**Δημήτρης Κοκκίνης 03118896**
**Νοέμβριος, 2024**

# Αλγόριθμοι και Πολυπλοκότητα
# 1η Σειρά Γραπτών Ασκήσεων

## Άσκηση 1: Υπολογισμός Κυρίαρχων Θέσεων

Η brute force λύση είναι για κάθε στοιχείο απο $i = 2$ μέχρι n, να εξετάζουμε ένα ένα τα στοιχεία στα αριστερά του, μέχρι να βρούμε κάποιο το οποίο είναι μεγαλύτερό του. Αυτό υλοποιείται με ένα φωλιασμένο βρόχο και έχει πολυπλοκότητα στη χειρότερη περίπτωση $\mathcal{O}(n^2)$

Ένας αποδοτικότερος τρόπος είναι να αξιοποιήσουμε μια στοίβα. Η ιδέα του αλγόριθμου, βασίζεται στην αρχή πως κάθε αριθμός στην ακολουθία απο την αρχή μέχρι το τέλος, είναι "πιθανή" κυρίαρχη θέση για τα επακόλουθα στοιχεία, μέχρι να βρεθεί κάποιος αριθμός ο οποίος είναι μεγαλύτερός του. Τότε παύει να είναι "πιθανή" κυρίαρχη θέση, διότι τα επακόλουθα στοιχεία θα έχουν τον νέο μεγαλύτερο αριθμό σε πλησιέστερη θέση. Θα αξιοποιήσουμε τη στοίβα ώστε να κρατάμε τις "πιθανές κυρίαρχες θέσεις".

$$[100, 70, 80, 50, 25, 30, 40, 53, 60, 66, 61, 77, 90]$$

Ο αλγόριθμος θα εξετάσει γραμμικά όλα τα στοιχεία απο $i = 2$ έως n. Κάθε στοιχείο, θα το συγκρίνει με το προηγούμενό του. Αν είναι μεγαλύτερο απο το προηγούμενό του, τότε προφανώς το προηγούμενό του δεν μπορεί να είναι κυρίαρχη θέση, οπότε θα ελέγχει στη στοίβα με τα "πιθανά κυρίαρχα στοιχεία". Θα βγάζει απο τη στοίβα ένα ένα τα στοιχεία μέχρι να βρει κάποιο το οποίο είναι μεγαλύτερο απο το ίδιο. Τότε θα του αναθέτει το στοιχείο αυτό ως τη κυρίαρχη θέση και ο αλγόριθμος θα συνεχίζει μέχρι να φτάσει στο τελευταίο στοιχείο. Αν, ωστόσο, το προηγούμενο στοιχείο $i - 1$ είναι μεγαλύτερο απο το $i$, αυτό σημαίνει ότι το $i - 1$ είναι κυρίαρχη θέση για το $i$, αλλά και για τα επακόλουθα, μέχρι να βρεθεί κάποιο μεγαλύτερο. Οπότε τοποθετούμε το στοιχείο της θέσης $i - 1$ στη στοίβα.

### Algorithm

1. Initialize a stack $s$ and push 0
2. Initialize an array $M[n]$, where we will keep the results, with all values set to zero
3. $i = 2$
4. **while** $i < n$

    **if** $A[i-1] > A[i]$

        $M[i] = i - 1$

        s.push($(i - 1)$)

    **if** $A[i-1] < A[i]$

        start comparing elements in the stack until a larger value

        **if** $A[s.top()] > A[i]$     *//since there is inf value there will always be an s.top() greater than*

$A[i]$

$M[i] = A[s.top()]$ *//store the index of the greatest*

**if** $A[s.top()] \leq A[i]$

s.pop()

5. return M

Ο αλγόριθμος βασίζεται στο γεγονός πως αν το εξεταζόμενο στοιχείο είναι μεγαλύτερο απο το προηγούμενό του, τότε θα το συγκρίνουμε με τη κυρίαρχη θέση του προηγούμενού του, κι αν είναι μεγαλύτερο κι απο αυτό, τότε η επόμενη πιθανή κυρίαρχη θέση είναι η κυρίαρχη θέση της κυρίαρχης θέσης του προηγούμενού του και ούτω καθεξής. Βάζοντας τα στοιχεία που είναι πιθανό να είναι κυρίαρχα στη στοίβα, γλυτώνουμε αρκετές συγκρίσεις. Στο τέλος, ο αλγόριθμος θα επιστρέφει τον πίνακα M στον οποίον έχουμε αποθηκεύσει τις κυρίαρχες θέσεις για όλα τα στοιχεία του πίνακα.

Η πολυπλοκότητα του αλγόριθμου είναι $\mathcal{O}(n)$ καθώς θα κάνει n επαναλήψεις.

## Άσκηση 2: Επιλογή

### (a)

Θα χρησιμοποιήσουμε binary search καλώντας τη συνάρτηση $Fs$ στο διάστημα $[0, M]$. Ξεκινώντας απο το M, ο αλγόριθμος θα καλεί την $Fs$ για το $mid$ του διαστήματος και θα συγκρίνει τη τιμή του με το ζητούμενο k. Αν το k είναι μεγαλύτερο τότε θα ψάχνουμε στο δεξί τμήμα, αλλιώς αν είναι μικρότερο θα ψάχνουμε στο αριστερό τμήμα.

**Algorithm** ( $Fs$, k )

1. Set low = 1 and high = M
2. **while** $low < up - 1$

   $mid = \frac{low+high}{2}$

   **if** $k \leq Fs(mid)$

   high = mid

   **else if** $k > Fs(mid)$

   low = mid
3. return up

Ο αλγόριθμος τερματίζει καθώς σε κάθε επανάληψη τα *left* και *right* έρχονται πιο κοντά. Για το upper bound του αλγόριθμου που είναι το high, ισχύει ότι πάντα $Fs(high) \geq k$ και για το lower bound που είναι το low ισχύει ότι πάντα $Fs(low) < k$. Στην επανάληψη όπου ο αλγόριθμος θα συγκλίνει στη ζητούμενη τιμή, το high θα λάβει τη τιμή mid και θα επιστρέψουμε το high.

Η πολυπλοκότητα του αλγόριθμου είναι $\mathcal{O}(logM)$, καθώς σε κάθε επανάληψη διαιρείται το μήκος του διαστήματος αναζήτησης στα 2

**(b)**

Τα στοιχεία του συνόλου S, δηλαδή οι πιθανές διαφορές όλων των στοιχείων του πίνακα είναι $\frac{n(n-1)}{2}$. Μια naive μέθοδος, είναι να υπολογίσουμε όλες τις πιθανές αυτές απόλυτες διαφορές και να τις αποθηκεύσουμε σε ένα πίνακα, τον οποίον στη συνέχεια θα ταξινομήσουμε για να βρούμε το k-οστό μικρότερο. Η μέθοδος αυτή θα έχει πολυπλοκότητα χειρότερης περίπτωσης $\mathcal{O}(n^2)$. Ένας καλύτερος τρόπος είναι να χρησιμοποιήσουμε δυαδική αναζήτηση.

Γενικά η εύρεση του k-οστού μικρότερου στοιχείου σε ένα πίνακα, έγκειται στην εύρεση του αριθμού με k μικρότερα στοιχεία στον πίνακα (συμπεριλαμβανομένου και του εαυτού του). Επομένως, μια γενική μεθοδολογία δυαδικής αναζήτησης για το συγκεκριμένο πρόβλημα είναι η εξής:

I. Ταξινόμησε τον πίνακα A

II. Ξεκινώντας απο τη μικρότερη και τη μέγιστη διαφορά ως low και high, όσο low < high

III. Θέσε το μέσον του διαστήματος $mid = \frac{low+high}{2}$

IV. Εάν *[ο αριθμός των διαφορών που είναι $\leq$ του μέσου]* είναι μικρότερος απο το k, τότε θέσε το low = mid+1 και επανάλαβε για το νέο διάστημα

V. Αλλιώς θέσε high = mid και επανάλαβε για το νέο διάστημα

VI. Επίστρεψε το low

Η μέγιστη διαφορά θα είναι $A[n-1] - A[0]$ και η ελάχιστη διαφορά $A[1] - A[0]$. Αυτό που μένει, είναι να βρούμε έναν αποδοτικό τρόπο να λαμβάνουμε τη πληροφορία για έναν αριθμό, έστω m, τον [*αριθμό των διαφορών που είναι $\leq$ του $m$*]. Η συνάρτηση $F$ λοιπόν, θα επιστρέφει για έναν δοσμένο αριθμό τη πληροφορία αυτή, και θα πρέπει ο τρόπος με τον οποίο θα λειτουργεί η συνάρτηση να είναι αποδοτικός. Έχοντας ταξινομήσει τον πίνακα A, βασιζόμαστε στην εξής αρχή:

*"Ο αριθμός των διαφορών που είναι $\leq$ ενός φυσικού m απο ένα στοιχείο i και μετά, θεωρώντας πως η αρίθμηση του πίνακα ξεκινάει απο το 0, είναι:*

$$\frac{(upper\_new - i)((upper\_new - i) + 1)}{2} - \frac{(upper\_prev - i)((upper\_prev - i) + 1)}{2},$$

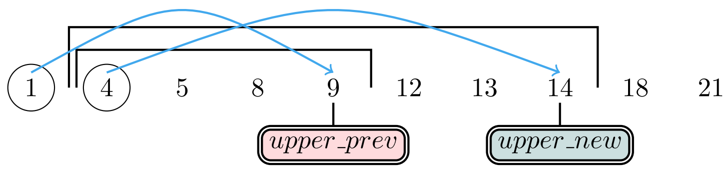
*όπου upper είναι το όριο για το οποίο ισχύει $A[upper] - A[i] \leq m$."*

"Οι δείκτες new και prev έχουν να κάνουν με το ότι θα πρέπει να αφαιρούμε και κάποιες ενδιάμεσες διαφορές που έχουν ξαναυπολογιστεί."

Για παράδειγμα, στον παρακάτω πίνακα, για είσοδο τον φυσικό αριθμό m=10, το upper για τον αριθμό 1 είναι ο αριθμός 9 και για τον αριθμό 4 ο αριθμός 14. Το πλήθος των διαφορών απο το 4 έως και το 14, χωρίς να λογαριάζουμε τις διαφορές έως και τον αριθμό 9 (τις οποίες υπολογίσαμε στο προηγούμενο $i = 0$), είναι:

$$count = \frac{(7-1)((7-1)+1)}{2} - \frac{(4-1)((4-1)+1)}{2} = 21 - 6 = 15$$

και είναι οι διαφορές 14-4, 13-4, 12-4, 14-5, 13-5, 12-5, 14-8, 13-8, 12-8, 14-9, 13-9, 12-9, 14-12, 13-12, 14-13




Αυτό ισχύει, καθώς αν η διαφορά ανάμεσα στο ακριανό στοιχείο με το i είναι μικρότερη ή ίση απο το ζητούμενο m, αυτό σημαίνει πως και οι ενδιάμεσες πιθανές διαφορές θα είναι μικρότερες, διότι η ακολουθία είναι σε αύξουσα σειρά. Έτσι, ο έλεγχος των διαφορών για το επόμενο στοιχείο μπορεί να ξεκινήσει απο το upper + 1 και έπειτα. Για κάθε στοιχείο θα ελέγχουμε τις διαφορές μέχρι να βρούμε το άνω όριο, ξεκινώντας απο το upper όριο που έθεσε ο προηγούμενος.

**Algorithm** F(m)

1. Initialize $total\_count = 0$, $upper\_prev = 0$, $upper\_new = 0$
2. $i = 0$, $j = 0$
3. **while** $upper\_new \leq n - 1$

    $j \mathrel{+}= 1$

    $diff = 0$, $count = 0$

    **if** $A[j] - A[i] \leq m$

        **break**

    **else if** $A[j] - A[i] > m$

        $j \mathrel{-}= 1$ $\quad$ *//since we did one extra iteration to check whether the difference is* $\leq m$

        $upper\_new = j$

        $count = \dfrac{(upper\_new - i)((upper\_new - i) + 1)}{2} - \dfrac{(upper\_prev - i)((upper\_prev - i) + 1)}{2}$

        $i \mathrel{+}= 1$

        $total\_count \mathrel{+}= count$

        $upper\_prev = upper\_new$

4. return $total\_count$

Ο αλγόριθμος αυτός έχει πολυπλοκότητα $\mathcal{O}(nlogn)$, και εφόσον θα χρησιμοποιήσουμε τον αλγόριθμο του υποερωτήματος (a) όπου καλούμε την $F$, $\mathcal{O}(logM)$ φορές, συνολικά η πολυπλοκότητα θα είναι $\mathcal{O}(nlogn+logM)$.

## Άσκηση 3: Άθροισμα Στοιχείων Υποσυνόλων και Υπακολουθιών

### (a)

Το πρόβλημα αυτό είναι μια παραλλαγή του κλασικού προβλήματος subset sum με έναν επιπρόσθετο περιορισμό που είναι η παράμετρος k.

Θα λύσουμε το πρόβλημα με δυναμικό προγραμματισμό, αφού προσπαθήσουμε να "διασπάσουμε" το αρχικό πρόβλημα σε υποπροβλήματα των οποίων οι λύσεις επικαλύπτονται. Θεωρούμε λοιπόν ένα στιγμιότυπο του προβλήματος $I(a_1, ..., a_n; B; k)$, το οποίο επιστρέφει ναι αν υπάρχει ακολουθία $(a_1, ..., a_n)$ με μέγιστο αριθμό στοιχείων k, της οποίας το άθροισμα ισούται με B. Θα εξετάσουμε την απόφαση που πρέπει να πάρει ο αλγόριθμος για το τελευταίο στοιχείο της ακολουθίας.

$$\begin{cases} (a_1, ..., a_n) = B - a_n \ and \ k-1, & \text{if } a_n \ is \ needed \\ (a_1, ..., a_n) = B \ and \ k, & \text{if } a_n \ is \ not \ needed \end{cases} \tag{1}$$

Αν το τελευταίο στοιχείο $a_n$ ήταν στο σύνολο S που αποτελεί λύση για το I, τότε το σύνολο $T_1 = S \backslash a_k$ είναι λύση στο υποπρόβλημα $I_1 = (a_1, ..., a_n - 1; B - a_n; k - 1)$, δηλαδή στο υποπρόβλημα όπου επιτρέπονται εώς k-1 αριθμοί στην ακολουθία και το σύνολο $(a_1, ..., a_n)$ έχει άθροισμα $B - a_n$. Το k-1 είναι απαραίτητο, καθώς για να μπορούμε να προσθέσουμε το τελευταίο στοιχείο θα πρέπει να υπάρχει χώρος για ένα ακόμη στοιχείο, έτσι τα υπόλοιπα στοιχεία θα ικανοποιούν τη λύση για το υποπρόβλημα με k-1.

Αν το τελευταίο στοιχείο $a_n$ δεν ανήκει στο σύνολο S που αποτελεί λύση για το πρόβλημα I, τότε το σύνολο S αποτελεί λύση σε ένα άλλο υποπρόβλημα $I_2 = (a_1, ..., a_n - 1; B; k)$, διότι το σύνολο $S = (a_1, ..., a_n)$ περιέχει μέσα του ήδη ένα υποσύνολο $(a_1, ..., a_n - 1)$ που έχει άθροισμα B.

Παρατηρούμε πως απο τα δύο αυτά υποπροβλήματα, μπορούμε να "χτίσουμε" τη λύση για το γενικότερο πρόβλημα I.

Ορισμός: Για κάθε ακέραιο $0 \leq i \leq n$, $0 \leq b \leq B$, $0 \leq \kappa \leq k$ ορίζουμε τη συνάρτηση $F(i, b, \kappa) = 1$ αν υπάρχει υποσύνολο με αριθμό στοιχείων μικρότερο ή ίσο απο $\kappa$, τέτοιο ώστε το άθροισμα των στοιχείων αυτών να ισούται με b, αλλιώς 0. Τα base cases είναι τα εξής:

1. $F(i, 0, \kappa) = 1$ for all $0 \leq i \leq n$, $0 \leq \kappa \leq k$
2. $F(i, b, 0) = 0$ for all $0 \leq i \leq n$, $0 \leq b \leq B$
3. $F(0, b, \kappa) = 0$ for all $0 \leq b \leq B$, $0 \leq \kappa \leq k$

Αναδρομική σχέση:

$$F(i, b, \kappa) = \begin{cases} F(i-1, b, \kappa), & \text{if } a_i > b \\ \\ max(F(i-1, b, \kappa), (F(i-1, b - a_i, \kappa - 1)), & \text{otherwise} \end{cases} \tag{2}$$

Έτσι, σύμφωνα με τη παραπάνω σχέση, θα φτιαχτεί ένας $n \times B \times k$ πίνακας. Στο τέλος, για να επιστρέψουμε το ζητούμενο πλήθος των υποσυνόλων, θα πάμε στο $n \times B \times k$ πινακάκι, και για κάθε στοιχείο απο το τελευταίο μέχρι το πρώτο, θα αφαιρούμε απο το B το $a_i$ ($diff = B - a_i$) και θα πηγαίνουμε στη στήλη diff, επαναλαμβάνοντας μέχρι diff=0. Αν βρεθεί diff=0, τότε θα αυξάνουμε το μέτρημα. Για κάθε $i$, θα επαναλαμβάνουμε τη διαδικασία μέχρι να ελέγξουμε όλα τα προηγούμενα νούμερα.

### Algorithm

1. Initialize array $F[n, B, k]$ filled with zeros
2. **for** $\kappa = 1$ **to** B

   **for** $i = 1$ **to** N

   **for** $b = 0$ **to** B

   **if** $b = 0$

   $F[i, 0, \kappa] = 1$

   **else if** $a_i > b$

   $F[i, b, \kappa] = F[i-1, b, \kappa]$

   **else**

   $F[i, b, \kappa] = max\{F[i-1, b, \kappa], F[i-1, b-a_i, \kappa-1]\}$

Η πολυπλοκότητα του αλγόριθμου θα είναι $\mathcal{O}(n \times B \times k)$.

### (b)

Το συγκεκριμένο πρόβλημα είναι μια παραλλαγή του προβλήματος *Maximum sum increasing subsequence* όπου έχουμε μια επιπλέον μεταβλητή που είναι το k, και για το οποίο θέλουμε να υπολογίσουμε μια k-σχεδόν αύξουσα ακολουθία με το μέγιστο άθροισμα.

Θα χρησιμοποιήσουμε δυναμικό προγραμματισμό και θα ορίσουμε μια αναδρομική συνάρτηση $M[i, k]$ όπου για δεδομένο στοιχείο στην i-οστή θέση θα επιστρέφει το μέγιστο άθροισμα της $k$-σχεδόν αύξουσας υπακολουθίας.

Η απόφαση που πρέπει να παίρνει ο αλγόριθμος για έναν αριθμό είναι αν θα συμπεριλάβει έναν αριθμό στην υπακολουθία ή όχι. Οπότε για κάθε $i$, $k$:

i Αν το στοιχείο $i$ δεν συμπεριλαμβάνεται στην υπακολουθία, τότε: $M[i, k] = M[i-1, k]$

ii Αν συμπεριλάβουμε το στοιχείο $i$ τότε υπάρχουν δύο περιπτώσεις:

   (a) Το $a_i$ συνεχίζει την αύξουσα υπακολουθία $M[i, k] = M[i-1, k] + a_i$

   (b) Το $a_i$ είναι σημείο διακοπής: $M[i, k] = M[i-1, k-1] + a_i$

Αυτό μπορούμε να το γράψουμε ως αναδρομική σχέση:

$$M(i, k) = \begin{cases} 0, & if \quad i = 0 \\ M[i, k] = M[i-1, k] + a_i, & if \quad a_i > a_{i-1} \\ max(M[i-1, k], (M[i-1, k-1] + a_i)), & if \quad a_i \leq a_{i-1} \end{cases}$$

Αυτό δηλαδή, σημαίνει πως αν το τελευταίο στοιχείο $i$ είναι μεγαλύτερο απο το προηγούμενό του, οπότε η αύξουσα σειρά συνεχίζεται, πρόσθεσέ το στην μέγιστη αύξουσα σειρά που λήγει στο προηγούμενό του. Αλλιώς, αν είναι μικρότερο ή ίσο, τότε η αύξουσα σειρά που υπήρχε προηγουμένως διακόπτεται, οπότε είτε δεν το παίρνεις καθώς απαγορεύεται να προσθέσουμε σημείο διακοπής, είτε το προσθέτεις στη μέγιστη αύξουσα υπακολουθία του που λήγει στο προηγούμενο νούμερο με k-1 σημεία διακοπής.

### Algorithm

1. Initialize array $M[n, k]$ **and** $max = 0$
2. **for** $\kappa$ **from** 0 **to** k

   **for** i **from** 1 **to** n-1

   **if** $a_i > a_j$

   $M[i, \kappa] = M[i-1, \kappa] + \alpha_i$

   **else if** $a_i \leq a_{i-1}$

   $M[i, \kappa] = max\{M[i-1, \kappa], M[i-1, \kappa-1] + a_i\}$
3. array result[]
4. **for** i=n **to** 0

   **if** $M[i, k]$ != $M[i-1, k]$

   result.append($a_i$)

Έτσι θα φτιάξουμε ένα πινακάκι $n \times k$. Η πολυπλοκότητα χωρίς το περιορισμό του k ήταν $\mathcal{O}(n^2)$ επομένως στη περίπτωσή μας θα έχουμε $\mathcal{O}(kn^2)$.

Για base cases, έχουμε πως για $i = 0$ τότε $M[0, k] = 0$ και για $k = 0$, που είναι η απλή περίπτωση της αύξουσας υπακολουθίας με μέγιστο άθροισμα, είναι $M[i, 0] = a_i + M[j, 0]$ όπου j η θέση του στοιχείου για την οποία ισχύει $j < i$ και $a_j < a_i$ και $M[j, 0]$ η αύξουσα υπακολουθία με το μέγιστο άθροισμα που τελειώνει στο $j$.

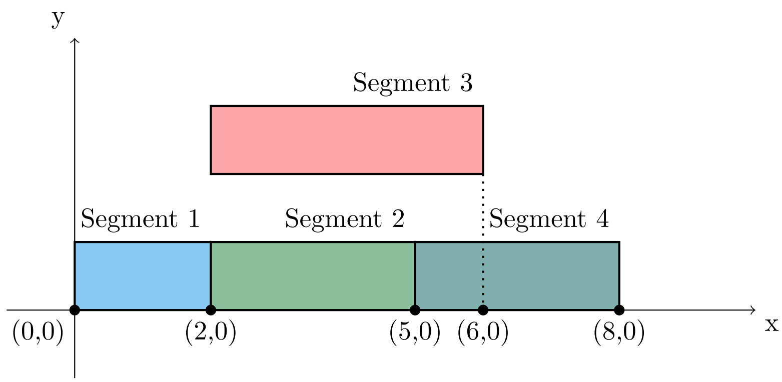
Έστω για παράδειγμα η ακολουθία $\alpha = (2, 15, 4, 6, 6, 14)$ και ψάχνουμε για k=2 τη μέγιστη 2-σχεδόν υπακολουθία της. Το πινακάκι που σχηματίζεται είναι το εξής:

| $i \backslash k$ | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 2 | 2 | 2 | 2 |
| 15 | 17 | 17 | 17 |
| 4 | 17 | 21 | 21 |
| 6 | 17 | 27 | 27 |
| 6 | 17 | 27 | 33 |
| 14 | 26 | 41 | 47 |

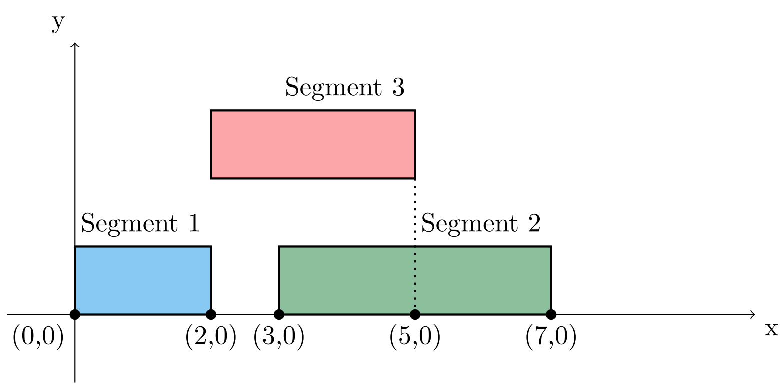
Οι αριθμοί που αποτελούν μέρος της λύσης, είναι αυτοί που στην απο πάνω γραμμή έχουν μικρότερη τιμή.

## Άσκηση 4:Μη Επικαλυπτόμενα Διαστήματα Μέγιστου Συνολικού Μήκους

**(a)**




Στη παραπάνω περίπτωση διαστημάτων, ο άπληστος αλγόριθμος θα διαλέξει το segment 3 διότι έχει το μεγαλύτερο μήκος, ωστόσο φαίνεται καθαρά πως η επιλογή των άλλων διαστημάτων οδηγεί σε καλύτερο αποτέλεσμα με συνολικό μήκος 8 έναντι του μήκους 6.




Αντίστοιχα με προηγουμένως, ο άπληστος αλγόριθμος και σε αυτή τη περίπτωση θα αποτύχει διότι θα επιλέξει το διάστημα 3 έναντι του 2, καθώς το segment 3 έχει μικρότερο χρόνο ολοκλήρωσης. Ωστόσο η επιλογή αυτή θα οδηγήσει σε συνολικό μήκος 5 ενώ αν είχε επιλέξει το διάστημα 2 θα είχε συνολικό μήκος 6.

**(b)**

Θα χρησιμοποιήσουμε δυναμικό προγραμματισμό. Προφανώς όταν υπάρχει μόνο ένα διάστημα, η βέλτιστη επιλογή είναι να διαλέξουμε αυτό, με συνολικό μήκος το μήκος του διαστήματος αυτού. Ορίζουμε ως OPT(i) τη συνάρτηση που επιστρέφει το συνολικό μήκος μη επικαλυπτόμενων διαστημάτων, έχοντας διαθέσιμα τα πρώτα i διαστήματα. Για κάθε διάστημα i, ο αλγόριθμος θα πρέπει να παίρνει την εξής απόφαση:

i Επιλέγει το διάστημα i στη βέλτιστη λύση, έτσι η βέλτιστη λύση είναι η βέλτιση λύση μέχρι και το j που δεν επικαλύπτεται με το i, μαζί με το i

ii Δεν επιλέγει το διάστημα i στη βέλτιστη λύση, οπότε η βέλτιστη λύση είναι μέχρι και το i-1, εφόσον δεν το διαλέξαμε

Απο αυτές τις δύο επιλογές διαλέγει τη μεγαλύτερη, άρα η αναδρομική σχέση γίνεται:

$$OPT(i) = max\{OPT(j_i) + (f_i - s_i), OPT(i-1)\}$$

### Algorithm

1. Sort the given intervals in ascending order by their finish time
2. Initialize arrays $A[n]$, $B[n]$
3. $A[1] = f_1 - s_1$, $B[1] = \{1\}$ *//B holds the indices of the intervals that are part of the solution*
4. **for** i=2 to n

   find $j_i$, the largest j such that $f_j < s_i$

   **if** $A[j_i] + (f_i - s_i) \geq A[i-1]$

   $A[i] = A[j-1] + (f_i - s_i)$ **and** $B[i] = B[j_i] \cup \{i\}$

   **else** $A[i] = A[i-1]$ **and** $B[i] = B[i-1]$
5. return $A[n]$

Αρχικά ταξινομούμε τα δοσμένα διαστήματα σε αύξουσα σειρά χρόνου ολοκλήρωσης, το οποίο παίρνει $\mathcal{O}(nlogn)$ χρόνο και στη συνέχεια κάνουμε $n$ επαναλήψεις. Επομένως, συνολικά η χρονική πολυπλοκότητα που επικρατεί είναι $\mathcal{O}(nlogn)$.

## Άσκηση 5:Θέσεις Στάθμευσης

### (a)

Η στρατηγική μας για να μεγιστοποιήσουμε τη συνολική αξία των αυτοκινήτων θα είναι να διαλέγουμε κάθε φορά το αυτοκίνητο με τη μέγιστη αξία. Αυτό μπορούμε να το επιτύγχουμε ταξινομώντας τα αυτοκίνητα σε φθίνουσα σειρά αξίας. Για κάθε θέση $\ell$, ξεκινώντας απο το τέλος, θα διαλέγουμε το διαθέσιμο αυτοκίνητο με τη μέγιστη αξία, δηλαδή θα παρκάρουμε το αυτοκίνητο στην όσο το δυνατό τελευταία θέση που είναι διαθέσιμη για παρκάρισμα. Με αυτό το τρόπο, θα αφήνουμε χώρο για τα υπόλοιπα αυτοκίνητα να παρκάρουν στις προηγούμενες θέσεις. Επομένως, τα γενικά βήματα για την επίλυση θα είναι:

1. Ταξινόμησε τα αυτοκίνητα σε φθίνουσα σειρά αξίας
2. Για κάθε αυτοκίνητο:

   i Βρες τη μέγιστη πιθανή θέση $\ell$ η οποία είναι κενή και $\ell \leq d_i$. Πάρκαρε το αυτοκίνητο στη θέση αυτή και σημείωσέ την ως κατειλλημένη

   ii Εάν δεν υπάρχει τέτοια θέση, τότε μην παρκάρεις το συγκεκριμένο αυτοκίνητο

Η ταξινόμηση παίρνει $\mathcal{O}(nlogn)$ και επειδή στη χειρότερη περίπτωση θα χρειαστεί να ελέγξουμε για κάθε ένα απο τα $n$ αυτοκίνητα, $n$ θέσεις, συνολικά η πολυπλοκότητα του αλγόριθμου θα είναι $\mathcal{O}(n^2)$.

Θα αποδείξουμε την ορθότητα του άπληστου κριτηρίου. Υποθέτουμε πως υπάρχει κάποιο άλλο πρόγραμμα παρκαρίσματος "*optimal*", έστω **S***, το οποίο παρκάρει τα αυτοκίνητα σε διαφορετική σειρά απότι ο αλγόριθμός μας, έστω **S**. Θα δείξουμε πως το βέλτιστο αυτό πρόγραμμα μπορεί να μετατραπεί στο άπληστο με ανταλλαγές, χωρίς να μειωθεί η συνολική αξία (επιχείρημα ανταλλαγής).

Έστω πως στη λύση **S***, ένα αυτοκίνητο με χαμηλότερη αξία, παρκαρίστηκε νωρίτερα στη σειρά απότι ένα άλλο αυτοκίνητο με μεγαλύτερη αξία, αυτό σημαίνει πως η λύση **S*** θα μπορούσε να είναι και καλύτερη.

Σε αντίθεση με το πρόγραμμα **S**, η οποία έχει παρκάρει τα αυτοκίνητα σε φθίνουσα σειρά αξίας, η υποθετική λύση **S*** μπορεί να έχει αυτοκίνητα σε διαφορετική σειρά. Έστω λοιπόν, το πρώτο αυτοκίνητο που βρίσκουμε στην **S*** $CarA$ το οποίο διαφέρει απο αυτό που παρκαρίστηκε σύμφωνα με την **S**, έστω $CarB$. Εάν το $CarB$ έχει μεγαλύτερη αξία απο το $CarA$, αυτό σημαίνει πως η **S*** μπορεί να βελτιωθεί, οπότε ανταλλάσουμε τα δύο αυτοκίνητα στη συγκεκριμένη θέση. Μετά την ανταλλαγή, εφόσον το $CarB$ έχει πιο υψηλή αξία απο το $CarA$ τότε η ανταλλαγή επέφερε περισσότερη αξία (ή την ίδια αλλά ποτέ χαμηλότερη). Συνεχίζοντας τις ανταλλαγές με τον ίδιο τρόπο, στο τέλος η **S*** θα γίνει η **S**, οπότε η λύση με το άπληστο κριτήριο θεωρούμε πως είναι βέλτιστη.

**(b)**

Στη περίπτωση που τα αυτοκίνητα έχουν διαφορετικό μήκος, θα προσεγγίσουμε το πρόβλημα με δυναμικό προγραμματισμό. Αρχικά ταξινομούμε τα αυτοκίνητα σε αύξουσα σειρά με βάση τη μέγιστη απόσταση $d_i$.

Θεωρούμε τα υποσύνολα $A_i = \{1, ..., i\}$ που αποτελούνται απο τα πρώτα $i$ αυτοκίνητα στο σύνολο A των αυτοκινήτων, για κάθε $i = 0, ..., n$. Ορίζουμε τη συνάρτηση $P(A_i, D)$ η οποία επιστρέφει το βέλτιστο κέρδος απο το υποσύνολο αυτοκινήτων $A_i$ με μέγιστη απόσταση $D$ για το τελευταίο αυτοκίνητο. Υποθέτουμε ότι γνωρίζουμε τα $P(A_{n-1}, d_{n-1})$ και $P(A_{n-1}, min\{d_n - s_n, d_{n-1}\}$ και ελέγχουμε το τελευταίο αυτοκίνητο. Η επιλογή που έχει να κάνει ο αλγόριθμος για το τελευταίο αυτοκίνητο, είναι:

i Να παρκάρει το τελευταίο αυτοκίνητο και να έχει κέρδος $v_n + P(A_{n-1}, min\{d_n - s_n, d_{n-1})\}$

ii Να μην παρκάρει το τελευταίο αυτοκίνητο και να έχει κέρδος $P(A_{n-1}, d_{n-1})$

Απο τις δύο αυτές επιλογές, ο αλγόριθμος θα διαλέξει αυτή που επιφέρει μεγαλύτερο κέρδος.Επομένως, το βέλτιστο κέρδος μπορεί να υπολογιστεί απο την αναδρομική σχέση:

$$P(A_i, D) = \begin{cases} max\{v_i + P(A_{i-1}, D - s_i), P(A_{i-1}, D)\}, & \text{if} \quad i \geq 1, \quad 1 \leq D \leq d_i \\ P(A_i, d_i), & \text{if} \quad i \geq 1, \quad d_i < D \\ 0, & \text{if} \quad i = 0 \quad or \quad D \leq 0 \end{cases} \tag{3}$$

## Algorithm

1. Sort cars in ascending order of their maximum distance
2. Initialize array $P[i, D]$ filled with zeros *//i represents the last car and D the maximum allowable distance*
3. **for** each car i in the sequence, $1 \leq i \leq N$

   **for** D = 1 **to** $max\_D$ *//max _D is the greatest maximum distance*

   **if** $d_i < D$

   $P[i, D] = P[i, d_i]$

   **else** $P[i, D] = max\{v_i + P[i-1, D - s_i], P[i-1, D]\}$

Για τη ταξινόμηση θα χρειαστεί $\Theta(nlogn)$ και στη συνέχεια $\Theta(Nmax\_D)$ για τον υπολογισμό των τιμών της αναδρομικής σχέσης. Οπότε συνολικά $\Theta(nlogn + Nmax\_D)$.

Στο τέλος του αλγόριθμου, για να βρούμε ποια αυτοκίνητα αποτελούν μέρος τη λύσης, ξεκινώντας απο το τελευταίο στοιχείο του πίνακα, θα ανεβαίνουμε γραμμές μέχρι να βρούμε κάποιο αυτοκίνητο που στην πάνω γραμμή έχει μικρότερη τιμή. Τότε, θα βάζουμε το αυτοκίνητο αυτό στη λύση και θα πηγαίνουμε στην $D - s_i$ στήλη, επαναλαμβάνοντας την ίδια διαδικασία για τα υπόλοιπα αυτοκίνητα, μέχρι να φτάσουμε στο 0.

Για παράδειγμα, έστω ότι έχουμε τα αυτοκίνητα $C_1 = (1, 3, 1)$, $C_2 = (2, 2, 2)$, $C_3 = (1, 4, 1)$, $C_4 = (1, 4, 2)$. Μετά την ταξινόμηση, ο αλγόριθμος θα δημιουργήσει το παρακάτω πινακάκι

| $A_i \backslash D$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| $C_2$ | 0 | 0 | 2 | 2 | 2 |
| $C_1$ | 0 | 1 | 2 | 3 | 3 |
| $C_3$ | 0 | 1 | 2 | 3 | 4 |
| $C_4$ | 0 | 1 | 2 | 3 | 4 |

Τα αυτοκίνητα τα οποία διαλέγουμε στη τελική ακολουθία, είναι αυτά τα οποία έχουν χρωματισμένο με πράσινο το κουτάκι στο οποίο θα παρκαριστούν. Δηλαδή το αυτοκίνητο $C_3$ θα παρκαριστεί στη θέση απο 3 έως 4, το $C_1$ στη θέση απο 2 έως 3 και το $C_2$ που έχει και μήκος 2 απο τη θέση 0 έως 2.