

Αλγόριθμοι και Προχωρημένες Δομές Δεδομένων

Αλγοριθμικές τεχνικές
Δυναμικός Προγραμματισμός
Γκόγκος Χρήστος 19/1/2023

Δυναμικός Προγραμματισμός

- Ο δυναμικός προγραμματισμός είναι μια τεχνική που, όταν μπορεί να εφαρμοστεί, εντοπίζει βέλτιστες λύσεις χωρίς να απαιτεί απαγορευτικά μεγάλο χρόνο εκτέλεσης
- Ο δυναμικός προγραμματισμός ξεκινά με έναν αναδρομικό αλγόριθμο ή έναν αναδρομικό ορισμό
- Είναι μια τεχνική αποδοτικής υλοποίησης αναδρομικών αλγορίθμων μέσω της αποθήκευσης ενδιάμεσων αποτελεσμάτων που αναπαριστούν τις συνέπειες όλων των πιθανών αποφάσεων μέχρι κάποιο σημείο κάθε φορά
- Πρόκειται για μια «ανταλλαγή» χώρου με χρόνο – αποφεύγοντας τον επαναυπολογισμό αποτελεσμάτων
- Χρησιμοποιώντας προηγούμενα αποτελέσματα, με συστηματικό τρόπο, επιτυγχάνει την αποδοτική λύση του προβλήματος

Ιδιότητες προβλημάτων δυναμικού προγραμματισμού

- Ο δυναμικός προγραμματισμός χρησιμοποιείται σε προβλήματα βελτιστοποίησης με:
 - **Επικαλυπτόμενα υποπροβλήματα (overlapping subproblems):** Ο αναδρομικός αλγόριθμος επισκέπτεται συνεχώς τα ίδια υποπροβλήματα
 - **Βέλτιστη υποδομή (optimal substructure):** Η βέλτιστη λύση ενός προβλήματος λαμβάνεται συνδυάζοντας βέλτιστες λύσεις υποπροβλημάτων (η βέλτιστη λύση του προβλήματος μπορεί να εκφραστεί με όρους βέλτιστων λύσεων υποπροβλημάτων)

Παράδειγμα 1

Η ακολουθία Fibonacci

<https://www.laweekly.com/5-mind-boggling-facts-about-fibonacci-sequences/>

Αριθμοί Fibonacci (με αναδρομή)

- Οι αριθμοί Fibonacci ορίζονται από την ακόλουθη αναδρομική σχέση:

$$F_0 = 0, F_1 = 1$$


$$F_n = F_{n-1} + F_{n-2}$$

- Πρόκειται για την ακολουθία:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- Ο απλοϊκός αναδρομικός αλγόριθμος έχει εκθετικό χρόνο εκτέλεσης!

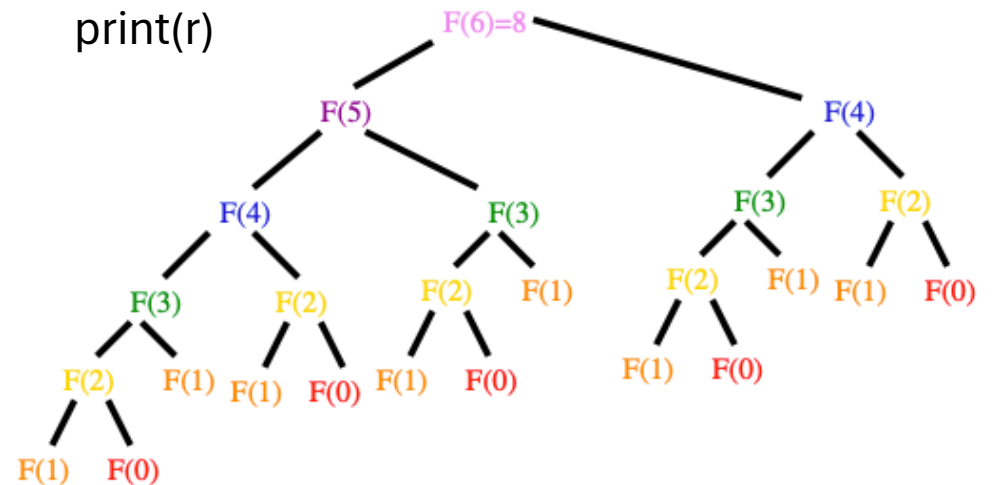
fib_r(100) ❌

```
def fib_r(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    return fib_r(n - 1) + fib_r(n - 2)
```

αναδρομή



r = fib_r(6)
print(r)



Αριθμοί Fibonacci (με caching)

- Αναδρομικές κλήσεις συνάρτησης
- memoization: αποθήκευση αποτελεσμάτων σε έναν πίνακα
- Η πολυπλοκότητα του αλγορίθμου είναι $O(n)$

`fib_c(100)` → 354224848179261915075

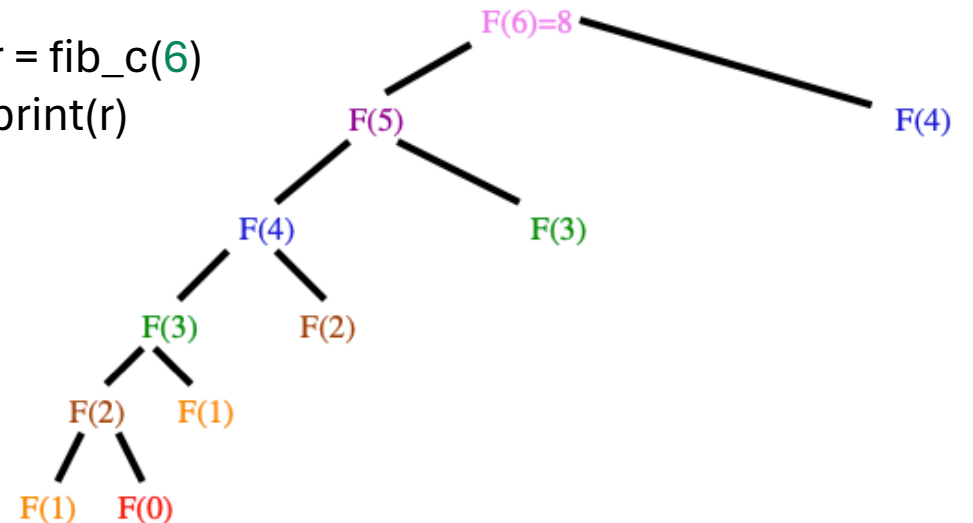
```
F = [None] * 1000  
F[0]=0  
F[1]=1
```

```
def fib_c(n):  
    if F[n] is None:  
        F[n] = fib_c(n-1) + fib_c(n-2)  
    return F[n]
```

αναδρομή



```
r = fib_c(6)  
print(r)
```



Αριθμοί Fibonacci (με δυναμικό προγραμματισμό) 1/2

- Η λύση αυτή δεν έχει αναδρομικές κλήσεις συνάρτησης
- Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n)$
- Ο χώρος που απαιτεί είναι $O(n)$

```
def fib_dp(n):  
    F = [None] * (n+1)  
    F[0], F[1] = 0, 1  
    for i in range(2, n+1):  
        F[i] = F[i-1] + F[i-2]  
    return F[n]
```

```
r = fib_dp(6)  
print(r)
```

Αριθμοί Fibonacci (με δυναμικό προγραμματισμό) 2/2

- Αποθήκευση μόνο των δύο τελευταίων αποτελεσμάτων κάθε φορά και όχι όλων των ενδιάμεσων αποτελεσμάτων
- Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n)$
- Ο χώρος που απαιτεί είναι $O(1)$

```
def fib_ultimate(n):  
    if n == 0:  
        return 0  
    back2, back1 = 0, 1  
    for i in range(2, n):  
        next = back1 + back2  
        back2 = back1  
        back1 = next  
    return back1 + back2
```

```
r = fib_ultimate(6)  
print(r)
```


LRU cache - functools της Python

functools — Higher-order functions and operations on callable objects

Source code: [Lib/functools.py](#)

The **functools** module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

LRU = Least Recently Used

<https://docs.python.org/3/library/functools.html>

```
from functools import lru_cache
```

```
@lru_cache(maxsize=None)
```

```
def fib_r(n):
```

```
    if n == 0:
```

```
        return 0
```

```
    if n == 1:
```

```
        return 1
```

```
    return fib_r(n - 1) + fib_r(n - 2)
```

```
r = fib_r(50)
```

```
print(r)
```

Παράδειγμα 2

Σύγκριση συμβολοσειρών

Προσεγγιστικό ταίριασμα συμβολοσειρών

- Αρχικά, θα πρέπει να οριστεί μια συνάρτηση κόστους που να επιστρέφει πόσο απέχουν δύο συμβολοσειρές
- Ορίζονται 3 τύποι στοιχειωδών αλλαγών που μπορούν να γίνουν έτσι ώστε η συμβολοσειρά P να μετατραπεί στη συμβολοσειρά T:
 - Αντικατάσταση (substitution): αντικατάσταση ενός χαρακτήρα στη P
 - Εισαγωγή (insertion): εισαγωγή ενός χαρακτήρα στη P
 - Διαγραφή (deletion): διαγραφή ενός χαρακτήρα από τη P
- Κάθε αλλαγή κοστολογείται με 1



Παράδειγμα

- P = “thou-shalt”
- T = “you should”
- Ζητείται ο συντομότερος τρόπος που το P μπορεί να μετασχηματιστεί σε T:
 1. thou-shalt \rightarrow hou-shalt (διαγραφή του t)
 2. hou-shalt \rightarrow you-shalt (αντικατάσταση του h με y)
 3. you-shalt \rightarrow you-sholt (αντικατάσταση του a με o)
 4. you-sholt \rightarrow you-shoult (εισαγωγή του u)
 5. you-shoult \rightarrow you-should (αντικατάσταση του t με d)

Αναδρομικός αλγόριθμος

- Ο αλγόριθμος είναι ορθός, αλλά πολύ αργός!
- Είναι αργός επειδή επαναλαμβάνει τους ίδιους υπολογισμούς ξανά και ξανά
- Για κάθε κλήση της συνάρτησης σε μια θέση της συμβολοσειράς πραγματοποιούνται 3 αναδρομικές κλήσεις, άρα η πολυπλοκότητα είναι εκθετική (τουλάχιστον 3^n)

```
def string_compare_r(p, t, i, j):  
    if i == 0:  
        return j  
    if j == 0:  
        return i  
  
    if p[i] == t[j]:  
        match_cost = string_compare_r(p, t, i - 1, j - 1)  
    else:  
        match_cost = string_compare_r(p, t, i - 1, j - 1) + 1  
    insert_cost = string_compare_r(p, t, i, j - 1) + 1  
    delete_cost = string_compare_r(p, t, i - 1, j) + 1  
  
    return min([match_cost, insert_cost, delete_cost])
```

```
p = "ABBC"  
t = "AXB"  
cost = string_compare_r(p, t, len(p)-1, len(t)-1)  
print(cost)
```

Δυναμικός προγραμματισμός

- Έστω $D[i,j]$ ο ελάχιστος αριθμός διαφορών ανάμεσα στις υποσυμβολοσειρές $P_1P_2...P_i$ και $T_1T_2...T_j$
- Η $D[i,j]$ είναι η **ελάχιστη τιμή** από:
 - $D[i-1,j-1]$ αν $P_i = T_j$
 - $D[i-1,j-1] + 1$ αν $P_i \neq T_j$ (πρέπει να γίνει αντικατάσταση του i χαρακτήρα της P με τον j χαρακτήρα της T)
 - $D[i,j-1] + 1$ (ολίσθηση προς τα δεξιά του $P[i]$ και των χαρακτήρων που ακολουθούν, εισαγωγή του $T[j]$ στο $P[i]$)
 - $D[i-1,j] + 1$ (διαγραφή του $P[i]$ από την P)

Κώδικας δημιουργίας πίνακα αποθήκευσης αποτελεσμάτων

```
from dataclasses import dataclass

@dataclass
class Cell:
    cost: int
    parent: str

    def __lt__(self, other):
        return self.cost < other.cost

    def __repr__(self):
        if self.parent is None:
            return f"{self.cost}"
        else:
            return f"{self.cost},{self.parent}"
```

Οργάνωση περιεχομένων κελιών πίνακα

```
def string_compare(p, t):
    m = [[Cell(cost=0, parent=None)] * (len(t) + 1) for _ in range(len(p) + 1)]
    for j in range(1, len(t) + 1):
        m[0][j] = Cell(cost=j, parent="I") # INSERT
    for i in range(1, len(p) + 1):
        m[i][0] = Cell(cost=i, parent="D") # DELETE
    for i in range(1, len(p) + 1):
        for j in range(1, len(t) + 1):
            if p[i - 1] == t[j - 1]:
                a = Cell(m[i - 1][j - 1].cost, "M") # MATCH
            else:
                a = Cell(m[i - 1][j - 1].cost + 1, "S") # SUBSTITUTION
            b = Cell(m[i][j - 1].cost + 1, "I") # INSERT
            c = Cell(m[i - 1][j].cost + 1, "D") # DELETE
            m[i][j] = min([a, b, c])
    return m
```

Εκτέλεση αλγορίθμου δυναμικού προγραμματισμού

```
def string_compare(p, t):
    m = [[Cell(cost=0, parent=None)] * (len(t) + 1) for _ in range(len(p) + 1)]
    for j in range(1, len(t) + 1):
        m[0][j] = Cell(cost=j, parent="I")
    for i in range(1, len(p) + 1):
        m[i][0] = Cell(cost=i, parent="D")
    for i in range(1, len(p) + 1):
        for j in range(1, len(t) + 1):
            if p[i - 1] == t[j - 1]:
                a = Cell(m[i - 1][j - 1].cost, "M")
            else:
                a = Cell(m[i - 1][j - 1].cost + 1, "S")
            b = Cell(m[i][j - 1].cost + 1, "I")
            c = Cell(m[i - 1][j].cost + 1, "D")
            m[i][j] = min([a, b, c])
    return m
```

Πολυπλοκότητα: $O(n^2)$

Ο πίνακας η μετά την κλήση της:

```
p = "thou-shalt"  
t = "you-should"  
m = string_compare(p, t)
```

			y	o	u	-	s	h	o	u	l	d
			0	1	2	3	4	5	6	7	8	9
t	0	1,D	1,S	2,S	3,S	4,S	5,S	6,S	7,S	8,S	9,S	10,S
h	1	2,D	2,S	2,S	3,S	4,S	5,S	5,M	6,I	7,I	8,I	9,I
o	2	3,D	3,S	2,M	3,S	4,S	5,S	6,S	5,M	6,I	7,I	8,I
u	3	4,D	4,S	3,D	2,M	3,I	4,I	5,I	6,I	5,M	6,I	7,I
-	4	5,D	5,S	4,D	3,D	2,M	3,I	4,I	5,I	6,I	6,S	7,S
s	5	6,D	6,S	5,D	4,D	3,D	2,M	3,I	4,I	5,I	6,I	7,S
h	6	7,D	7,S	6,D	5,D	4,D	3,D	2,M	3,I	4,I	5,I	6,I
a	7	8,D	8,S	7,D	6,D	5,D	4,D	3,D	3,S	4,S	5,S	6,S
l	8	9,D	9,S	8,D	7,D	6,D	5,D	4,D	4,S	4,S	4,M	5,I
t	9	10,D	10,S	9,D	8,D	7,D	6,D	5,D	5,S	5,S	5,S	5,S

Κατασκευή της διαδρομής από τα περιεχόμενα του πίνακα m

```

def reconstruct_path(p, t, i, j, m):
    if m[i][j].parent is None:
        return
    if m[i][j].parent in "MS":
        reconstruct_path(p, t, i - 1, j - 1, m)
        print(m[i][j].parent, end="")
        return
    if m[i][j].parent == "I":
        reconstruct_path(p, t, i, j - 1, m)
        print(m[i][j].parent, end="")
        return
    if m[i][j].parent == "D":
        reconstruct_path(p, t, i - 1, j, m)
        print(m[i][j].parent, end="")
        return

p = "thou-shalt"
t = "you-should"
m = string_compare(p, t)
reconstruct_path(p, t, len(p), len(t), m)

```

			y	o	u	-	s	h	o	u	l	d
			0	1	2	3	4	5	6	7	8	9
t	0	0	1,I	2,I	3,I	4,I	5,I	6,I	7,I	8,I	9,I	10,I
h	1	2,D	2,S	2,S	3,S	4,S	5,S	5,M	6,I	7,I	8,I	9,I
o	2	3,D	3,S	2,M	3,S	4,S	5,S	6,S	5,M	6,I	7,I	8,I
u	3	4,D	4,S	3,D	2,M	3,I	4,I	5,I	6,I	5,M	6,I	7,I
-	4	5,D	5,S	4,D	3,D	2,M	3,I	4,I	5,I	6,I	6,S	7,S
s	5	6,D	6,S	5,D	4,D	3,D	2,M	3,I	4,I	5,I	6,I	7,S
h	6	7,D	7,S	6,D	5,D	4,D	3,D	2,M	3,I	4,I	5,I	6,I
a	7	8,D	8,S	7,D	6,D	5,D	4,D	3,D	3,S	4,S	5,S	6,S
l	8	9,D	9,S	8,D	7,D	6,D	5,D	4,D	4,S	4,S	4,M	5,I
t	9	10,D	10,S	9,D	8,D	7,D	6,D	5,D	5,S	5,S	5,S	5,S

Αποτέλεσμα: DSMMMMMMISMS

Παράδειγμα 3

Το πρόβλημα του 0/1 σακιδίου

Το πρόβλημα του 0/1 σακιδίου

- Έστω n αντικείμενα
- Αξίες αντικειμένων (θετικές τιμές): $v_1, v_2, v_3, \dots, v_n$
- Βάρη αντικειμένων (θετικές τιμές): $w_1, w_2, w_3, \dots, w_n$
- Χωρητικότητα σακιδίου (ακέραια θετική τιμή): C
- Ζητείται να βρεθεί το σύνολο $S \subseteq \{1, 2, 3, \dots, n\}$ για το οποίο ισχύει:

$$\max \sum_{i \in S} v_i$$
$$\sum_{i \in S} w_i \leq C$$

Βασική ιδέα

- Αν θεωρήσουμε ότι ήδη γνωρίζουμε τη βέλτιστη λύση $S \subseteq \{1, 2, 3, \dots, n\}$ με συνολική αξία $V = \sum_{i \in S} v_i$ τότε η λύση αυτή είτε θα περιέχει είτε δεν θα περιέχει το τελευταίο αντικείμενο n :
 - Αν η βέλτιστη λύση δεν περιέχει το n τότε η $S^* = \{1, 2, 3, \dots, n - 1\}$ θα είναι η βέλτιστη λύση στο υποπρόβλημα των πρώτων $n - 1$ αντικειμένων. Αν δεν ίσχυε αυτό τότε θα υπήρχε καλύτερη λύση και για το πρόβλημα των n αντικειμένων
 - Αν η βέλτιστη λύση περιέχει το n τότε θα πρέπει $w_n \leq C$ και η απομένουσα λύση των πρώτων $n - 1$ αντικειμένων θα αποτελεί τη βέλτιστη λύση στο υποπρόβλημα επιλογής από τα πρώτα $n - 1$ αντικείμενα με διαθέσιμη χωρητικότητα $C - w_n$.

Απόδειξη

- Γιατί αν στη βέλτιστη λύση περιέχεται το αντικείμενο n , τότε η λύση θα περιέχει τη βέλτιστη λύση για τα υπόλοιπα αντικείμενα με μέγιστη χωρητικότητα $C - w_n$;
- Για την περίπτωση που η βέλτιστη λύση $S \subseteq \{1, 2, 3, \dots, n\}$ και $n \in S$ ισχύει ότι:
 - Δεσμεύοντας w_n χωρητικότητα προκειμένου να συμπεριληφθεί στη λύση το αντικείμενο n παραμένει διαθέσιμη $C - w_n$ χωρητικότητα
 - Αν υπάρχει καλύτερη λύση $S^* \subseteq \{1, 2, 3, \dots, n - 1\}$ με συνολική αξία $V^* > V - v_n$ για μέγιστη χωρητικότητα $C - w_n$ τότε η $S^* \cup \{n\}$ θα είχε συνολική αξία $V^* + v_n > V - v_n + v_n = V$ που είναι άτοπο διότι υποθέσαμε ότι η λύση S είναι βέλτιστη

Υποπροβλήματα (i, j)

- Τα υποπροβλήματα ορίζονται με βάση δύο παραμέτρους, το πλήθος των i πρώτων αντικειμένων που συμμετέχουν στο πρόβλημα και τη χωρητικότητα του σακιδίου j
- Το «υποπρόβλημα» (n, C) είναι το αρχικό πρόβλημα
- Αν $V_{i,j}$ είναι η μέγιστη συνολική αξία του υποσυνόλου των i πρώτων αντικειμένων με συνολική χωρητικότητα το πολύ j τότε ισχύει ότι για κάθε $i = 1, 2, \dots, n$ και κάθε $j = 0, 1, 2, \dots, C$:

$$V_{i,j} = \begin{cases} V_{i-1,j} & \text{εάν } w_i > j \\ \max(V_{i-1,j}, V_{i-1,j-w_i} + v_i) & \text{εάν } w_i \leq j \end{cases}$$

Παράδειγμα

Πίνακας V

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	3	3	3
2	0	0	0	2	3	3	3
3	0	0	4	*4	4	6	7
4	0	0	4	4	4	8	*8

Αντικείμενο	Αξία v	Βάρος w
1	3	4
2	2	3
3	4	2
4	4	3

$C = 6$

Εντοπισμός λύσης: Ξεκινώντας από την κάτω δεξιά γωνία του πίνακα ($i = 4, j = 6$ στο παράδειγμα), το κελί αυτό ορίζεται ως **τρέχον κελί**. Ο αλγόριθμος ελέγχει αν το ακριβώς από πάνω κελί από το τρέχον έχει διαφορετική τιμή από το τρέχον και σταματά όταν βρεθεί στη γραμμή 0.

- Αν ναι τότε το αντικείμενο της γραμμής του τρέχοντος κελιού επιλέγεται και το νέο τρέχον κελί είναι πλέον αυτό που βρίσκεται στην αμέσως από πάνω γραμμή και στη στήλη που προκύπτει αφαιρώντας από το βάρος της στήλης του τρέχοντος κελιού το βάρος του επιλεγμένου αντικειμένου
- Αν όχι τότε τρέχον κελί γίνεται το κελί που βρίσκεται στην ακριβώς από πάνω γραμμή του πίνακα

$$V_{i,j} = \begin{cases} V_{i-1,j} & \text{εάν } w_i > j \\ \max(V_{i-1,j}, V_{i-1,j-w_i} + v_i) & \text{εάν } w_i \leq j \end{cases}$$

Κωδικοποίηση σε Python για το πρόβλημα 0/1 σακιδίου

```
def knapsack_dp(v, w, C, n):  
    V = [[0 for j in range(C + 1)] for i in range(n + 1)]  
  
    for i in range(1, n + 1):  
        for j in range(C + 1):  
            if w[i - 1] > j:  
                V[i][j] = V[i - 1][j]  
            else:  
                V[i][j] = max(V[i - 1][j], V[i - 1][j - w[i - 1]] + v[i - 1])  
    print("Συνολική αξία σακιδίου = ", V[n][C])  
    while n!=0:  
        if V[n][C] != V[n-1][C]:  
            print(f"Αντικείμενο = {n} Αξία = {v[n-1]} Βάρος = {w[n-1]}")  
            C = C - w[n-1]  
        n-=1
```

```
n = 4 # πλήθος αντικειμένων  
v = [3, 2, 4, 4] # αξίες αντικειμένων  
w = [4, 3, 2, 3] # βάρη αντικειμένων  
C = 6 # χωρητικότητα  
knapsack_dp(v, w, C, n)
```

$$V_{i,j} = \begin{cases} V_{i-1,j} & \text{εάν } w_i > j \\ \max(V_{i-1,j}, V_{i-1,j-w_i} + v_i) & \text{εάν } w_i \leq j \end{cases}$$

Συνολική αξία σακιδίου = 8
Αντικείμενο = 4 Αξία = 4 Βάρος = 3
Αντικείμενο = 3 Αξία = 4 Βάρος = 2

Απόδοση αλγορίθμου δυναμικού προγραμματισμού για το πρόβλημα 0/1 σακιδίου

- Ο αλγόριθμος αποτελείται από δύο φάσεις, τη συμπλήρωση του πίνακα V και τον εντοπισμό των αντικειμένων που συμμετέχουν στη λύση από τον πίνακα V :
 - Για τη συμπλήρωση του πίνακα ο αλγόριθμος διαθέτει $O(1)$ χρόνο για την επίλυση καθενός από τα $n + 1 * C + 1 = O(nC)$ προβλήματα, συνεπώς, εκτελείται σε $O(nC)$ χρόνο
 - Για τον εντοπισμό των αντικειμένων που συμμετέχουν στη λύση ο αλγόριθμος διαθέτει $O(1)$ χρόνο για κάθε αντικείμενο, συνεπώς, εκτελείται σε $O(n)$ χρόνο
- Άρα, καθώς η πολυπλοκότητα του αλγορίθμου καθορίζεται από τη φάση με την υψηλότερη πολυπλοκότητα, η πολυπλοκότητα του αλγορίθμου είναι **$O(nC)$**

Χρήση εξωτερικής βιβλιοθήκης (ortools) για επίλυση του προβλήματος knapsack

```
from ortools.algorithms.python import knapsack_solver

n = 4 # πλήθος αντικειμένων
v = [3, 2, 4, 4] # αξίες αντικειμένων
w = [4, 3, 2, 3] # βάρη αντικειμένων
C = 6 # χωρητικότητα

solver = knapsack_solver.KnapsackSolver(
    knapsack_solver.SolverType.KNAPSACK_MULTIDIMENSION_BRANCH_AND_BOUND_SOLVER,
    "KnapsackExample")

solver.init(v, [w], [C])
computed_value = solver.solve()
print("Total value =", computed_value)

packed_items = []
total_weight = 0
for i in range(n):
    if solver.best_solution_contains(i):
        packed_items.append(i+1)
        total_weight += w[i]
print("Total weight:", total_weight)
print("Packed items:", packed_items)
```

Total value = 8
Total weight: 5
Packed items: [3, 4]

<https://developers.google.com/optimization/pack/knapsack>

Αναφορές

- The Algorithm Design Manual, Steven Skiena, 3rd edition