[](http://www.teilar.gr/)

**Αλγόριθμοι και Στοιχεία Πολυπλοκότητας.**

**Ενότητα 7:** Άπληστοι Αλγόριθμοι.

Διδάσκων: Ηλίας Κ Σάββας, Αναπληρωτής Καθηγητής.

Τμήμα Μηχανικών Πληροφορικής, Τεχνολογικής Εκπαίδευσης.

**Άδειες χρήσης.**

* Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons (C C). **Αναφορά δημιουργού (B Y), Μη εμπορική χρήση (N C), Μη τροποποίηση (N D), 3.0, Μη εισαγόμενο.**
* Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

[](http://creativecommons.org/licenses/by-nc-nd/3.0/deed.el)

**Χρηματοδότηση.**

* Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
* Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

[](http://www.edulll.gr/)

# Σκοποί ενότητας.

Ο αναγνώστης να μπορεί να:

1) χρησιμοποιεί την τεχνική της απληστίας,

2) επιλύει προβλήματα εφαρμόζοντας την παραπάνω τεχνική.

# ΠΕΡΙΕΧΌΜΕΝΑ ΕΝΌΤΗΤΑΣ.

[Σκοποί ενότητας. 2](#_Toc368166210)

[ΠΕΡΙΕΧΌΜΕΝΑ ΕΝΌΤΗΤΑΣ. 3](#_Toc368166211)

[ΑΛΓΌΡΙΘΜΟΙ. 3](#_Toc368166212)

[ΠΡΟΓΡΆΜΜΑΤΑ. 3](#_Toc368166213)

[ΣΧΉΜΑΤΑ. 3](#_Toc368166214)

[ΠΊΝΑΚΕΣ 3](#_Toc368166215)

[7. ΆΠΛΗΣΤΟΙ ΑΛΓΌΡΙΘΜΟΙ. 4](#_Toc368166216)

[7.1 Προγραμματισμός εργασιών. 6](#_Toc368166217)

[7.2 “Knapsack Problem”. 9](#_Toc368166218)

# ΑΛΓΌΡΙΘΜΟΙ.

|  |  |
| --- | --- |
| ***Αλγόριθμος 1****: Το πρόβλημα της επιστροφής ρέστων.* | 8 |
| ***Αλγόριθμος 2****: Χρονοπρογραμματισμός εργασιών.* | 2 |

# ΠΡΟΓΡΆΜΜΑΤΑ.

|  |  |
| --- | --- |
| ***Πρόγραμμα 1****: Το πρόβλημα της επιστροφής ρέστων.* | 6 |
| ***Πρόγραμμα 2****: Χρονοπρογραμματισμός εργασιών.* | 7 |
| ***Πρόγραμμα 3****: Knapsack problem – Αναδρομικός αλγόριθμος.* | 6 |
| ***Πρόγραμμα 4****: Knapsack problem – Άπληστος αλγόριθμος.* | 8 |

# ΣΧΉΜΑΤΑ.

|  |  |
| --- | --- |
| ***Σχήμα 7.1****: Χρονοπρογραμματισμός εργασιών.* | 4 |
| ***Σχήμα 7.2****: Επιλογή εργασιών προς εξυπηρέτηση.* | 2 |
| ***Σχήμα 7.3****: Knapsack problem.* | 2 |
| ***Σχήμα 7.4****: Μία λύση του knapsack problem.* | 3 |

# ΠΊΝΑΚΕΣ

|  |  |
| --- | --- |
| ***Πίνακας 7.1****: Παράδειγμα του knapsack problem.* | 6 |

# 7. ΆΠΛΗΣΤΟΙ ΑΛΓΌΡΙΘΜΟΙ.

Πολλές φορές, η τεχνική του δυναμικού προγραμματισμού, είναι εντελώς ασύμφορη για κάποια προβλήματα βελτιστοποίησης, και αυτό γιατί ψάχνει όλες τις πιθανές λύσεις, επιζητώντας την βέλτιστη. Σε αντίθεση με αυτή την τεχνική, είναι πιθανώς καλό σε κάποιες περιπτώσεις, αντί να αναζητά κανείς την συνολικά βέλτιστη λύση, να αναζητά την τοπικά βέλτιστη η οποία μπορεί να οδηγήσει (μπορεί και όχι), στην συνολική βέλτιστη. Αυτή η τεχνική αλγορίθμων ονομάζεται «απληστία» (greedy algorithms), διότι κάθε φορά επιλέγει με καθαρά τοπικά κριτήρια, μία λύση η οποία μπορεί να οδηγήσει στην βέλτιστη. Αυτή η μέθοδος χρησιμοποιείται τακτικότατα στην καθημερινή ζωή, με πιο χαρακτηριστικό παράδειγμα το πρόβλημα των «ρέστων». Όταν κάποιος επιχειρεί να επιστρέψει ρέστα, αντί να υπολογίσει τον ολικά βέλτιστο αριθμό νομισμάτων, αρχίζει με το μεγαλύτερο νόμισμα που μπορεί να επιστρέψει, και συνεχίζει με αυτόν τον τρόπο (*Αλγόριθμος 1* και *Πρόγραμμα 1*). Σε αντίθεση με την τεχνική του δυναμικού προγραμματισμού, η οποία εξαρτάται από τις λύσεις των επιμέρους υπό-προβλημάτων, και κινείται από κάτω προς τα πάνω (bottom up), οι άπληστοι αλγόριθμοι εξαρτώνται από τις επιλογές που έχουν κάνει μέχρι τώρα, για να συνεχίσουν (top down). Φυσικά, δεν είναι δυνατή η λύση όλων των προβλημάτων βελτιστοποίησης με αυτή τη τεχνική, αλλά ενδείκνυται για κάποια από αυτά.

***Αλγόριθμος 1****: Το πρόβλημα της επιστροφής ρέστων.*

1: Δεδομένα: Πίνακας *C*8 διαθέσιμων νομισμάτων (200 - 100 50 - 20 - 10 - 5 - 2 - 1), και *L* πίνακας κερμάτων που απαιτούνται.

2: Ακέραιος αριθμός: Συνάρτηση Υπολογισμού ρέστων (ακέραιος Ν).

3: /\* Το Ν αναπαριστά το ποσό που πρέπει να επιστραφεί \*/.

4: Ακέραιοι: i 🡨 0, x, *S* 🡨 0.

5: /\* To x αναπαριστά το εκάστοτε νόμισμα, και x το σύνολο που έχει υπολογισθεί. Το i αναπαριστά το σύνολο των νομισμάτων που θα χρειαστούν \*/.

6: Εφόσον (*S* < *N*),

7: Επέλεξε το μεγαλύτερο νόμισμα x από τον πίνακα *C,* τέτοιο ώστε να μην

υπερβαίνει το υπολειπόμενο ποσό N -s,

8: i 🡨 i + 1,

9: S 🡨 S + x,

10: Τέλος Εφόσον.

11: Επέστρεψε το i,

12: Τέλος αλγόριθμου «Υπολογισμού ρέστων».

***Πρόγραμμα 1****: Το πρόβλημα της επιστροφής ρέστων.*

#include <stdio.h>

int c[8] = {200, 100, 50, 20, 10, 5, 2, 1}; /\* Κέρματα \*/,

int L[100]; /\* Πίνακας κερμάτων που θα επιστραφούν σαν ρέστα \*/,

int resta(int);

main()

{

int N; /\* Επιστρεφόμενο ποσό \*/,

int p; /\* Πλήθος επιστρεφόμενων νομισμάτων \*/,

int i;

int s = 0;

printf("\n Εισαγωγή του επιστρεφόμενου ποσού (σε λεπτά):");

scanf("%d", &N);

p = resta(N);

for (i=0; i<p; i++) {

printf("\n %5d: %d", i+1, L[i]);

s += L[i];

}

printf("\n\n Χρειάστηκαν %d για το σύνολο του επιστρεφόμενου ποσού %d λεπτά

\n", p, s);

}

/\* Συνάρτηση υπολογισμού ρέστων \*/,

int resta(int n)

{

int i=0, j;

int x, s=0;

while (s < n) {

j = -1;

while (j<7 && (x = c[++j]) > n -s);

L[i++] = x;

s += x;

}

return i;

}

Γενικά, ένα πρόβλημα που επιδέχεται σαν λύση έναν άπληστο αλγόριθμο, εμφανίζει τα ακόλουθα χαρακτηριστικά:

***Ιδιότητα Άπληστης Επιλογής***: Η ολικά βέλτιστη λύση είναι δυνατή να παραχθεί, κατόπιν μίας και μοναδικής τοπικά βέλτιστης επιλογής, και προχωρώντας σε επόμενη κίνηση. Δηλαδή, είναι περιττή η διερεύνηση υπό-προβλημάτων και σύνθεση των λύσεών τους.

***Βέλτιστη Υποδομή***: Οι άπληστοι αλγόριθμοι δρουν επαγωγικά, και αποδεικνύεται πως η άπληστη επιλογή παρέχει συνολική βέλτιστη λύση, (αφού παράγει την βέλτιστη λύση του κάθε μοναδικού υπό-προβλήματος που παράγεται).

## 7.1 Προγραμματισμός εργασιών.

Έστω ένα σύνολο εργασιών , οι οποίες εκτελούνται από μία μηχανή (πχ προγράμματα που πρέπει να εκτελεσθούν σε έναν υπολογιστή). Η κάθε εργασία *t*i ορίζεται από το χρόνο που απαιτεί για να εκτελεσθεί, και χαρακτηρίζεται από τον χρόνο έναρξης *s*i, και το χρόνο λήξης *e*i, ενώ η μηχανή μπορεί να εκτελεί μία εργασία κάθε φορά. Το ζητούμενο είναι να βρεθεί το πλήθος των εργασιών, που μπορούν να εκτελεσθούν από την μηχανή εντός συγκεκριμένου χρόνου. Αυτό αποτελεί ένα θεμελιώδες πρόβλημα, όπου πολλά άλλα αντίστοιχα ανάγονται σε αυτό, όπως για παράδειγμα πόσα μαθήματα (μέγιστος αριθμός), μπορούν να γίνουν σε μία αίθουσα, εάν είναι γνωστά τα μαθήματα και οι ώρες έναρξης και λήξης τους.

Επειδή υπάρχει μόνο ένας διαθέσιμος πόρος (πχ μηχανή, αίθουσα, και τα λοιπά), είναι πολύ πιθανόν να μην μπορέσουν να εξυπηρετηθούν όλες οι εργασίες. Έστω για παράδειγμα ότι  με χρόνους έναρξης και λήξης {(1, 4), (2, 5), (0, 7), (5, 9), (2, 10), (7, 11), (10, 13), (2, 14)}, αντίστοιχα (*Σχήμα 7.1*). Είναι προφανές ότι τουλάχιστον μία από τις εργασίες *t*1 και *t*2 δεν πρόκειται να εξυπηρετηθούν. Το ζητούμενο είναι να προγραμματισθούν όσο τον δυνατόν περισσότερες.

***Σχήμα 7.1****: Χρονοπρογραμματισμός εργασιών.*

Εφόσον πρέπει να μεγιστοποιηθεί ο αριθμός των εργασιών που θα εξυπηρετηθούν, είναι εύκολο το συμπέρασμα, ότι δεν συμφέρει να προγραμματίζονται χρονοβόρες εργασίες, αλλά εργασίες με μικρό χρόνο εκτέλεσης. Αυτή η διαπίστωση οδηγεί στην εξής άπληστη στρατηγική: Επαναληπτική επιλογή της εργασίας, με τον μικρότερο χρόνο εξυπηρέτησης (*e*i -*s*i), και προγραμματισμός της, αρκεί να μην αλληλεπιδρά με την προηγούμενη επιλεγείσα εργασία.

Ένας λοιπόν άπληστος αλγόριθμος που βασίζεται σε αυτή την στρατηγική, θα μπορούσε να είναι ο εξής (*Αλγόριθμος 2*). Υποτίθεται ότι το πλήθος των εργασιών είναι *N*, και ότι στην μεταβλητή pr αποθηκεύεται η προηγούμενη εργασία που έχει ήδη προγραμματισθεί. Επίσης, πρέπει να ταξινομηθεί το σύνολο των εργασιών με βάση το χρόνο λήξης τους, έτσι ώστε πάντα να επιλέγεται σαν πρώτη εργασία, η πρώτη του νέου ταξινομημένου πλέον συνόλου εργασιών. Τέλος, σε ένα πίνακα, έστω *Α*, αποθηκεύονται οι εργασίες που δρομολογούνται προς εξυπηρέτηση.

***Αλγόριθμος 2****: Χρονοπρογραμματισμός εργασιών.*

1: Ταξινόμηση όλων των εργασιών με κλειδί τον χρόνο λήξης τους, έτσι ώστε .

2: i 🡨 1,

3: *A*i 🡨 i,

4: pr 🡨 1.

5: Για k 🡨2 μέχρι *Ν.*

6: Εάν (sk ≥ epr). Τότε,

7: i 🡨 i +1,

8: *A*i 🡨 k,

9: pr 🡨 k,

10: Τέλος Εάν.

11: Τέλος Για (k).

12: Επέστρεψε τον πίνακα *A.*

13: Τέλος αλγόριθμου «Χρονοπρογραμματισμός εργασιών».

Το *Πρόγραμμα 2* υλοποιεί τον *Αλγόριθμο 2*.

***Πρόγραμμα 2****: Χρονοπρογραμματισμός εργασιών.*

#include <stdio.h>

/\* Προγραμματισμός Εργασιών \*/,

#define N 10

typedef struct {

int arxi;

int telos;

} Task;

Task task[N]; /\* Πίνακας εργασιών \*/,

int D[N]; /\* Πίνακας δραστηριοτήτων \*/,

int task\_sch(int);

main()

{

int n; /\*

Ταξινόμηση εργασιών (αύξουσα) ως προς τον χρόνο λήξης \*/,

for (i=0; i<n-1; i++)

for (j=n-1; j>i; j--)

if (task[j].telos < task[j-1].telos) {

temp = task[j];

task[j] Πλήθος εργασιών \*/,

int i, j;

int p; /\* Πλήθος επιλεγέντων εργασιών \*/,

Task temp;

/\* Εισαγωγή πλήθους εργασιών και χρόνων \*/,

printf("\n Πλήθος Εργασιών (<%d):", N);

scanf("%d", &n);

while (n > N)

scanf("%d", &n);

for (i=0; i<n; i++) {

printf("\n Εισαγωγή %d εργασίας (χρόνοι έναρξης και λήξης):", i+1);

scanf("%d %d", &task[i].arxi, &task[i].telos);

}

/\* = task[j-1];

task[j-1] = temp;

}

printf("\n\n ΤΑΞΙΝΟΜΗΜΈΝΕΣ ΕΡΓΑΣΊΕΣ \n\n");

for (i=0; i<n; i++)

printf("\n %d: %d -%d", i+1, task[i].arxi, task[i].telos);

p = task\_sch(n);

printf("\n\n ΑΠΟΤΕΛΈΣΜΑΤΑ \n");

for (i=0; i<=p; i++)

printf("\n %d Εργασία : %d", i+1, D[i]+1);

}

/\* Συνάρτηση χρονοπρογραμματισμού \*/,

int task\_sch(int n)

/\* Ενημερώνει τον πίνακα δραστηριοτήτων, και επιστρέφει το πλήθος των εργασιών που προγραμματίστηκαν \*/,

{

int i, k=0;

int pr; /\* Προηγούμενη προγραμματισμένη εργασία \*/,

D[0] = 0;

pr = 0;

for (i=1; i<n; i++)

if (task[i].arxi >= task[pr].telos) {

D[++k] = i;

pr = i;

}

return k;

}

Εάν στο πρόγραμμα εισαχθούν οι εργασίες του παραδείγματος, η λύση φαίνεται στο *Σχήμα 7.2*.

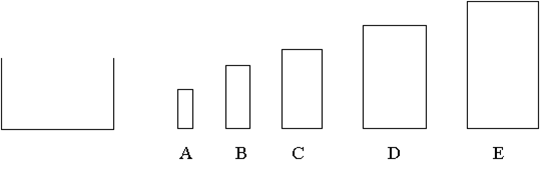
***Σχήμα 7.2****: Επιλογή εργασιών προς εξυπηρέτηση.*

## 7.2 “Knapsack Problem”.

Ένα πολύ ενδιαφέρον πρόβλημα, το οποίο εμφανίζεται και με πάρα πολλές παραλλαγές, είναι το λεγόμενο “knapsack problem”: Ας υποτεθεί ότι ένας κλέφτης παραβιάζει ένα θησαυροφυλάκιο, το οποίο περιέχει *N* αντικείμενα διαφορετικού μεγέθους αλλά και αξίας. Ο κλέφτης έχει ένα σάκο, περιορισμένου φυσικά μεγέθους, και το ζητούμενο είναι ποιά αντικείμενα πρέπει να πάρει, ώστε να έχουν την μεγαλύτερη δυνατή αξία. Για παράδειγμα, έστω ότι υπάρχουν 5 αντικείμενα με μεγέθη και αξίες όπως στον *Πίνακα 7.1*, και *Σχήμα 7.3*.

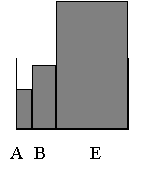
***Πίνακας 7.1****: Παράδειγμα του knapsack problem.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Αντικείμενα Θησαυροφυλακίου** | **1** | **2** | **3** | **4** | **5** |
| **Αντικείμενο** | A | B | C | D | E |
| **Μέγεθος** | 3 | 4 | 7 | 8 | 9 |
| **Αξία** | 4 | 5 | 10 | 11 | 13 |



***Σχήμα 7.3****: Knapsack problem.*

Το ζητούμενο είναι να επιλεγούν αντικείμενα με τέτοιο τρόπο, ώστε και να χωρούν στον σάκο, ενώ ταυτόχρονα, και η αξία του να είναι η όσο το δυνατόν μεγαλύτερη (παράδειγμα στο *Σχήμα 7.4*). Βέβαια, ένας τρόπος, είναι να ελεγχθούν όλοι οι δυνατοί συνδυασμοί, όμως αυτό οδηγεί σε λύσεις εκθετικού χρόνου, και άρα απαγορευτικές, ιδιαίτερα για μεγάλες τιμές. Η λύση αυτού του προβλήματος είναι ιδιαίτερα σημαντική, γιατί πάρα πολλές εφαρμογές οδηγούν σε ανάλογα προβλήματα, όπως για παράδειγμα, ο τρόπος που θα φορτωθεί ένα πλοίο ή αεροπλάνο, ο τρόπος που θα τοποθετηθούν αντικείμενα σε μία αποθήκη, και πολλά άλλα. Ένας αναδρομικός αλγόριθμος, θα προσέγγιζε το πρόβλημα ως εξής: Για κάθε αντικείμενο που θα επέλεγε, θα εξέταζε όλους τους πιθανούς συνδυασμούς, και τέλος θα κρατούσε την βέλτιστη λύση. Η πολυπλοκότητα της μεθόδου είναι της τάξης  - (πλήθος \* χωρητικότητα), δηλαδή ιδιαίτερα «ακριβή» για μεγάλα μεγέθη. Ο αναδρομικός αυτός αλγόριθμος περιγράφεται στο *Πρόγραμμα 3*.



***Σχήμα 7.4****: Μία λύση του knapsack problem.*

***Πρόγραμμα 3****: Knapsack problem – Αναδρομικός αλγόριθμος.*

#include <stdio.h>

#define N 5

/\* Μην εκτελέσετε αυτό το πρόγραμμα για μεγάλο N και x \*/,

/\* Ο χρόνος εκτέλεσης αυξάνει εκθετικά \*/,

typedef struct {

int size;

int val;

} Item;

Item items[N] ;

int knap(int);

main()

{

int i;

int megisto;

int x; /\* Χωρητικότητα \*/,

for (i=0; i<N; i++) {

printf("\n Εισαγωγή μεγέθους – τιμής του %d αντικειμένου:", i);

/\* Η εισαγωγή με αύξουσα σειρά τάξης μεγέθους \*/,

scanf("%d %d", &items[i].size, &items[i].val);

}

printf("\n Χωρητικότητα σάκου =");

scanf("%d", &x);

/\* Προτεινόμενα μεγέθη του *Πίνακα 7.1* και χωρητικότητα έως 40 \*/,

megisto = knap(x);

printf("\n Μέγιστο κέρδος = %d \n", megisto);

}

/\* Υπολογισμός βέλτιστου \*/,

int knap(int megethos);

{

int i, space, meg, t;

for (i=0, meg=0; i<N; i++)

if ((space = megethos -items[i].size) >= 0)

if ((t = knap(space) + items[i].val) > meg)

meg = t;

return meg;

}

Βέβαια, ένας άπληστος αλγόριθμος δεν θα λειτουργούσε ποτέ με αυτόν τον τρόπο. Κατ’ αρχήν, θα επέλεγε το αντικείμενο με την μεγαλύτερη δυνατή αξία, και θα συνέχιζε με αυτόν τον τρόπο, μέχρι να γεμίσει ο σάκος ή μέχρι να μην μπορεί να βρει άλλο αντικείμενο, που να χωράει στον σάκο. Αυτό σημαίνει, ότι ή άπληστη προσέγγιση του προβλήματος μπορεί να είναι σαφώς ταχύτερη, αλλά δεν βρίσκει πάντα την βέλτιστη λύση. Ο άπληστος αλγόριθμος του προβλήματος αυτού, δίνεται στο *Πρόγραμμα 4*.

***Πρόγραμμα 4****: Knapsack problem – Άπληστος αλγόριθμος.*

#include <stdio.h>

#define N 5

typedef struct {

int size;

int val;

} Item;

Item items[N] ;

int V[N]; /\* Πίνακας που θα κρατάει τα αντικείμενα που εισήχθησαν στον σάκο \*/,

int knap(int);

main()

{

int i, p;

int megisto = 0;

int x; /\* Χωρητικότητα \*/,

for (i=0; i<N; i++) {

printf("\n Εισαγωγή μεγέθους – τιμής του %d αντικειμένου:", i);

/\* Η εισαγωγή με αύξουσα σειρά τάξης μεγέθους \*/,

scanf("%d %d", &items[i].size, &items[i].val);

}

printf("\n Χωρητικότητα =");

scanf("%d", &x);

p = knap(x);

for (i=0; i<=p; i++) {

printf("\n Αντικείμενο: %d", V[i]+1);

megisto += items[V[i]].val;

}

printf("\n Μέγιστο = %d \n", megisto);

}

/\* Υπολογισμός βέλτιστου \*/,

int knap(int megethos)

{

int i, U, k =-1;

for (i=0; i<megethos; i++)

V[i] = 0;

U = megethos;

I = N-1;

while (i >= 0)

if ( items[i].size <= U) {

V[++k] = i;

U = U -items[i].size;

}

else

i--;

return k;

}

Τέλος έβδομης ενότητας.

[](http://creativecommons.org/licenses/by-nc-nd/3.0/deed.el)[](http://www.edulll.gr/)