n$. Ενώ η OPT λύση έχει $s_1, s_2, \cdots, s_n$. Έστω τώρα δείκτης k για τον οποίο ισχύει πως οι δυο αυτές λύσεις είναι ίδιες μέχρι αυτόν, δηλαδή $t_i = s_i$ για i από 0 μέχρι k και $t_{k+1} \neq s_{k+1}$. Στη βέλτιστη λύση αλλάζουμε την s_{k+1} με την t_{k+1} και

παρατηρούμε πως η λύση OPT δεν χειροτερεύει καθώς η επιλογή t_{k+1}. έχει γίνει με το βέλτιστο τρόπο. Εφαρμόζοντας αυτό το επιχείρημα επαγωγικά για τις υπόλοιπες επιλογές προκύπτει ότι η λύση OPT είναι ίδια με τη λύση greedy, άρα η greedy είναι ορθή και βέλτιστη.

1.6)

Έχουμε ένα δυσδιάστατο πίνακα A[n, C]. Θα λύσουμε το πρόβλημα λύνοντας υποπροβλήματα του προβλήματος με δυναμικό προγραμματισμό. Τα στοιχεία του πίνακα θα παίρνουν 0-1 τιμές και όσο προχωράμε σε κάθε γραμμή θα κοιτάμε τις προηγούμενες αφού για να λυθεί ένα υποπρόβλημα χρειάζεται τη λύση ενός μικρότερου. Η λογική του αλγορίθμου είναι ότι για να δούμε άμα ένα αντικείμενο είναι μέρος του υποσυνόλου τότε πρέπει να υπάρχει συνδυασμός αντικειμένων στο υποσύνολο αντικειμένων χωρίς αυτό και για βάρος/χωρητικότητα C- Wi. Ο στόχος είναι να βρούμε το τελευταίο στοιχείο του πίνακα (A[n, C]) και να το επιστρέψουμε. Το πρόβλημα είναι σαν το 0-1 Knapsack με values = weight για κάθε αντικείμενο.

```
Αρχικοποιούμε τον πίνακα με μηδενικά (A[n,C]),
ο πίνακας είναι από i = 0 έως n και από j = 0 έως C,
όπου n το μέγεθος του πίνακα W

Για i από 1 έως n+1:
    Για j από 1 έως c+1:
        Άν w[i-1] <= C:
            A[i][j] = max(w[i-1] + A[i-1][j-w[i-1]], A[i-1][j])
        Αλλιώς:
            A[i][j] = A[i-1][j]
        Άν A[i][j] == C and j == C:
            return True
return False
```

Κατασκευάζουμε τον πίνακα του 0-1 knapsack με τον πίνακα weight να είναι και ο πίνακας value και στην πορεία άμα βρούμε κάποιο στοιχείο της τελευταίας στήλης (για j == C) το οποίο είναι και ίσο με το C, τότε το πρόβλημα έχει λύση και επιστρέφουμε True, αλλιώς επιστρέφουμε False.

Πολυπλοκότητα:

Έχουμε ένα πίνακα δυσδιάστατο με μέγεθος nxC και για κάθε στοιχείο του έχουμε Θ(1) πράξεις. Άρα η πολυπλοκότητα χρόνου (και χώρου) είναι O(nC). Όπου n το ο αριθμός των αντικειμένων και C το βάρος. Θα μπορούσαμε να εξετάσουμε αν υπάρχει μέγιστος κοινός διαιρέτης για τα W και το C ώστε να έχουμε μικρότερο C και πιθανόν μικρότερο πίνακα.

1.7)

```
//Διαδικασία

Επιλεγώ έναν τυχαίο κόμβο και βρισκω με ποιους κόμβους συνδέονται
Διαγραφώ τον κόμβο από το γράφημα και ελέγχω για το καινούργιο γράφημα
άμα υπάρχει κόμβος που HC-D == true και συνδεότανε με τον προηγούμενο
κόμβο
Επαναλαμβάνω την διαδικασία για τον καινούργιο κόμβο που βρήκα και κάθε
φορά που διαγραφώ έναν κόμβο από το γράφημα τον προσθέτω στην λύση

//Επιχείρημα βήματος
Αν έχω κάποιο κόμβο και βρω έναν γείτονα κόμβο για τον οποίο ισχύει ότι
υπάρχει
Hamiltonian κύκλος για το γράφημα χωρίς τον πρώτο κόμβο, σημαίνει ότι
μπορώ να περάσω από του υπόλοιπους κόμβους χωρίς να περάσω πάλι από τον
πρώτο, άρα η επιλογή του δευτέρου κόμβου είναι σωστή. Το επιχείρημα
είναι το ίδιο επαγωγικά και για τους υπόλοιπους κόμβους.
```

Πολυπλοκότητα:
Έχω το πολύ n κόμβους και για κάθε κόμβο πρέπει να βρω με πόσους συνδέεται, άρα το πολύ $n-1$ γείτονες. Τώρα για κάθε κόμβο πρέπει να εξετάσω για το πολύ $n-1$ γείτονες άμα υπάρχει Hamiltonian κύκλος (HC-D) στο γράφημα με $n, n-1, n-2 \ldots 1$ κόμβους το οποίο είναι πολυπλοκότητας n^k.
Το πολύ δηλαδή να έχω n κόμβους, n-1 γείτονες και γραφήματα με n κόμβους. Άρα ένα άνω όριο πολυπλοκότητας με πρώτη μάτια το οποίο δεν είναι τόσο αυστηρό είναι $\boldsymbol{O(n^{k+2})}$, δηλαδή αλγόριθμος πολυωνυμικού χρόνου.

1.8) Το πρόβλημα TSP για συγκεκριμένο b (έστω b ακέραιο) μου απαντάει στην ερώτηση αν μπορώ να φτιάξω μια λύση τέτοια ώστε η περιήγηση να στοιχίζει **το πολύ** b. Υποθέτουμε ότι για το αρχικό b στην TSP υπάρχει λύση (η οποία πιθανόν να μην είναι optimal), ώστε να υπάρχει ένα άνω όριο στο κόστος του προβλήματος.

Για την TSP-OPT μπορούμε να φανταστούμε ότι θεωρητικά υπάρχει ένας πίνακας A από i= 1 μέχρι i = b ο οποίος περιέχει τις τιμές True ή False αν για το συγκεκριμένο i το πρόβλημα λύνεται ή όχι. Παρατηρούμε ότι σε αυτόν τον πίνακα όλες οι τιμές False που πιθανών υπάρχουν θα είναι όλες στην αρχή και μετά από ένα σημείο θα είναι True. Με αυτήν την λογική ψάχνουμε το κατάλληλο i για το οποίο A[i] == True και A[i-1] == False. Το πρόβλημα αυτό μπορεί να λυθεί με προσαρμογή της δυαδικής αναζήτηση στον πίνακα με θέσεις b και πολυπλοκότητα $O(logn)$ και υπολογισμό σε κάθε βήμα το TSP του i.

Μόλις βρω το κατάλληλο i προφανώς επιστρέφω το TSP(i).

Πολυπλοκότητα:

Πρέπει να τρέξω τον TSP για κάθε στοιχείο του πίνακα που περνάω με την δυαδική αναζήτηση (και το αριστερό του). Αρά έχω περίπου $2logn$ φορές το TSP το οποίο είναι πολυπλοκότητας $O(n^k)$. Άρα η πολυπλοκότητα του TSP-OPT είναι $\boldsymbol{O(n^k logb)}$ δηλαδή πολυωνυμικός χρόνος.