ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΔΥΤΙΚΗΣ ΕΛΛΑΔΑΣ

ΠΡΩΗΝ ΤΜΗΜΑ ΕΦΑΡΜΟΓΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΣΤΗΝ ΔΙΟΙΚΗΣΗ ΚΑΙ ΣΤΗΝ ΟΙΚΟΝΟΜΙΑ

ΤΜΗΜΑ ΔΙΟΙΚΗΣΗΣ ΕΠΙΧΕΙΡΗΣΕΩΝ (ΠΑΤΡΑ)

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΑΝΩΤΕΡΕΣ ΤΕΧΝΙΚΕΣ ΑΝΑΛΥΣΗΣ ΚΑΙ ΣΧΕΔΙΑΣΗΣ ΑΛΓΟΡΙΘΜΩΝ: ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

ADVANCED TECHNIQUES O
ANALYSIS AND DESIGN O
ALGORITHMS: DYNAMI
PROGRAMMIN

ΠΑΤΡΑ 2015

άζαρος Κήττας

πιβλέπων Καθηγητής

ρ. Πιερρακέας Χρήστος

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ

Το θέμα της παρούσας πτυχιακής εργασίας αφορά στις τεχνικές ανάλυσης και σχεδίασης αλγορίθμων και πιο συγκεκριμένα στον δυναμικό προγραμματισμό. Στο αρχικό σκέλος γίνεται προσέγγιση στην θεωρία των αλγορίθμων αλλά και την μελέτη τους, (γνωστή και ως «algorithmics») καθώς και στις πιο γνωστές τεχνικές σχεδίασης αλγορίθμων. Στην συνέχεια επικεντρώνεται στην τεχνική του δυναμικού προγραμματισμού και την σχέση του με την επιχειρησιακή έρευνα και τον χώρο της βελτιστοποίησης, καθώς και στα πλεονεκτήματα-μειονεκτήματα του. Τέλος γίνεται επίλυση κάποιων προβλημάτων με την βοήθεια του δυναμικού προγραμματισμού που αναδεικνύουν τον τρόπο εφαρμογής του.

SYNOPSIS

The subject of this dissertation is the analysis and design techniques of algorithms and namely, dynamic programming. In the initial part we approach the science of algorithms and their study (also known as “algorithmics”) and also the most commonly known algorithm design techniques. Consequently we focus on the dynamic programming technique and its relation with operations research and the field of optimization, as well as its advantages and disadvantages. Finally, with the help of dynamic programming we solve a few problems that highlight its method of application.

ΚΕΦΑΛΑΙΟ 1: ΘΕΩΡΙΑ ΑΛΓΟΡΙΘΜΩΝ

Οι αλγόριθμοι παίζουν καθοριστικό ρόλο τόσο στην επιστήμη όσο και στην εφαρμογή της πληροφορικής. Η αναγνώριση του γεγονότος αυτού έχει οδηγήσει στην εμφάνιση ενός σημαντικού αριθμού βοηθημάτων και εγχειριδίων πάνω στο ζήτημα. Κατά κανόνα, ακολουθούν ένα από τους δυο παρακάτω τρόπους παρουσίασης αλγορίθμων.

Ο πρώτος τρόπος κατατάσσει τους αλγόριθμους σύμφωνα με το είδος του προβλήματος (αλγόριθμοι ταξινόμησης, ευρετικοί, γραφημάτων κλπ). Το πλεονέκτημα αυτής της προσέγγισης είναι ότι επιτρέπει την άμεση σύγκριση της αποδοτικότητας διαφορετικών αλγορίθμων για το ίδιο πρόβλημα. Το μειονέκτημα αυτής της προσέγγισης είναι ότι επικεντρώνεται στο είδος του προβλήματος αφήνοντας σε δεύτερη μοίρα τις τεχνικές σχεδίασης αλγορίθμου.

Ο δεύτερος τρόπος οργανώνει την παρουσίαση με επίκεντρο τις τεχνικές σχεδίασης αλγορίθμων. Σε αυτήν την οργάνωση ομαδοποιούνται αλγόριθμοι από όλο τον χώρο της πληροφορικής με μόνο κριτήριο το αν έχουν την ίδια σχεδιαστική προσέγγιση. Επικρατεί η άποψη ότι αυτή η οργάνωση είναι καταλληλότερη για την βασική περιήγηση στον σχεδιασμό και την ανάλυση αλγορίθμων (Levitin A., 2012, σελ. xix).

1.1 Ορισμός και ανάλυση αλγορίθμων

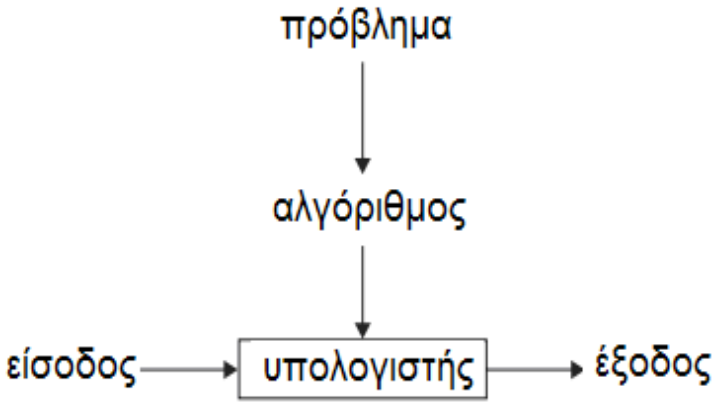
1.1.1 Η έννοια του αλγορίθμου

Σύμφωνα με τον Levitin A.(2012, σελ. 3) , η μελέτη των αλγορίθμων είναι επιτακτική ανάγκη για κάποιον που σκοπεύει να ασχοληθεί σε βάθος με την πληροφορική. Αυτό συμβαίνει γιατί πρέπει να γνωρίζουμε ένα βασικό αριθμό σημαντικών αλγορίθμων από τους διάφορους τομείς της πληροφορικής. Επιπρόσθετα θα πρέπει να έχουμε την ικανότητα να σχεδιάζουμε νέους αλγόριθμους και να αναλύουμε την αποδοτικότητά τους. Η θεωρητική μελέτη των αλγορίθμων που συχνά αποκαλείται algorithmics έχει

αναγνωριστεί ως ακρογωνιαίος λίθος της επιστήμης της πληροφορικής. Ο David Harel στο βιβλίο του με τίτλο «*Algorithmics: the spirit of Computing*» αναφέρει τα εξής:

«Η μελέτη των αλγορίθμων (Algorithmics) είναι κάτι παραπάνω από ένα παρακλάδι της πληροφορικής. Είναι ο πυρήνας της πληροφορικής και δικαίως μπορούμε να πούμε ότι σχετίζεται με τις περισσότερες επιστήμες , την διοίκηση και την τεχνολογία. Αλλά ακόμα και αν δεν είμαστε μαθητές στον χώρο της πληροφορικής, υπάρχουν επιτακτικοί λόγοι για τη μελέτη αλγορίθμων».

Για να το θέσουμε πιο απλά, τα προγράμματα των υπολογιστών δεν θα υπήρχαν χωρίς τους αλγόριθμους. Και με τις εφαρμογές των υπολογιστών να έχουν γίνει απαραίτητες σε σχεδόν όλους τους τομείς της επαγγελματικής αλλά και της προσωπικής μας ζωής, η μελέτη των αλγορίθμων έχει καταστεί αναγκαία για όλο και περισσότερους ανθρώπους. Άλλος ένας λόγος για την μελέτη αλγορίθμων είναι η χρησιμότητά τους στην ανάπτυξη δεξιοτήτων ανάλυσης. Οι αλγόριθμοι αποτελούν ξεχωριστό είδος επίλυσης προβλημάτων, καθώς δεν είναι μόνο απαντήσεις αλλά διαδικασίες λήψης αποφάσεων καθορισμένες με ακρίβεια. Φυσικά η ακρίβεια που εμφύτως επιβάλλεται από την αλγοριθμική σκέψη περιορίζει τα είδη των προβλημάτων που μπορούν να λυθούν με έναν αλγόριθμο. Βάσει των παραπάνω, μπορούμε να αποδώσουμε τον ακόλουθο ορισμό: Αλγόριθμος είναι μια ακολουθία από σαφείς οδηγίες για την επίλυση ενός προβλήματος, δηλαδή, για την απόκτηση μιας απαιτούμενης εξόδου για κάθε πιθανή είσοδο σε ένα πεπερασμένο χρονικό διάστημα. Ο ορισμός αυτός μπορεί να παρουσιαστεί με ένα απλό διάγραμμα (1.1) :

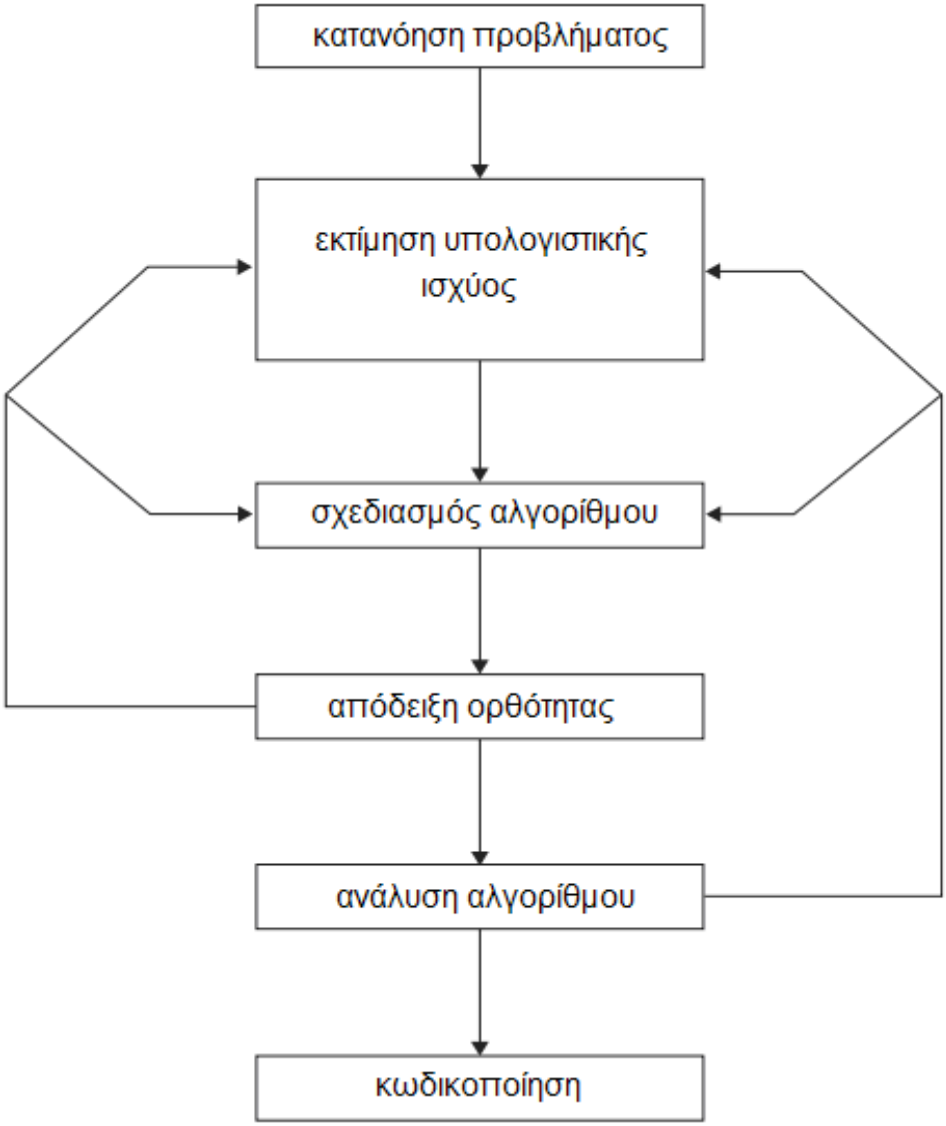


Διάγραμμα 1.1 – Σχηματική αναπαράσταση αλγορίθμου

Η αναφορά στις «οδηγίες» στον ανωτέρω ορισμό συνεπάγεται ότι υπάρχει μια οντότητα ικανή να κατανοήσει και να εφαρμόσει τις παραπάνω οδηγίες. Αυτό το αποκαλούμε «υπολογιστή».

1.1.2 Σχεδιασμός αλγορίθμων

Έχοντας ως βάση το ίδιο βιβλίο (σελ.10) θα μελετήσουμε τα απαραίτητα βήματα για τον σχεδιασμό και την ανάλυση ενός αλγορίθμου (διάγραμμα 1.2).



Διάγραμμα 1.2 – Ανάλυση αλγορίθμου

α) Κατανόηση του προβλήματος

Το πρώτο αναγκαίο βήμα που πρέπει να λάβει χώρα πριν το σχεδιασμό ενός αλγορίθμου είναι η απόλυτη κατανόηση του προβλήματος που μας δόθηκε. Διαβάζοντας την περιγραφή του προβλήματος προσεκτικά, συγκεντρώνουμε τις απορίες μας και αν υπάρχουν αμφιβολίες σχετικές με το πρόβλημα, διατυπώνουμε μικρά παραδείγματα «εκτέλεσης με το χέρι» μέχρι να μας λυθούν οι όποιες απορίες. Η είσοδος σε ένα αλγόριθμο καθορίζει ένα στιγμιότυπο του προβλήματος που λύνει αυτός ο αλγόριθμος. Είναι πολύ σημαντικό να καθορίζεται ακριβώς το σύνολο των στιγμιοτύπων που θα αντιμετωπίσει ο αλγόριθμος. Αν δεν το καταφέρουμε αυτό, ο αλγόριθμος μπορεί να δουλεύει για μια πλειάδα στιγμιοτύπων αλλά θα αποτυγχάνει σε κάποιες «οριακές» τιμές. Στο σημείο αυτό πρέπει να τονιστεί ιδιαίτερα ότι σωστός αλγόριθμος δεν είναι αυτός που λειτούργει για τις περισσότερες τιμές εισόδου, αλλά αυτός που λειτουργεί για όλες τις νόμιμες τιμές εισόδου. Εν κατακλείδι, το στάδιο αυτό δεν πρέπει να παραλείπεται στην διαδικασία επίλυσης αλγοριθμικών προβλημάτων διότι υπάρχει περίπτωση δαπάνησης χρόνου για επανασχεδιασμό τους.

β) Εκτίμηση της υπολογιστικής ισχύος

Μόλις κατανοήσουμε πλήρως το πρόβλημα θα πρέπει να διαπιστώσουμε τις δυνατότητες της διαθέσιμης υπολογιστικής συσκευής που θα εκτελεστεί ο αλγόριθμος. Η συντριπτική πλειοψηφία των αλγορίθμων που χρησιμοποιούνται σήμερα είναι σχεδιασμένοι για να προγραμματίζουν έναν ηλεκτρονικό υπολογιστή που μοιάζει αρκετά με την μηχανή του von Neumann – μια αρχιτεκτονική που διατυπώθηκε από τον διακεκριμένο μαθηματικό John von Neumann (1903-1957), σε συνεργασία με τον A. Burks και τον H. Goldstone, το 1946. Η ουσία αυτής της αρχιτεκτονικής βρίσκεται στο λεγόμενο μηχάνημα τυχαίας προσπέλασης (random-access machine, RAM).

Η κεντρική θεωρία του είναι ότι οι οδηγίες εκτελούνται η μια μετά την άλλη με μια διαδικασία την κάθε φορά. Αναλόγως, οι αλγόριθμοι που σχεδιαστήκαν για να εκτελούνται σε τέτοια μηχανήματα ονομάζονται *σειριακοί αλγόριθμοι* (sequential algorithms). Η κεντρική ιδέα του μοντέλου RAM δεν ισχύει για κάποιους νεότερους υπολογιστές που μπορούν να εκτελέσουν παράλληλες λειτουργίες. Οι αλγόριθμοι που μπορούν να εκμεταλλευτούν αυτήν την ικανότητα ονομάζονται *παράλληλοι*

αλγόριθμοι (parallel algorithms). Όπως και να έχει όμως η μελέτη των παραδοσιακών τεχνικών σχεδίασης και ανάλυσης αλγορίθμων συμφώνα με το μοντέλο RAM παραμένει ο ακρογωνιαίος λίθος των algorithmics για το εγγύς μέλλον .

Στη σημερινή εποχή δεν πρέπει να ανησυχούμε για την ταχύτητα και το μέγεθος της μνήμης του υπολογιστή που χρησιμοποιούμε, γιατί οι περισσότεροι επιστήμονες του κλάδου της πληροφορικής προτιμούν να μελετούν αλγόριθμους ανεξάρτητα από τα τεχνικά χαρακτηριστικά ενός συγκεκριμένου υπολογιστή. Παρ' όλα αυτά υπάρχουν προβλήματα που είναι πολυσύνθετα από την φύση τους, ή απαιτείται η επεξεργασία τεράστιων όγκων δεδομένων ή ο παράγοντας χρόνος είναι καθοριστικός. Σε τέτοιες περιπτώσεις είναι επιτακτικό να γνωρίζουμε την ταχύτητα και την ποσότητα μνήμης του μηχανήματος.

γ) Σχεδιασμός αλγορίθμων

Με τον όρο σχεδιασμός αλγορίθμων εννοούμε τη γενική προσέγγιση της αλγοριθμικής επίλυσης προβλημάτων που είναι εφαρμόσιμη σε ένα ευρύ φάσμα προβλημάτων από διαφόρους χώρους της πληροφορικής. Η εκμάθηση των τεχνικών σχεδίασης είναι υψίστης σημασίας για τους δύο παρακάτω λογούς:

i. Παρέχουν καθοδήγηση για τον σχεδιασμό αλγορίθμων για νέα προβλήματα πχ προβλήματα για τα όποια δεν υπάρχουν γνωστοί ικανοποιητικοί αλγόριθμοι.
ii. Οι αλγόριθμοι είναι ο ακρογωνιαίος λίθος της πληροφορικής. Κάθε επιστήμη επιδιώκει την καθιέρωση της βασικής αρχής της και η πληροφορική δεν αποτελεί εξαίρεση. Με τις τεχνικές σχεδίασης αλγορίθμων επιτυγχάνεται ο διαχωρισμός των αλγορίθμων συμφώνα με την σχεδιαστική φιλοσοφία τους και έτσι γίνεται με φυσικό τρόπο η ταξινόμηση και η μελέτη τους.

δ) Απόδειξη ορθότητας αλγορίθμου

Όταν ένας αλγόριθμος έχει καθοριστεί πρέπει να αποδείξουμε την ορθότητα του. Δηλαδή πρέπει να αποδείξουμε ότι ο αλγόριθμος αποδίδει ένα ζητούμενο αποτέλεσμα για κάθε θεμιτή είσοδο σε ένα πεπερασμένο χρονικό διάστημα. Για παράδειγμα, η ορθότητα του αλγορίθμου του Ευκλείδη για τον υπολογισμό του μεγίστου κοινού διαιρέτη πηγάζει από την ορθότητα της εξίσωσης $gcd(m,n) = gcd(\mathrm{n}, \mathrm{m} \bmod \mathrm{n})$ με την

απλή παρατήρηση ότι ο δεύτερος ακέραιος γίνεται μικρότερος σε κάθε επανάληψη του αλγορίθμου και ότι ο αλγόριθμος σταματά όταν ο δεύτερος ακέραιος γίνεται 0.

Για κάποιους αλγόριθμους η απόδειξη ορθότητας είναι πολύ εύκολη και για κάποιους άλλους αρκετά σύνθετη. Μια κοινή τεχνική για την απόδειξη ορθότητας είναι να χρησιμοποιήσουμε μαθηματική επαγωγή γιατί οι επαναλήψεις ενός αλγορίθμου παρέχουν μια φυσική ακολουθία των βημάτων που απαιτείται για τέτοιες αποδείξεις. Η επιλογή τυχαίων τιμών εισόδου δεν μπορεί να εξασφαλίσει την ορθότητα του αλγορίθμου, καθώς για να αποδείξουμε ότι ένας αλγόριθμος είναι εσφαλμένος χρειαζόμαστε έστω και μια τιμή εισόδου για την οποία αποτυγχάνει. Η έννοια της ορθότητας για προσεγγιστικούς αλγόριθμους είναι λιγότερο απλή σε σχέση με τους ακριβείς αλγόριθμους. Για ένα προσεγγιστικό αλγόριθμο θα πρέπει να αποδείξουμε ότι το σφάλμα που παράγεται από τον αλγόριθμο δεν ξεπερνά ένα προκαθορισμένο όριο.

ε) Ανάλυση αλγορίθμου

Από τους βασικότερους στόχους μας είναι οι αλγόριθμοι που θα δημιουργήσουμε να έχουν κάποιες ιδιότητες. Μετά την ορθότητα η πιο σημαντική ιδιότητα με διάφορα είναι αυτή της αποδοτικότητας. Για την ακρίβεια υπάρχουν δυο ειδών αποδοτικότητες αλγορίθμου (ανακτήθηκε στις 25 Μαΐου 2014 από το http://www.slideshare.net/ankkatiyar/time-and-space-complexity):

- η χρονική αποδοτικότητα που δείχνει ποσό γρήγορα εκτελείται ο αλγόριθμος,
- η αποδοτικότητα χώρου που δείχνει πόση παραπάνω μνήμη χρησιμοποιεί.

Μπορούμε να ορίσουμε σαν χρονική αποδοτικότητα τον συνολικό αριθμό των βημάτων που απαιτείται για την επίλυση ενός προβλήματος (συναρτήσει του μεγέθους του) ενώ ως αποδοτικότητα χώρου το μέγεθος υπολογιστικής μνήμης που απαιτείται κατά τη διάρκεια της εκτέλεσης ενός προγράμματος σε συνάρτηση με το πλήθος των τιμών εισόδου. Η διαφορά ανάμεσα στα δυο παραπάνω είδη αποδοτικότητας είναι ότι ο χώρος (πχ μνήμη RAM) μπορεί να επαναχρησιμοποιηθεί. Η ανάλυση ως προς την αποδοτικότητα είναι χρήσιμη για να αξιολογήσουμε αν ένας αλγόριθμος χρησιμοποιεί ένα λογικό αριθμό πόρων για την επίλυση ενός προβλήματος και επίσης για να συγκρίνουμε την αποδοτικότητα μεταξύ

διαφορετικών αλγορίθμων. Σύμφωνα με τον Leiss L. (2007), αμφότερες είναι σημαντικές αλλά δεδομένου του ρυθμού αύξησης της μνήμης των σημερινών Η/Υ, η αποδοτικότητα του χώρου χάνει την σημασία της.

Ένα άλλο επιθυμητό χαρακτηριστικό αλγορίθμου είναι η απλότητα. Σε αντίθεση με την αποδοτικότητα που μπορούμε να την ορίσουμε με ακρίβεια και να την διερευνήσουμε με μαθηματική αυστηρότητα, η απλότητα είναι κάτι εντελώς υποκειμενικό. Σίγουρα, οι απλούστερα σχεδιασμένοι αλγόριθμοι είναι πιο εύκολοι στην κατανόηση και στην κωδικοποίηση τους με αποτέλεσμα τα προγράμματα που δημιουργούνται να έχουν λιγότερα σφάλματα και ταυτόχρονα να είναι πιο γρήγορα στην εκτέλεση τους. Ένας σύνθετος αλγόριθμος δεν μπορεί να μας εγγυηθεί καλύτερο αποτέλεσμα από έναν απλούστερο αλγόριθμο.

στ) Κωδικοποίηση αλγορίθμου

Οι περισσότεροι αλγόριθμοι προορίζονται για να ενσωματωθούν εν τέλει σε πρόγραμμα, με την προϋπόθεση ότι έχει αποδειχθεί η ορθότητα ενός προγράμματος σε υπολογιστή με μαθηματική αυστηρότητα, αλλιώς το πρόγραμμα δεν μπορεί να θεωρηθεί σωστό. Στην παρούσα πτυχιακή εργασία, η κωδικοποίηση των αλγορίθμων θα πραγματοποιηθεί με χρήση της γλώσσας προγραμματισμού Java.

1.2 Τεχνικές σχεδίασης αλγορίθμων

Οι μέθοδοι λύσης ενός προβλήματος που προκύπτουν από την ανάλυση του, οδηγούν στην σχεδίαση ενός αλγορίθμου που συνιστά την ακολουθία βημάτων που πρέπει να ακολουθηθούν για να επιλυθεί το πρόβλημα. Κάθε τεχνική σχεδίασης έχει τα δικά της χαρακτηριστικά και τις δικές τις ιδιαιτερότητες. Τρεις από αυτές τις τεχνικές είναι η «διαίρει και βασίλευε», ο δυναμικός προγραμματισμός και η άπληστη μέθοδος (greedy).

1.2.1 Η μέθοδος «Διαίρει και Βασίλευε»

Η μέθοδος «Διαίρει και Βασίλευε» εφαρμόζεται διασπώντας ένα πρόβλημα σε περισσότερα από δύο μικρότερα υπο-προβλήματα του ίδιου προβλήματος και στην συνέχεια στην σύνθεση των επιμέρους λύσεων για την δημιουργία της καθολικής λύσης του προβλήματος. Βέβαια σε κάθε ένα από τα επιμέρους προβλήματα πρέπει αναδρομικά να εφαρμοστεί ή ίδια μέθοδος μέχρι να καταλήξει σε προβλήματα τάξης μεγέθους ένα ή το πολύ δύο όπου η λύση είναι εύκολη. Περιληπτικά, τα βήματα επίλυσης ενός προβλήματος με αυτή την μέθοδο είναι τα ακόλουθα (Σάββας Η., 2005):

1. Διαίρει (διάσπαση του αρχικού προβλήματος σε επιμέρους μικρότερα προβλήματα),

2. Βασίλευε (λύνουμε το κάθε επιμέρους πρόβλημα με αναδρομική χρήση της ίδιας τακτικής),

3. Σύνθεσε (συνθέτουμε τις επιμέρους λύσεις σε μία καθολική λύση του αρχικού προβλήματος)

1.2.2 Η μέθοδος Δυναμικού Προγραμματισμού

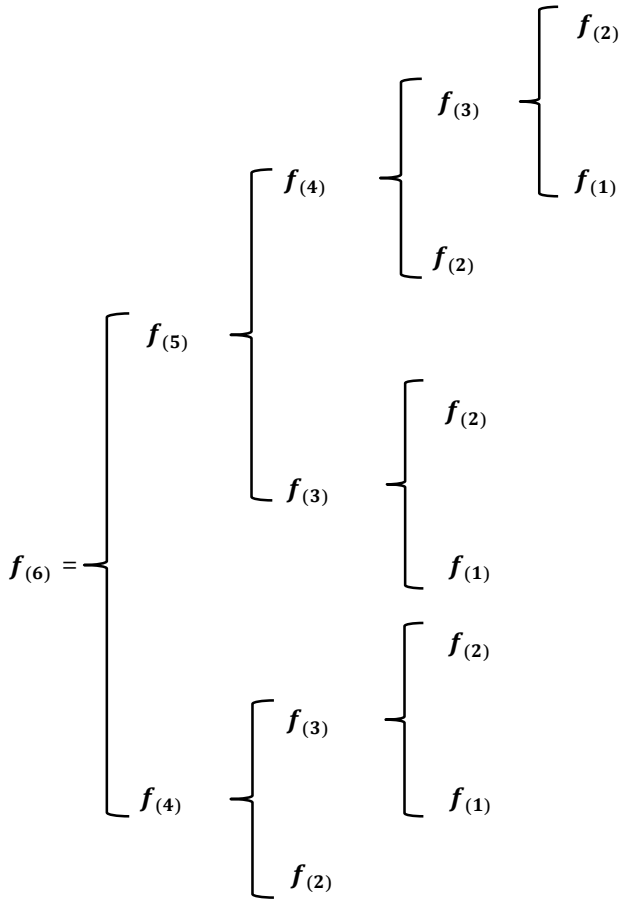
Σύμφωνα με τον Δρ. Σάββα (2005) η τεχνική του δυναμικού προγραμματισμού, όπως και η «διαίρει και βασίλευε», λύνει ένα πρόβλημα συνδυάζοντας λύσεις σε προβλήματα με την διαφορά ότι η τεχνική του δυναμικού προγραμματισμού εφαρμόζεται όταν τα προβλήματα δεν είναι ανεξάρτητα. Η μέθοδος του δυναμικού προγραμματισμού προϋποθέτει μία αναδρομική επίλυση προβλημάτων αλλά με μία από κάτω προς τα πάνω (bottom-up) εκτίμηση των λύσεων. Πολλές φορές συναντάμε τα ιδία προβλήματα και για την αποφυγή επαναλαμβανόμενης επίλυσης του ιδίου προβλήματος, η λύση του κάθε προβλήματος αποθηκεύεται σε πινάκα για την επαναχρησιμοποίηση τους. Η συνολική βέλτιστη λύση χρησιμοποιώντας την μέθοδο αυτή προκύπτει από τον συνδυασμό ή σύνθεση των βέλτιστων λύσεων των υπο-προβλημάτων. Αυτή η τεχνική χρησιμοποιείται κυρίως σε προβλήματα

βελτιστοποίησης τα οποία σχετίζονται με πολλές λύσεις αλλά και μια τιμή κόστους για το κάθε ένα από αυτά. Ένα τυπικό παράδειγμα αποτελεί η επίλυση της συνάρτησης Fibbonaci η οποία ορίζεται ως εξής:

$$f_n = \begin{cases} f_{(n-1)} + f_{(n-2)}, \forall n > 2 \\ \\ f_{(1)} = f_{(2)} = f_{(1)} \end{cases}$$

Δηλαδή η $f_{(6)}$ αναλύεται ως εξής:

`



Σε αυτήν την περίπτωση μια από πάνω προς τα κάτω προσέγγιση της λύσης θα οδηγούσε στον υπολογισμό του $f_{(4)}$ δύο φορές και του $f_{(3)}$ τρεις φορές ώστε τελικά να υπολογίσουμε το $f_{(6)}$ γεγονός που κάνει αυτήν την προσέγγιση μη ικανοποιητική. Αντιθέτως, με την τεχνική του δυναμικού προγραμματισμού οι λύσεις των υπο-προβλημάτων υπολογίζονται μία φορά και αποθηκεύονται σε ένα πίνακα για τις φορές που θα τις χρειαστούμε. Ο δυναμικός προγραμματισμός χρησιμοποιείται

συνήθως σε προβλήματα βελτιστοποίησης, δηλαδή για την εύρεση μίας βέλτιστης λύσης ενώ ταυτόχρονα η λύση αυτή υπόκειται και σε κάποιους περιορισμούς. Η τεχνική μοιάζει αρκετά με την μέθοδο του «διαίρει και βασίλευε» σε σχέση με την διαίρεση του προβλήματος σε μικρότερα και απλούστερα υπο-προβλήματα αλλά είναι δυνατόν να μην είναι του ίδιου τύπου. Τα βασικά συστατικά των αλγορίθμων δυναμικού προγραμματισμού είναι τα:

1) Διαίρεση του προβλήματος σε μικρότερα και απλούστερα μέρη.

2) Αποθήκευση των λύσεων σε ένα πίνακα. Αυτό γίνεται γιατί πολλές αν όχι όλες από τις λύσεις των υπό-προβλημάτων πρόκειται να ξαναχρησιμοποιηθούν και δεν ενδείκνυται το να επιλυθούν ξανά και ξανά.

3) Συνδυασμός των λύσεων των μικρών κομματιών του προβλήματος ώστε να οδηγούν σε λύσεις μεγαλύτερων κομματιών μέχρι να επιλυθεί όλο το πρόβλημα.

1.2.3 Η μέθοδος Greedy

Όπως αναφέρει στο βιβλίο του ο Ζάχος Ε. (2005), τα περισσότερα προβλήματα που επιδέχονται λύση με την χρησιμοποίηση ενός greedy (άπληστου) αλγορίθμου απαιτούν να βρεθεί ένα υποσύνολο των δεδομένων εισόδου το όποιο να ικανοποιεί κάποιους περιορισμούς και να είναι βέλτιστο ως προς κάποιο αντικειμενικό κριτήριο. Κάθε υποσύνολο του συνόλου δεδομένων εισόδου που ικανοποιεί τους περιορισμούς λέγεται εφικτή λύση.

Η απαίτηση ενός προβλήματος με την χρήση της μεθόδου Greedy είναι να βρεθεί όχι απλώς μια εφικτή λύση, αλλά η βέλτιστη λύση, δηλαδή, μια λύση η οποία να ελαχιστοποιεί ή να μεγιστοποιεί μια δεδομένη αντικειμενική συνάρτηση. Συνήθως είναι εύκολο να βρεθεί μια εφικτή λύση, όχι όμως και μια βέλτιστη. Ένας άπληστος αλγόριθμος κάνει πάντα την επιλογή που φαίνεται καλύτερη τη δεδομένη χρονική στιγμή. Δηλαδή κάνει την τοπικά βέλτιστη επιλογή με την ελπίδα ότι αυτή η επιλογή θα τον οδηγήσει στην ολικά βέλτιστη λύση. Οι greedy αλγόριθμοι δεν δίνουν πάντα βέλτιστες λύσεις. Επειδή ο άπληστος αλγόριθμος δουλεύει με ένα κριτήριο τοπικής

βελτιστότητας, για να δίνει βέλτιστη λύση για κάποιο πρόγραμμα, θα πρέπει στο συγκεκριμένο πρόβλημα να ισχύει ότι η τοπικά βέλτιστη επιλογή είναι πράγματι και ολικά βέλτιστη. Ο αλγόριθμος ξεκινά με την κενή λύση και σε κάθε βήμα την μεγαλώνει με τέτοιο τρόπο ώστε η επιλογή που κάνει να είναι τοπικά η καλύτερη (ανάλογα με το χειρισμό της αντικειμενικής συνάρτησης) και συγχρόνως εφικτή (ικανοποιεί τους περιορισμούς).

Τα δεδομένα είναι διατεταγμένα συμφώνα με μια διαδικασία επιλογής, η όποια βασίζεται σε κάποιο κανόνα βελτιστοποίησης. Ο κανόνας μπορεί να είναι η αντικειμενική συνάρτηση, μπορεί όμως και όχι. Σε κάθε πρόβλημα μπορεί να βρεθούν πολλοί κανόνες βελτιστοποίησης οι όποιοι να μην οδηγούν όλοι σε βέλτιστη λύση.

1.3 Επίλυση προβλημάτων

Όπως γνωρίζουμε το πλήθος των προς εξέταση προβλημάτων, τόσο στην καθημερινότητα μας, όσο και στην πληροφορική είναι άπειρο. Ωστόσο υπάρχουν μερικά είδη προβλημάτων που είτε λόγω της πρακτικής τους σημασίας, είτε λόγω κάποιων συγκεκριμένων χαρακτηριστικών, αποκτούν ένα εξαιρετικά ενδιαφέρον αντικείμενο έρευνας. Οι πιο σημαντικοί τύποι προβλημάτων είναι (Levitin A. 2012, σελ. 19):

- ταξινόμηση
- αναζήτηση
- αναγνώριση συμβολοσειράς
- προβλήματα γραφημάτων
- γεωμετρικά προβλήματα
- αριθμητικά προβλήματα

1.3.1 Το πρόβλημα της ταξινόμησης

Ως πρόβλημα ταξινόμησης, ο Levitin (2012, σελ. 19) ορίζει την αναδιάταξη των αντικειμένων σε μια οποιαδήποτε λίστα με μια μη-μειωτική σειρά. Φυσικά για να έχει νόημα αυτό το πρόβλημα η φύση των αντικειμένων στη λίστα πρέπει να επιτρέπουν την αναδιάταξη. Πρακτικά συνήθως χρειάζεται να ταξινομήσουμε λίστες με νούμερα, με γράμματα του αλφάβητου, με συμβολοσειρές και κυρίως λίστες με αρχεία παρόμοια με αυτά που διατηρούν τα σχολεία για τους μαθητές τους, βιβλιοθήκες για την συλλογή τους και εταιρείες για τους υπάλληλους τους.

Έστω ότι μας δίνεται ένα πολύ-σύνολο Σ που περιλαμβάνει το σύνολο των ακεραίων αριθμών. Ταξινόμηση του Σ ορίζεται η επιβολή μιας διάταξης στα στοιχεία του συνόλου. Επομένως, ένας αλγόριθμος ταξινόμησης δέχεται ως είσοδο το πολύ-σύνολο Σ και με βάση μια δεδομένη σχέση διάταξης δίνει στην έξοδο τα στοιχεία του συνόλου διατεταγμένα. Οι αλγόριθμοι που επιλύουν το πρόβλημα αυτό είναι:

- *ο αλγόριθμος ταξινόμησης σωρού (heap sort)*
- *ο αλγόριθμος ταξινόμησης με συγχώνευση (merge sort)*
- *ο αλγόριθμος γρήγορης ταξινόμησης (quick sort)*
- *ο αλγόριθμος φυσαλίδας (bubble sort)*
- *ο αλγόριθμος ενθετικής ταξινόμησης (insertion sort)*
- *ο αλγόριθμος ταξινόμησης με επιλογή (selection sort)*

Η αποδοτικότητα των παραπάνω αλγόριθμων εξαρτάται από το εκάστοτε πρόβλημα το οποίο επιλύουν.

1.3.2 Το πρόβλημα της αναζήτησης

Το πρόβλημα της αναζήτησης έχει να κάνει την εύρεση μια δεδομένης τιμής που ονομάζεται κλειδί αναζήτησης (search key) σε ένα δεδομένο σύνολο (ή πολύ-σύνολο που επιτρέπει πολλά στοιχεία να έχουν την ίδια τιμή). Υπάρχουν αρκετοί αλγόριθμοι αναζήτησης για να διαλέξουμε (Levitin A. 2012*)*:

- *ο αλγόριθμος γραμμικής αναζήτησης*
- *ο αλγόριθμος δυαδικής αναζήτησης*
- *ο αλγόριθμος αναζήτησης με παρεμβολή*

Οι παραπάνω αλγόριθμοι έχουν ειδική σημασία στις πρακτικές εφαρμογές γιατί είναι απαραίτητοι για αποθήκευση και ανάκτηση πληροφοριών από μεγάλες βάσεις δεδομένων. Για την αναζήτηση ισχύει επίσης ότι δεν υπάρχει ένας αλγόριθμος που είναι βέλτιστος για όλες τις περιπτώσεις. Μερικοί λειτουργούν γρηγορότερα από άλλους αλλά έχουν μεγαλύτερες απαιτήσεις σε μνήμη. Μερικοί είναι πολύ γρήγοροι αλλά εφαρμόσιμοι μόνο σε αποθηκευμένους πίνακες κλπ. Σε αντίθεση με τους αλγόριθμους ταξινόμησης δεν υπάρχει πρόβλημα σταθερότητας. Συγκεκριμένα σε εφαρμογές που τα δεδομένα αλλάζουν συχνά σε σχέση με τον αριθμό των αναζητήσεων, η αναζήτηση πρέπει να γίνεται έχοντας υπόψη την σύνδεση με άλλες δυο λειτουργίες: την πρόσθεση και την αφαίρεση από το σύνολο των δεδομένων. Σε τέτοιες περιπτώσεις, δομές δεδομένων και αλγόριθμοι πρέπει να επιλεγούν για να υπάρχει μια ισορροπία ανάμεσα στις απαιτήσεις κάθε διαδικασίας.

1.3.3 Το πρόβλημα της αναγνώρισης συμβολοσειρών

Τις τελευταίες δεκαετίες η ραγδαία αύξηση των εφαρμογών που ασχολούνται με μη αριθμητικά δεδομένα έχει εντείνει το ενδιαφέρον των ερευνητών για τη βελτιστοποίηση αλγορίθμων που χειρίζονται συμβολοσειρές. Συμβολοσειρά ονομάζεται μια ακολουθία από αλφαριθμητικούς χαρακτήρες. Συμβολοσειρές με

ιδιαίτερο ενδιαφέρον είναι αυτές του κειμένου που αποτελούνται από γράμματα, νούμερα και ειδικούς χαρακτήρες, δυαδικές συμβολοσειρές (που αποτελούνται από 0 και 1) και ακολουθίες γονίδιων που μπορούμε να αναπαραστήσουμε με συμβολοσειρά από το αλφάβητο τεσσάρων χαρακτήρων (A, C, G, T).

Σύμφωνα με τον Levitin (2012), οι αλγόριθμοι αναζήτησης συμβολοσειράς μπορούν να χωριστούν σε τέσσερις κατηγορίες:

- *αλγόριθμοι που βασίζονται στη σύγκριση χαρακτήρων (Brute Force, Knuth-Morris-Pratt, Simon, Boyer Moore)*
- *αλγόριθμοι που χρησιμοποιούν suffix αυτόματα (Reverse Factor, Turbo Reverse Factor)*
- *αλγόριθμοι που χρησιμοποιούν τον παραλληλισμό των χειριστών των bit στις ψηφιολέξεις για να εκτελούν παράλληλα πολλές λειτουργίες (Shift-Or, Shift-and και BNDM)*
- *αλγόριθμοι που χρησιμοποιούν τεχνικές κατακερματισμού (Harrison, Karp-Rabin).*

1.3.4 Το πρόβλημα των γραφημάτων

Μια από τις παλιότερες και πιο ενδιαφέρουσες περιοχές στην μελέτη των αλγορίθμων είναι οι αλγόριθμοι γραφημάτων. Γενικότερα, ένα γράφημα μπορεί να θεωρηθεί ως μια συλλογή από σημεία που αποκαλούνται κορυφές, μερικές από τις οποίες συνδέονται από ευθύγραμμα τμήματα που ονομάζονται ακμές. Τα γραφήματα αποτελούν ένα ενδιαφέρον ζήτημα για μελέτη και για θεωρητικούς αλλά και πρακτικούς λογούς. Μπορούν να χρησιμεύσουν στην αναπαράσταση μιας μεγάλης ποικιλίας εφαρμογών , συμπεριλαμβανόμενων των μεταφορών, της επικοινωνίας, των κοινωνικών και οικονομικών δικτύων, χρονοδιαγραμμάτων και παιγνίων. Συγκεκριμένα η μελέτη διαφόρων τεχνικών και κοινωνικών πτυχών του διαδικτύου είναι ένας ενεργός τομέας των τρεχουσών ερευνών με εμπλεκομένους επιστήμονες του χώρου της πληροφορικής, οικονομολόγους και κοινωνιολόγους. Οι βασικοί

αλγόριθμοι γραφημάτων συμπεριλαμβάνουν αλγόριθμους γραφημάτων-διάσχισης (δηλαδή για παράδειγμα πώς μπορεί κάποιος να πάει σε όλα τα σημεία ενός δικτύου), αλγόριθμους συντομότερης-διαδρομής (πχ ποιά είναι η βέλτιστη διαδρομή ανάμεσα σε δυο πόλεις), και τοπολογική ταξινόμηση για τα γραφήματα με κατευθυνόμενες ακμές (σύνολο από διαδρομές με τις προϋποθέσεις τους συνεπείς ή αντιφατικές). Μερικά προβλήματα γραφημάτων είναι -υπολογιστικά- πολύ δύσκολα με πιο γνωστά παραδείγματα του περιοδεύοντος πωλητή και το πρόβλημα χρωματισμού του γραφήματος.

Το πρόβλημα του περιοδεύοντος πωλητή (Traveling Salesman Problem, TSP και το οποίο αναλύεται στο κεφάλαιο 3) είναι το πρόβλημα της εύρεσης της συντομότερης διαδρομής μέσω n πόλεων επισκεπτόμενος την κάθε πολύ ακριβώς μια φορά. Εκτός από την προφανή συσχέτιση με εφαρμογές σχεδιασμού διαδρομών, επίσης συναντάται η εφαρμογή του σε πλακέτες κυκλωμάτων, στην κατασκευή VLSI chip, σε ακτινογραφίες με κρυσταλλογραφία και στην γενετική μηχανική.

Το πρόβλημα χρωματισμού γραφήματος προσπαθεί να αναθέσει τον μικρότερο αριθμό χρωμάτων στις κορυφές του γραφήματος έτσι ώστε να μην υπάρχουν παρακείμενες κορυφές του ιδίου χρώματος. Το πρόβλημα συναντάται σε διάφορες εφαρμογές όπως στην οργάνωση εκδηλώσεων.

1.3.5 Γεωμετρικά προβλήματα

Οι γεωμετρικοί αλγόριθμοι (Levitin A. 2012) έχουν να κάνουν με γεωμετρικά αντικείμενα όπως σημεία, γραμμές και πολύγωνα. Οι αρχαίοι Έλληνες ενδιαφέρονταν πολύ για την ανάπτυξη διαδικασιών για την επίλυση μιας ευρείας κλίμακας γεωμετρικών προβλημάτων, συμπεριλαμβανομένων και των προβλημάτων κατασκευής απλών γεωμετρικών σχημάτων (τριγώνων, κύκλων κλπ) με έναν κανόνα και μια πυξίδα. Έτσι για περίπου 2000 χρόνια το έντονο ενδιαφέρον για γεωμετρικούς αλγόριθμους εξαφανίστηκε για να αναγεννηθεί στα χρόνια των υπολογιστών. Φυσικά σήμερα οι άνθρωποι ενδιαφέρονται για τους γεωμετρικούς αλγόριθμους αλλά με πολύ διαφορετικές εφαρμογές κατά νου, όπως τα γραφικά υπολογιστών, την

ρομποτική και την τομογραφία. Τα δύο κλασικότερα προβλήματα γεωμετρικού υπολογισμού είναι:

- το πρόβλημα “Closest Pair”. Το πρόβλημα των κοντινότερων σημείων εξετάζει τη μικρότερη δυνατή απόσταση μεταξύ δύο σημείων, έστω $A(x_1, y_1)$ και $B(x_2, y_2)$ από ένα σύνολο n σημείων. (Cormen T., 2009, σελ. 1039)

- η εύρεση κυρτών περιβλημάτων (αλγόριθμοι Graham και Jarvis). Κυρτό περίβλημα ενός συνόλου S από n σημεία είναι το ελάχιστο κυρτό πολύγωνο P τέτοιο ώστε κάθε ένα από τα n σημεία στο S βρίσκεται πάνω στο σύνορο του S ή στο εσωτερικό του S. (Cormen T., 2009, σελ. 1037)

1.3.6 Αριθμητικά προβλήματα

Τα αριθμητικά προβλήματα είναι προβλήματα που συνεπάγονται μαθηματικά αντικείμενα διαρκούς χαρακτήρα (επίλυση εξισώσεων και συστήματα εξισώσεων, υπολογισμός ορισμένων ολοκληρωμάτων κλπ). Η πλειοψηφία από αυτά τα μαθηματικά προβλήματα μπορούν να λυθούν μόνο προσεγγιστικά και αυτή η δυσκολία πηγάζει από το γεγονός ότι τέτοιου είδους προβλήματα απαιτούν χειρισμό πραγματικών αριθμών που μπορούν να αναπαρασταθούν σε ηλεκτρονικό υπολογιστή μόνο προσεγγιστικά. Επιπρόσθετα ένας μεγάλος αριθμός μαθηματικών πράξεων που πραγματοποιούνται κατά προσέγγιση και αναπαριστώνται μπορεί να οδηγήσει σε σφάλματα λόγω συσσώρευσης στρογγυλοποιήσεων σε σημείο που μπορεί να αλλοιώσει δραματικά την έξοδο που παράχθηκε από ένα φαινομενικώς σωστό αλγόριθμο. Πολλοί εξεζητημένοι αλγόριθμοι έχουν αναπτυχθεί τα τελευταία χρόνια και συνεχίζουν να παίζουν καθοριστικό παράγοντα σε πολλές επιστημονικές και μηχανολογικές εφαρμογές. Αυτές οι νέες εφαρμογές απαιτούν κυρίως αλγόριθμους για αποθήκευση πληροφορίας, ανάκτησης, μεταφοράς μέσω δικτύων και παρουσίασης σε χρηστές. Σαν αποτέλεσμα αυτής της επαναστατικής αλλαγής η αριθμητική ανάλυση έχει χάσει την πρώην δεσπόζουσα θέση τόσο στη βιομηχανία όσο και την επιστήμη των προγραμματιστών πληροφορικής. Από την άλλη, είναι πολύ σημαντικό για κάθε γνώστη της πληροφορικής να έχει τουλάχιστον μια

στοιχειώδη ιδέα σχετικά με τους αριθμητικούς αλγόριθμους. Ένα από τα πιο γνωστά παραδείγματα αριθμητικών αλγορίθμων είναι ο υπολογισμός της τετραγωνικής ρίζας με τη μέθοδο Newton.

ΚΕΦΑΛΑΙΟ 2: ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

2.1 Ιστορική αναφορά

Ο όρος Δυναμικός Προγραμματισμός όπως διατυπώνεται στο αντίστοιχο λήμμα της ιστοσελίδας www.wikipedia.org (ανακτήθηκε στις 28 Αυγούστου 2013 από το http://en.wikipedia.org/wiki/Dynamic_programming), αρχικά χρησιμοποιήθηκε την δεκαετία του 1940 από τον Richard Bellman στην προσπάθειά του να περιγράψει την διαδικασία επίλυσης προβλημάτων όπου απαιτείται η εύρεση διαδοχικών βέλτιστων αποφάσεων. Το 1953 ο όρος Δυναμικός Προγραμματισμός πήρε την σύγχρονη του έννοια και αναφέρεται συγκεκριμένα στην ένταξη απλούστερων αποφάσεων μέσα σε πιο σύνθετες αποφάσεις που πρέπει να ληφθούν για την επίλυση σύνθετων πολυσταδιακών προβλημάτων. Ο όρος μετέπειτα αναγνωρίστηκε από το Ινστιτούτο Ηλεκτρολόγων Μηχανολόγων σαν θέμα ανάλυσης συστημάτων και μηχανολογίας. Η συμβολή του Bellman συναντάται και στην εξίσωση του Bellman (Ljungqvist, L., Recursive Macroeconomic Theory, 2012) ως αποτέλεσμα του Δυναμικού Προγραμματισμού, η οποία επαναδιατυπώνει ένα πρόβλημα βελτιστοποίησης σε αναδρομική μορφή.

2.2 Γενικά χαρακτηριστικά ΔΠ

Ο δυναμικός προγραμματισμός αποτελεί μια μεθοδολογία λήψης αποφάσεων σε σύνθετα πολυσταδιακά αλληλοεξαρτώμενα προβλήματα, δηλαδή σε προβλήματα των οποίων η περιοχή εφαρμογής (π.χ. κόστος, απόσταση) επιτρέπει τον καταμερισμό του κύριου προβλήματος σε μικρότερα υπο-προβλήματα τα όποια αποκαλούμε στάδια. Η επίλυση προβλημάτων με την χρήση δυναμικού προγραμματισμού επιτυγχάνεται μέσω του προσδιορισμού του βέλτιστου συνδυασμού διαδοχικών αποφάσεων με στόχο την βελτιστοποίηση του ζητούμενου κριτηρίου. Πιο συγκεκριμένα, εντοπίζονται οι βέλτιστες λύσεις για τα υπο-προβλήματα και τις ανασυνθέτουμε σε

μια καθολική βέλτιστη λύση. Αξίζει να σημειωθεί ότι η μέθοδος καταμερισμού του κύριου προβλήματος σε (άνω των δυο) υπο-προβλήματα και η επίλυση τους δεν υπακούει σε κάποια κοινώς αποδεκτή τυποποιημένη μεθοδολογία, αντιθέτως κάθε σύστημα όντας διαφορετικό με τα δικά του χαρακτηριστικά απαιτεί επίλυση προσαρμοσμένη στις δικές του ανάγκες. Ο δυναμικός προγραμματισμός εφαρμόζεται κυρίως σε προβλήματα βελτιστοποίησης (Δημητρίου Τ., 2001). Σε ένα τέτοιο πρόβλημα μπορεί να υπάρχουν πολλές λύσεις. Κάθε λύση έχει μία τιμή και μας ενδιαφέρει να βρούμε τη λύση με τη βέλτιστη τιμή. Μία τέτοια λύση την ονομάζουμε βέλτιστη. Για παράδειγμα, ας θεωρήσουμε ότι μας δίνεται ένας χάρτης, μια πόλη Α, και μας ζητείται να βρούμε τις διαδρομές με τη μικρότερη απόσταση από την Α προς όλες τις άλλες τις πόλεις του χάρτη. Ως λύσεις θα μπορούσαν να θεωρηθούν οι διάφορες εναλλακτικές διαδρομές, αλλά ως βέλτιστη θα ήταν εκείνη η λύση στην οποία όλες οι διαδρομές θα ήταν ελάχιστες. Για την εφαρμογή δυναμικού προγραμματισμού σε τέτοιο πρόβλημα λοιπόν θα πρέπει να τηρούνται οι δυο παρακάτω προϋποθέσεις:

1. Βέλτιστη Υπό-δομή (Optimal Substructure)

Θεωρούμε ότι ένα πρόβλημα παρουσιάζει βέλτιστη υπό-δομή αν η βέλτιστη λύση περιέχει βέλτιστες λύσεις στα διάφορα υπο-προβλήματα. Δηλαδή αν η λύση κάποιου προβλήματος δεν είναι βέλτιστη τότε και η λύση στο αρχικό πρόβλημα δε θα είναι βέλτιστη. Αυτή η παρατήρηση οδηγεί αυτόματα και σε μία τεχνική ελέγχου αν το πρόβλημα παρουσιάζει βέλτιστη υπό-δομή. Αν η λύση ενός προβλήματος δεν είναι η βέλτιστη και αυτό επηρεάζει την τελική λύση τότε το δοθέν πρόβλημα παρουσιάζει βέλτιστη υπό-δομή. Σε αντίθετη περίπτωση πιθανόν ο δυναμικός προγραμματισμός να μην αποτελεί κατάλληλη μέθοδο επίλυσης.

2. Κοινά Προβλήματα

Ο αριθμός των υπο-προβλημάτων πρέπει να είναι μικρός, δηλαδή πολυωνυμικός στο μέγεθος της εισόδου. Ο αλγόριθμος συναντά τα ίδια υπο-προβλήματα ξανά και ξανά (όπως για παράδειγμα στους αριθμούς Fibonacci) τα οποία λύνει μία φορά, αποθηκεύει τις λύσεις σε κάποιο πίνακα και στη συνέχεια τις ανακαλεί με σταθερό κόστος για να συνθέσει τη γενική λύση.

2.3 Βελτιστοποίηση και επιχειρησιακή έρευνα

Ο Δρ. Βασίλειος Μούσας αναφέρει στο e-book "Εφαρμογές Βελτιστοποίησης και Επιχειρησιακής Έρευνας σε Προβλήματα Μηχανικών" (ανακτήθηκε στις 15 Οκτωβρίου 2013 από το http://users.teiath.gr/vmouss/ebooks/optimee/sections/section01_intro_i.), ότι ο όρος επιχειρησιακή έρευνα εμφανίστηκε λίγο πριν την αρχή του 2ου Παγκοσμίου Πολέμου, όταν ομάδες από επιστήμονες διαφορετικών ειδικοτήτων ασχολήθηκαν με την επιστημονική ανάλυση και το βέλτιστο σχεδιασμό στρατιωτικών επιχειρήσεων, όπως το σύστημα έγκαιρης προειδοποίησης (radar), η οργάνωση πτήσεων, η οργάνωση νηοπομπών, αποβάσεων, κλπ. Με την πάροδο του χρόνου, οι εταιρείες και οι βιομηχανίες άρχισαν να απλώνονται ολοένα και περισσότερο εντός και εκτός των συνόρων με αποτέλεσμα το ανθρώπινο δυναμικό και τα μηχανήματά τους να αυξάνονται με ραγδαίους ρυθμούς. Αποτέλεσμα αυτής της ανάπτυξης ήταν η επιτακτική ανάγκη της υποδιαίρεσης τους σε πολλά διαφορετικά τμήματα. Για παράδειγμα το τμήμα παραγωγής χωρίστηκε στα τμήματα αγοράς, συντήρησης, διακίνησης, ποιοτικού ελέγχου κλπ. Παράλληλα, δημιουργήθηκαν πολλά παραρτήματα, υποκαταστήματα και μονάδες παραγωγής σε απομακρυσμένα σημεία. Αποτέλεσμα της διασποράς των παραρτημάτων και της υποδιαίρεσης των τμημάτων ήταν η μείωση της αποτελεσματικότητας της διοίκησης. Παράγοντες που συντέλεσαν σε αυτό ήταν επίσης η εμφάνιση νέων πρώτων υλών και τεχνολογιών καθώς και οι νέες συνθήκες που επικρατούσαν την εποχή εκείνη. Έπρεπε λοιπόν οι επιχειρηματικές αποφάσεις να ξεφύγουν από τα πλαίσια της προσωπικής κρίσης και πείρας των στελεχών και να βασίζονται περισσότερο σε επιστημονικές μεθόδους, οπότε το έδαφος ήταν πρόσφορο για την εφαρμογή της επιχειρησιακής έρευνας και της βελτιστοποίησης γενικότερα.

Με τον όρο βελτιστοποίηση εννοούμε την επινόηση και εφαρμογή επιστημονικών μεθόδων για την επίλυση επιχειρησιακών προβλημάτων με σκοπό τον καλύτερο έλεγχο και τον βέλτιστο τρόπο λειτουργίας οργανωμένων συστημάτων ή διαδικασιών.

2.3.1 Τα στάδια της μεθόδου

Όπως αναφέρεται και στο e-book του Δρ. Βασίλειου Μούσα (http://users.teiath.gr/vmouss/ebooks/optimee/sections/section02_intro_ii.html), η αντιμετώπιση ενός προβλήματος όπου πρέπει να ληφθεί η βέλτιστη απόφαση περιλαμβάνει συνήθως τα εξής στάδια:

i. **Αναγνώριση και περιγραφή του προβλήματος**. Είναι αναγκαίο να εξεταστεί αν συνδέεται με άλλα προβλήματα και πρέπει να καθοριστούν οι στόχοι με αντικειμενικό τρόπο. Είναι το πιο σημαντικό βήμα γιατί οποιοδήποτε λάθος θα οδηγήσει σε αποτυχία τα επόμενα στάδια.
ii. **Καθορισμός των παραμέτρων του προβλήματος.** Σε αυτό το στάδιο εξετάζουμε ποιοι παράγοντες επηρεάζουν τη λύση και πως μπορούμε να τους μεταβάλουμε ώστε να έχουμε εναλλακτικές λύσεις.
iii. **Εντοπισμός των περιορισμών του προβλήματος**. Ποιοι είναι οι περιορισμοί ή τα όρια μέσα στα οποία μπορούμε να κινηθούμε.
iv. **Αναζήτηση λύσεων και επιλογή της βέλτιστης λύσης.** Αφού βρεθούν οι εφικτές λύσεις, επιλέγεται η βέλτιστη λύση, με βάση τον αντικειμενικό στόχο που θέσαμε στο πρώτο βήμα. Μετά την επιλογή του κατάλληλου μοντέλου και της κατασκευής του στην κατάλληλη μορφή, εφαρμόζονται οι απαραίτητες τεχνικές για να βρεθούν οι δυνατές λύσεις του. Από τις δυνατές ή εφικτές λύσεις θα προκύψει η βέλτιστη λύση η οποία θα ικανοποιεί αφ’ ενός τους περιορισμούς και αφ’ ετέρου θα πλησιάζει περισσότερο από όλες τις άλλες στον αντικειμενικό στόχο.
v. **Δοκιμή και υλοποίηση της βέλτιστης λύσης.** Στο τελευταίο στάδιο ελέγχεται η αξιοπιστία του μοντέλου και της βέλτιστης λύσης που μας δίνει και ταυτόχρονα χρησιμοποιούνται παλαιότερα δεδομένα με γνωστά αποτελέσματα. Αν το μοντέλο μας δώσει τα ίδια αποτελέσματα τότε μπορούμε να προχωρήσουμε στην εφαρμογή του.

2.3.2 Το μοντέλο του προβλήματος

Το μοντέλο είναι μια εξιδανικευμένη αναπαράσταση του πραγματικού συστήματος και αποτελείται από ένα σύνολο μαθηματικών σχέσεων που περιγράφουν τη κατάσταση (Μούσας Β., 2007). Συγκεκριμένα περιλαμβάνει:

- **Τις μεταβλητές,** δηλαδή τις ποσότητες που ελέγχουμε και μπορούμε να μεταβάλλουμε για να πετύχουμε το στόχο που έχουμε βάλει.
- **Τις παραμέτρους,** δηλαδή όλα τα άλλα δεδομένα που επηρεάζουν τη λύση του προβλήματος. Τέτοια δεδομένα είναι η τιμή ενός προϊόντος, η ταχύτητα ή ο χρόνος μεταφοράς του, η αναλογία ανάμιξης δυο υλικών κλπ.
- **Τους περιορισμούς** που πρέπει να πληρούν οι μεταβλητές και από τους οποίους προκύπτουν οι επιτρεπτές τιμές τους.
- **Τον αντικειμενικό στόχο** που έχουμε βάλει και ο οποίος δεν είναι πάντα μοναδικός αλλά μπορεί να αποτελείται από επί μέρους στόχους που πρέπει να επιτευχθούν. Συνήθως πρόκειται για μια συνάρτηση που πρέπει να πάρει μια μέγιστη ή ελάχιστη τιμή.

2.3.3 Τομείς εφαρμογής βελτιστοποίησης

Οι τομείς εφαρμογής της βελτιστοποίησης σχετίζονται με ένα ευρύ φάσμα προβλημάτων σε διάφορα πεδία της βιομηχανίας και της κοινωνίας για παράδειγμα σε επιχειρηματικούς και δημόσιους οργανισμούς, στο εμπόριο, σε υπηρεσίες κοινής ωφέλειας (ηλεκτρισμός, ύδρευση, τηλεπικοινωνίες), στα νοσοκομεία, στην παιδεία, σε χωροταξικές μελέτες κλπ.

Για παράδειγμα, αν μια βιομηχανία θέλει να αυξήσει τη παραγωγή της, θα της προταθούν πολλές διαφορετικές λύσεις. Ο υπεύθυνος προσωπικού θα ζητήσει προσλήψεις, ο μηχανολόγος νέα μηχανήματα, ο μηχανικός παραγωγής αναδιάρθρωση των διαδικασιών, ο αναλυτής βελτίωση των πληροφορικών συστημάτων, κλπ. Για να επιτευχθεί το καλύτερο αποτέλεσμα θα πρέπει να βρεθεί ο βέλτιστος συνδυασμός των παραπάνω λύσεων. Αλλά και από γενικότερη σκοπιά, σε κάθε μεγάλη επιχείρηση υπάρχουν ομάδες ενδιαφερομένων με διαφορετικούς στόχους, που επηρεάζουν τους συνολικούς στόχους της επιχείρησης, όπως: οι ιδιοκτήτες (κέρδος ή τζίρο), οι πελάτες

(καλή ποιότητα και τιμή), οι εργαζόμενοι στη παραγωγή (καλές αμοιβές και συνθήκες), οι πωλητές (πωλήσεις και έσοδα), το κράτος (φόροι και θέσεις εργασίας), κ.ά. Η ανάπτυξή της επιχείρησης θα πρέπει να συνδυάζει με Βέλτιστο τρόπο όλους αυτούς τους επιμέρους, και συχνά αντικρουόμενους, στόχους.

Εν κατακλείδι η βελτιστοποίηση δεν είναι η λύση για όλα τα προβλήματα. Όμως κατά την εξέλιξή της εντόπισε και έδωσε λύσεις σε γενικές περιοχές προβλημάτων, τα οποία εμφανίζονται σχεδόν σε όλους τους τομείς της ανθρώπινης δραστηριότητας.

2.4 Πλεονεκτήματα και μειονεκτήματα του ΔΠ

Ο δυναμικός προγραμματισμός αποτελεί μια αποδοτική μέθοδο επίλυσης ενός μεγάλου φάσματος προβλημάτων βελτιστοποίησης. Ταυτόχρονα όμως συνδυάζεται με κάποια μειονεκτήματα που εν δυνάμει μπορούν να καταστήσουν την εφαρμογή του αδύνατη ή ακατάλληλη. Στον πίνακα που ακολουθεί αναφέρονται επιγραμματικά κάποια από τα πλεονεκτήματα και τα μειονεκτήματα που χαρακτηρίζουν τον δυναμικό προγραμματισμό (ανακτήθηκε στις 22 Σεπτεμβρίου 2013 από το http://www.slideshare.net/paramalways/dynamic-programming).

ΠΛΕΟΝΕΚΤΗΜΑΤΑ
1) Μας επιτρέπει να αναπτύξουμε λύσεις για τα υπο-προβλήματα ενός μεγαλύτερου προβλήματος, οι οποίες είναι πιο εύκολο να διατηρηθούν και να εξεταστεί η ορθότητα τους. 2) Η αναδρομική διαδικασία δεν κάνει χρήση της αποθήκευσης σε αντίθεση με τον δυναμικό προγραμματισμό που αποθηκεύει προηγούμενες τιμές για την αποφυγή επαναλαμβανόμενων υπολογισμών. 3) Η διαδικασία διάσπασης ενός σύνθετου προβλήματος σε αλληλοσυνδεόμενα προβλήματα συχνά παρέχει ενόραση στην φύση του προβλήματος 4) Επειδή ο δυναμικός προγραμματισμός είναι μια προσέγγιση με στόχο την βελτιστοποίηση επιτρέπει ευελιξία στην εφαρμογή του σε ποικίλα προβλήματα

μαθηματικού προγραμματισμού. 5) Η υπολογιστική διαδικασία που ακολουθείται στον δυναμικό προγραμματισμό επιτρέπει εγγενώς μια ευαίσθητη ανάλυση που βασίζεται σε μεταβλητές κατάστασης και σε μεταβλητές με την μορφή σταδίων. 6) Επιτυγχάνει εξοικονόμηση υπολογιστικής ισχύος σε αντίθεση της ολοκληρωτικής απαρίθμησης.
ΜΕΙΟΝΕΚΤΗΜΑΤΑ
1) Απαιτείται μεγαλύτερη εξειδίκευση στην επίλυση προβλημάτων με την μέθοδο δυναμικού προγραμματισμού σε σχέση με άλλες μεθόδους επίλυσης 2) Έλλειψη γενικού αλγορίθμου όπως η μέθοδος simplex. Περιορίζει τους υπολογιστικούς κώδικες που είναι απαραίτητοι για ανέξοδες και ευρείες χρήσεις 3) Το μεγαλύτερο πρόβλημα είναι οι διαστάσεις του. Το πρόβλημα παρουσιάζεται όταν μια συγκεκριμένη εφαρμογή του χαρακτηρίζεται από πολλαπλά στάδια. Δημιουργεί πολλά προβλήματα για τις δυνατότητες του υπολογιστή και καταναλώνει πολύ χρόνο.

2.5 Διαφορές ΔΠ με «διαίρει και βασίλευε»

Ο δυναμικός προγραμματισμός αποτελεί μια τεχνική «διαίρει και βασίλευε» καθώς λύνει τα προβλήματα διασπώντας τα σε άλλα μικρότερα και ταυτόχρονα απλούστερα υπο-προβλήματα. Παρ’ όλα αυτά, υπάρχουν δυο ουσιώδεις διαφορές που στοιχειοθετούν τη διαφορά των δύο τεχνικών:

- Τα υπο-προβλήματα στο «διαίρει και βασίλευε» είναι ανεξάρτητα, ενώ στο δυναμικό προγραμματισμό είναι επικαλυπτόμενα. Αυτό σημαίνει ότι στο δυναμικό προγραμματισμό θα πραγματοποιηθούν περισσότερες από μία αναδρομικές κλήσεις με τις ίδιες παραμέτρους (δηλ. για την επίλυση του ίδιου υποπροβλήματος), ενώ στο «διαίρει και βασίλευε» κάθε αναδρομική κλήση

γίνεται με διαφορετικές παραμέτρους (δηλ. γίνεται για διαφορετικό υπο-πρόβλημα).

- Κάθε υπο-πρόβλημα στο «διαίρει και βασίλευε» είναι ένα σημαντικά μικρότερο στιγμιότυπο του αρχικού προβλήματος ενώ στον δυναμικό προγραμματισμό κάθε υπο-πρόβλημα είναι συνήθως ένα ελαφρώς μικρότερο στιγμιότυπο του αρχικού.

ΚΕΦΑΛΑΙΟ 3: ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ ΠΕΡΙΟΔΕΥΟΝΤΟΣ ΠΩΛΗΤΗ

3.1 Ορισμός προβλήματος περιοδεύοντος πωλητή

Το πρόβλημα του Περιοδεύοντος Πωλητή (TSP) αναφέρεται σε ένα πωλητή ο οποίος ξεκινάει μια περιοδεία σε διάφορες πόλεις με σκοπό να πουλήσει τα προϊόντα του. Θα πραγματοποιήσει το ταξίδι του με στόχο να επισκεφτεί την κάθε πόλη ακριβώς μια φορά πριν την επιστροφή στο σπίτι του. Δεδομένων των αποστάσεων μεταξύ των πόλεων μας ζητείται η εύρεση της σειράς με την οποία θα επισκεφτεί τις εν λόγω πόλεις με γνώμονα την ελάχιστη διανυθείσα διαδρομή.

Ορίζουμε τις πόλεις από $1, \ldots\ldots, n$, με την έδρα του πωλητή να είναι η πόλη 1. Επίσης ορίζουμε ως $D = (d_{i,j})$ τις αποστάσεις μεταξύ των πόλεων. Στόχος μας είναι να σχεδιάσουμε την διαδρομή που ξεκινά και τελειώνει στην πόλη 1 και που περιλαμβάνει την κάθε πόλη από μια φορά διανύοντας όμως την ελάχιστη συνολική απόσταση (Dasgupta S., 2006).

Το πρόβλημα αυτό εκτός από ότι μπορεί να φαίνεται δύσκολο να λυθεί χωρίς υπολογισμούς με το χέρι, ο βαθμός δυσκολίας του ανεβαίνει καθώς οι πόλεις αυξάνονται. Για να είμαστε ακριβείς, το πρόβλημα του περιοδεύοντος πωλητή αποτελεί ένα από τα πιο περιβόητα υπολογιστικά προβλήματα και αποτελεί δύσκολη διαδικασία ακόμα και για τους Η/Υ. Υπήρξαν αρκετές προσπάθειες επίλυσης του, πολλές αποτυχημένες και κάποιες μερικώς επιτυχημένες, κατά την πορεία της αλματώδους προόδου των αλγορίθμων και της θεωρίας της πολυπλοκότητας. Η πιο δυσάρεστη είδηση για το πρόβλημα του περιοδεύοντος πωλητή είναι ότι μάλλον είναι αδύνατο να μπορέσει να λυθεί σε πολυωνυμικό χρόνο.

Χρησιμοποιώντας την μέθοδο brute force για να αξιολογήσουμε κάθε πιθανή διαδρομή ώστε να βρούμε την βέλτιστη καταλήγουμε στο συμπέρασμα ότι από την στιγμή που υπάρχουν $(n-1)!$ ενδεχόμενα, με αυτήν την μέθοδο απαιτείται $O(n!)$

χρόνος. Στην συνεχεία θα δούμε ότι ο δυναμικός προγραμματισμός αποφέρει πολύ συντομότερα την βέλτιστη λύση, όχι όμως σε πολυωνυμικό χρόνο.

3.2 Μοντελοποίηση και μαθηματική επίλυση προβλήματος περιοδεύοντος πωλητή

Ξεκινώντας την επίλυση του προβλήματος, σύμφωνα με το βιβλίο “Algorithms” του S. Dasgupta (2006), σαν πρώτο βήμα θα πρέπει να χωρίσουμε το κύριο πρόβλημα σε μικρότερα υπο-προβλήματα. Στην περίπτωσή μας το πιο προφανές υπο-πρόβλημα είναι το αρχικό στάδιο της διαδρομής. Έστω ότι ο πωλητής ξεκινά από την πόλη 1 όπως είναι το ζητούμενο , επισκέπτεται κάποιες πόλεις και τώρα βρίσκεται στην πόλη j. Η απαραίτητη πληροφορία σε αυτό το σημείο ώστε να συνεχίσουμε αυτήν την επιμέρους διαδρομή, είναι να γνωρίζουμε την πόλη j διότι αυτό θα καθορίσει ποιές πόλεις μας συμφέρει να επισκεφτούμε στη συνέχεια. Επίσης πρέπει να γνωρίζουμε ποιές πόλεις έχουμε επισκεφτεί ήδη ώστε να μην τις επισκεφτούμε ξανά.

Δηλαδή για ένα υποσύνολο των πόλεων $S \subseteq \{1,2,\dots\dots,n\}$ που συμπεριλαμβάνει την πόλη 1 και την πόλη $j \in$ S , και $C(S,j)$ το μήκος της συντομότερης διαδρομής κάθε πόλης του S, επισκεπτόμενοι κάθε πόλη ακριβώς μια φορά ξεκινώντας από την πόλη 1 και καταλήγοντας στην πόλη j.

Όταν το $|S| > 1$ θεωρούμε ότι $C(S,1) = \infty$ αφού η διαδρομή δεν μπορεί να ξεκινά και να τελειώνει στην πόλη 1. Ας εκφράσουμε τώρα το $C(S,j)$ με την μορφή υπο-προβλημάτων. Πρέπει να ξεκινήσουμε από την πόλη 1 και να καταλήξουμε στην πόλη j. Ως προτελευταία πόλη θα πρέπει να διαλέξουμε μια πόλη πχ $i \in S$, οπότε το συνολικό μήκος της διαδρομής είναι η απόσταση της πόλης 1 έως την πόλη i ήτοι $C(S-\{j\},i)$, συν την απόσταση της τελευταίας πόλης $d_{i,j}$. Πρέπει να διαλέξουμε τη βέλτιστη πόλη i (Dasgupta S., 2006).:

$$C(S,j) = min_{\mathrm{i}\in \mathrm{S}:\mathrm{i}\neq \mathrm{j}} C(S-\{j\},i) + d_{ij}$$

όπου $C(i,\emptyset) = d[i,1]$ το κόστος από τις υπόλοιπες πόλεις προς την πόλη εκκίνησης.

Τα υπο-προβλήματα κατατάσσονται σύμφωνα με το $|S|$. Παραθέτουμε τον ψευδοκώδικα (Dasgupta S., 2006):

$C(\{1\},1) = 0$

$for\ s = 2\ to\ n$:

$for\ all\ subsets\ S \subseteq \{1,2,\ldots..,n\}\ of\ size\ s\ and\ containing\ 1$:

$C(S,1) = \infty$

$for\ all\ j \in S, j \neq 1$:

$$C(S,j) = min\{C(S-\{j\},i)\ +\ d_{i,j}: i\ \in\ S, i\ \neq j\ \}$$

$return\ min_j\ C\ (\{1,\ldots..,n\},j)\ +\ d_{j,1}$

Υπάρχουν το πολύ $2^n * n$ υπο-προβλήματα και κάθε ένα χρειάζεται γραμμικό χρόνο για να το λύσουμε . Έτσι λοιπόν ο συνολικός χρόνος εκτέλεσης είναι $O(n^2 2^n)$.

Πολυπλοκότητα Χρόνου: $(\mathrm{n}-1)\sum_{k=1}^{n-3}\binom{n-2}{k} + 2(n-1) \sim O(n^2 2^n) << O\ (n!)$

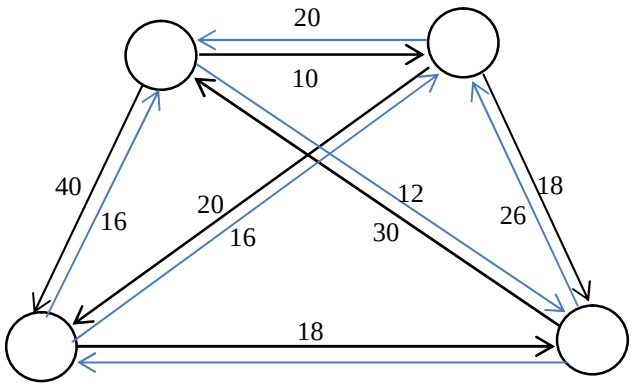
Πολυπλοκότητα Χώρου: $\sum_{k=1}^{n-1} k\binom{n-1}{k} = (\mathrm{n}-1)\ 2^{n-2} \sim O(n2^n)$

3.3 Επίλυση προβλήματος Περιοδεύοντος Πωλητή

Δίνεται ο παρακάτω γράφος τεσσάρων πόλεων με τις αποστάσεις που τις χωρίζουν και μας ζητείται να βρεθεί η βέλτιστη διαδρομή που πρέπει να ακολουθήσει ο περιοδεύων πωλητής ώστε επιστρέφοντας στην αρχική πόλη, να έχει πρώτα επισκεφτεί όλες τις υπόλοιπες ακριβώς μια φορά. Στον ακόλουθο πίνακα αναπαριστούμε τις αποστάσεις των πόλεων μεταξύ τους ως στοιχεία $d_{i,j}$ όπου i η γραμμή και j η στήλη του εκάστοτε στοιχείου. Για παράδειγμα η απόσταση των

πόλεων $1 \rightarrow 2$ στον κατευθυνόμενο γράφο είναι ίση με 20 επομένως $d_{12} = 10$ ενώ η απόσταση από την πόλη 2 στην πόλη 1 αντίστοιχα είναι 10, άρα $d_{21} = 20$. Κατά αυτή τη λογική συμπληρώνουμε ολόκληρο τον πίνακα.

$$\begin{bmatrix} 0 & 20 & 30 & 40 \\ 10 & 0 & 18 & 20 \\ 12 & 26 & 0 & 24 \\ 16 & 16 & 18 & 0 \end{bmatrix}$$



Από ότι διακρίνουμε και από το παραπάνω σχήμα η λύση προκύπτει από τον υπολογισμό του $C(1,\{2,3,4\})$, μέσω της γενικής σχέσης

$$C(1, V - \{1\}) = min_{1 \le k \le n-1}\{d[1,k] + C(k, V - \{1,k\})\}$$

δηλαδή από τον υπολογισμό της βέλτιστης διαδρομής που ξεκινά από την θέση 1, επισκέπτεται τις θέσεις $2,3,4$ με την βέλτιστη σειρά και επιστρέφει στην θέση 1 έχοντας επισκεφτεί την κάθε πόλη ακριβώς μια φορά.

Στο αρχικό στάδιο καταγράφουμε όλες τις αποστάσεις των πόλεων $2,3,4$ προς την πόλη 1.

$$|V| = 0$$

$$C(2, \emptyset) = d[2,1] = 10$$

$$C(3, \emptyset) = d[3,1] = 12$$

$$C(4, \emptyset) = d[4,1] = 16$$

Στο επόμενο στάδιο, υπολογίζουμε τις πιθανές διαδρομές ανάλογα με την πόλη της επιλογής μας. Όπως γίνεται ευκόλως κατανοητό ότι όταν ο πωλητής βρίσκεται πχ στην πόλη 2 οι πιθανές επιλογές του είναι οι πόλεις 3 και 4 ενώ όταν βρίσκεται στην πόλη 3 οι πιθανές επιλογές του είναι οι πόλεις 2 και 4. Αυτά σε συνδυασμό με το

γεγονός ότι ο πωλητής πρέπει να επιστρέψει στην πόλη 1 μας δίνουν το στάδιο $|V| = 1$.

$$|V| = 1$$

$$C(2,\{3\}) = d[2,3] + C(3,\emptyset) = 18 + 12 = 30$$

$$C(2,\{4\}) = d[2,4] + C(4,\emptyset) = 20 + 16 = 36$$

$$C(3,\{2\}) = d[3,2] + C(2,\emptyset) = 26 + 10 = 36$$

$$C(3,\{4\}) = d[3,4] + C(4,\emptyset) = 24 + 16 = 40$$

$$C(4,\{2\}) = d[4,2] + C(2,\emptyset) = 16 + 10 = 26$$

$$C(4,\{3\}) = d[4,3] + C(3,\emptyset) = 18 + 12 = 30$$

Εν συνεχεία, στο στάδιο $|V| = 2$ υπολογίζεται το ελάχιστο κόστος από κάθε πόλη προς τις δύο άλλες που μπορεί να επιλέξει κάνοντας σύγκριση των δύο διαφορετικών διαδρομών λαμβάνοντας υπόψη μας το κόστος επιστροφής στον κόμβο 1.

$$|V| = 2$$

$$C(2,\{3,\mathbf{4}\}) = min(d[2,3] + C(3,\{4\}), d[2,4] + C(4,\{3\})) = min(18 + 40, 20 + 30) = 50$$

$$C(3,\{2,\mathbf{4}\}) = min(d[3,2] + C(2,\{4\}), d[3,4] + C(4,\{2\})) = min(26 + 36, 24 + 26) = 50$$

$$C(4,\{\mathbf{2},3\}) = min(d[4,2] + C(2,\{3\}), d[4,3] + C(3,\{2\})) = min(16 + 30, 18 + 36) = 46$$

Το στάδιο $|V| = 3$ είναι το σημαντικότερο καθώς χρησιμοποιούμε τις σχέσεις από τα προηγούμενα στάδια προκειμένου να βρούμε την επόμενη πόλη που θα επισκεφθεί ο πωλητής. Με βάση το παράδειγμα μας βλέπουμε ότι η βέλτιστη επιλογή είναι η πόλη 2.

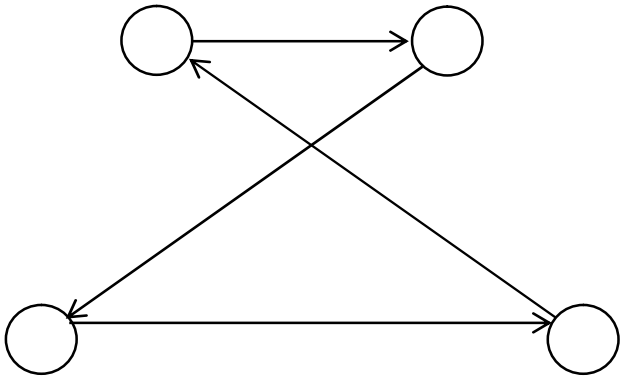
$$|V| = 3$$

$$C(1,\{\mathbf{2},3,4\}) = min\big(d[1,2] + C(2,\{3,4\}), d[1,3] + C(3,\{2,4\}), d[1,4] + C(4,\{2,3\})\big)$$
$$= min(20 + 50, 30 + 50, 40 + 46) = 70$$

*Με έντονο κόκκινο χρώμα υποδηλώνεται ο κόμβος που διαλέξαμε.

Από την παραπάνω επίλυση προκύπτει ότι η βέλτιστη διαδρομή που ξεκινά από την πόλη 1, επισκέπτεται όλες τις πόλεις ακριβώς μια φορά και επιστρέφει στην πόλη ένα είναι η εξής: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$.

Σχηματικά η επίλυση:



ΚΕΦΑΛΑΙΟ 4: ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ ΣΑΚΙΔΙΟΥ

4.1 Ορισμός προβλήματος σακιδίου

Το πρόβλημα του σακιδίου (Knapsack problem) είναι ένα πρόβλημα συνδυαστικής βελτιστοποίησης. Δεδομένου ενός συνόλου αντικειμένων, το κάθε ένα με την δική του μάζα και την δική του αξία, πρέπει να προσδιοριστεί ο συνδυασμός των αντικειμένων που θα συμπεριλάβουμε στη συλλογή ώστε το συνολικό βάρος να είναι μικρότερο ή ίσο με το ζητούμενο όριο και με σκοπό τη μεγιστοποίηση της αξίας της συλλογής. Η ονομασία του προβλήματος προκύπτει από το πρόβλημα που αντιμετωπίζει κάποιος όταν περιορίζεται από ένα δεδομένων διαστάσεων σακίδιο και πρέπει να το γεμίσει με τα αντικείμενα με την μέγιστη δυνατή αξία. Το πρόβλημα εντοπίζεται συχνά σε κατανομή πόρων όταν υπάρχουν οικονομικοί περιορισμοί και διδάσκεται σε τομείς όπως η πληροφορική, η θεωρία της πολυπλοκότητας, η κρυπτογράφηση και τα εφαρμοσμένα μαθηματικά (ανακτήθηκε στις 25 Νοεμβρίου 2013 από το http://en.wikipedia.org/wiki/Knapsack_problem).

4.2 Μοντελοποίηση και μαθηματική επίλυση προβλήματος σακιδίου

Μας δίνεται ένα σύνολο n αντικειμένων όπου κάθε αντικείμενο έχει αξία p και βάρος w, με $p, w \in \mathbb{Z}$, και μας ζητείται να βρούμε ένα υποσύνολο τέτοιο ώστε το άθροισμα του βάρους των επιλεγμένων αντικειμένων να μην ξεπερνά τη μέγιστη χωρητικότητα του σακιδίου M και το συνολικό κόστος που εκφράζεται με τη συνάρτηση $C[n, M]$ να μεγιστοποιείται. Στην παρούσα εργασία θα μελετήσουμε την περίπτωση του Knapsack 0/1 κατά την οποία τα αντικείμενα δεν μπορούν να διαιρεθούν σε μικρότερα αντικείμενα κατά βάρος ή αξία (Razvan C. Bunescu http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture16.pdf, ανακτήθηκε 27 Μαΐου 2014).

Βέλτιστη Υπό-Δομή του προβλήματος 0/1 του σακιδίου.

Έστω ότι το $KNAP(1, n, M)$ υποδηλώνει το πρόβλημα 0/1 του σακιδίου με δυνατές επιλογές αντικειμένων $[1..n]$ και μέγιστης χωρητικότητας M.

Έστω ότι τα $(x_1, x_2, \dots, x_n)$ είναι η βέλτιστη λύση στο πρόβλημα $KNAP(1, n, M)$ τότε έχουμε τα εξής:

1) Αν $x_n = 0$ (δηλαδή αν δεν επιλέξουμε το ν-οστό αντικείμενο), τότε το $(x_1, x_2, \dots, x_{n-1})$ πρέπει να είναι μια βέλτιστη λύση για το πρόβλημα $KNAP(1, n-1, M)$.
2) Αν $x_n = 1$ (δηλαδή αν επιλέξουμε το ν-οστό αντικείμενο), τότε το $(x_1, x_2, \dots, x_{n-1})$ πρέπει να είναι μια βέλτιστη λύση για το πρόβλημα $KNAP(1, n-1, M-w_n)$.

Επίλυση σε επίπεδο υπο-προβλημάτων

Βασισμένοι στην βέλτιστη υποδομή μπορούμε να εκφράσουμε την λύση στο πρόβλημα $0/1$ του σακιδίου κάπως έτσι (Razvan C. Bunescu http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture16.pdf, ανακτήθηκε 27 Μαΐου 2014):

· Έστω ότι $C[n, M]$ είναι η αξία της βέλτιστης λύσης για το $KNAP(1, n, M)$.

$$C[n, M] = \max(\textit{κέρδος για την περίπτωση } 1, \textit{κέρδος για την περίπτωση } 2)$$

$$= \max\left(C[n-1, M], C[n-1, M-w_n] + p_n\right).$$

Ομοίως

$C[n-1, M] = \max(C[n-2, M], C[n-2, M-w_{n-1}] + p_{n-1})$.

$C[n-1, M-w_n] = \max(C[n-2, M-w_n], C[n-2, M-w_n-w_{n-1}] + p_{n-1})$.

Χρήση πίνακα για την αποθήκευση του $C[.,.]$ και κατασκευή του με από-κάτω-προς-τα-πάνω-προσέγγιση

Για παράδειγμα, αν $n = 4$, $M = 9$; $w_4 = 4$, $p_4 = 2$, τότε $C[4,9] = \max\left(C[3,9], C[3,9-4] + 2\right)$.

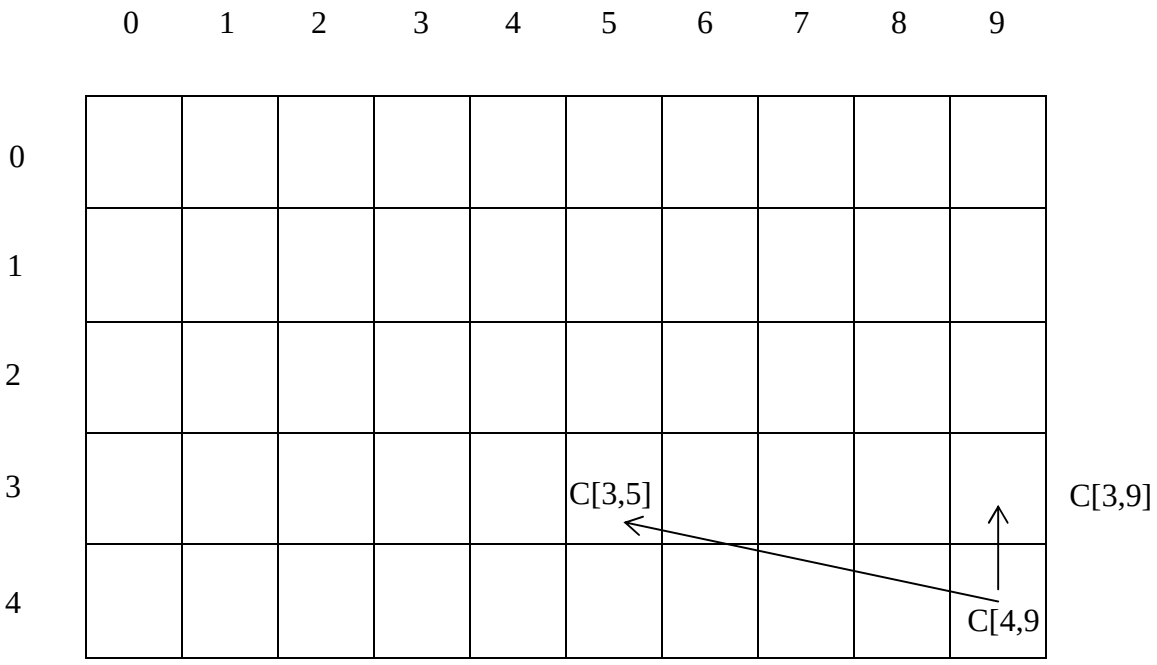
Μπορούμε να χρησιμοποίησουμε έναν δισδιάστατο πίνακα που θα περιέχει τα $C[.,.]$; αν θέλουμε να υπολογίσουμε το $C[4,9]$, τότε τα $C[3,9]$ και $C[3,9-4]$ πρέπει να είναι έτοιμα.

Παρατηρώντας την τιμή $C[n,M] = \max(C[n-1,M], C[n-1,M-w_n] + p_n)$, για να υπολογίσουμε την τιμή $C[n,M]$, χρειαζόμαστε μόνο τις τιμές στην σειρά $C[n-1,.]$.

Έτσι ο πίνακας $C[.,.]$ μπορεί να κατασκευαστεί με από-κάτω-προς-τα-πάνω-προσέγγιση:

1) Υπολογισμός της πρώτης σειράς $C[0,0], C[0,1], C[0,2]$…κ.ο.κ
2) Σειρά με την σειρά γεμίζουμε τον πίνακα.

Ο πίνακας



Κατασκευή του πίνακα: Η αναδρομική επίλυση

Έστω ότι το $C[i,w]$ είναι ένα κελί στον πίνακα $C[.,.]$ που αναπαριστά την αξία της βέλτιστης λύσης για το πρόβλημα $KNAP(1,i,w)$ που αφορά το υπο-πρόβλημα της επιλογής αντικειμένων από $[1 \ldots i]$ με μέγιστη χωρητικότητα w.

Τότε $C[i,w] = \max(C[i-1,w], C[i-1,w-w_i] + p_i)$.

Περιορισμοί

Πρέπει να λάβουμε υπόψη τους παρακάτω περιορισμούς (Razvan C. Bunescu http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture16.pdf, ανακτήθηκε 27 Μαΐου 2014):

- Όταν $i = 0$, δεν υπάρχει αντικείμενο για να επιλέξουμε, οπότε $C[i,w] = 0$
- Όταν $w = 0$, δεν υπάρχει διαθέσιμη χωριτικότητα, οπότε $C[i,w] = 0$
- Όταν $w_i > w$, τότε το επιλεγμένο αντικείμενο υπερβαίνει το όριο της χωριτικότητας και για αυτό δεν μπορούμε να το επιλέξουμε. Οπότε $C[i,w] = C[i-1,w]$ για αυτήν την περίπτωση.

Ολοκληρωμένη η αναδρομική μορφή

Ολοκληρωμένη η αναδρομική επίλυση έχει ως εξής (Razvan C. Bunescu http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture16.pdf, ανακτήθηκε 27 Μαΐου 2014):

$$C[i,w] = \begin{cases} 0 & \text{αν } i = 0 \text{ ή } w = 0 \\ C[i-1,w] & \text{αν } w_i > w \\ \max(C[i-1,w], C[i-1,w-w_i] + p_i & \text{αν } i > 0 \text{ και } w \geq w_i \end{cases}$$

Η λύση για το πρόβλημα $KNAP(1,n,M)$ βρίσκεται στο $C[n,M]$.

Αλγόριθμος

(Razvan C. Bunescu http://oucsace.cs.ohiou.edu/~razvan/courses/cs4040/lecture16.pdf, ανακτήθηκε 27 Μαΐου 2014)

$DP-01KNAPSACK(p[\,], w[\,], n, M)$ // n: πλήθος αντικειμένων, M: χωρητικότητα

$for\ w \coloneqq 0\ to\ M$ $C[0,w] \coloneqq 0;$

$for\ i \coloneqq 0\ to\ n$ $C[i,0] \coloneqq 0;$

$for\ i \coloneqq 1\ to\ n$

$for\ w \coloneqq 1\ to\ M$

$if\ (w[i] > w)$ //δεν μπορούμε να επιλέξουμε το αντικείμενο i

$C[i,w] \coloneqq C[i-1,w];$

$else$

$if(p[i] + C\left[i-1, w-w[i]\right] > C[i-1,w])$

$C[i,w] \coloneqq p[i] + C\left[i-1, w-w[i]\right];$

$else$

$C[i,w] \coloneqq C[i-1,w];$

$return\ C[n,M];$

4.3 Επίλυση προβλήματος σακιδίου

Έχουμε 4 αντικείμενα με βάρος w_i = [1,5,3,4] και με αξία v_i = [15,10,9,5] και για μέγιστο αποθηκευτικό χώρο έχουμε το W = 8

Αντικείμενο	$\boldsymbol{w_i}$	$\boldsymbol{v_i}$
1	1	15
2	5	10
3	3	9
4	4	5

Για $i = 1$ ($w_1 = 1$ και $v_1 = 15$)

w_i	$V[i]$
0	0
1	15
2	15
3	15
4	15
5	15
6	15
7	15
8	15

Ο παραπάνω πίνακας για $i = 1$ συμπληρώνεται μόνο με το αντικείμενο 1 διότι είναι το μόνο διαθέσιμο και μπορεί να τοποθετηθεί για κάθε $w_i > 0$ καθώς το βάρος του είναι ίσο με $w_1 = 1$.

Για $i = 2$ ($w_2 = 5$ και $v_2 = 10$)

w_i	$V[i]$
0	0
1	15
2	15
3	15
4	15
5	15

6	25
7	25
8	25

Στο βήμα αυτό μας δίνεται η δυνατότητα να συνδυάσουμε το αντικείμενο 1 μαζί με το αντικείμενο 2. Ο πίνακας παραμένει ο ίδιος με το προηγούμενο στάδιο μέχρι και το βάρος $w_i = 5$ διότι το άθροισμα των βαρών των αντικειμένων 1,2 είναι ίσο με $w_1 + w_2 = 6$. Έτσι, από $w_i = 6$ έως και $w_i = 8$ οι τιμές $V[i, j]$ του παραπάνω πίνακα είναι ίσες με $v_1 + v_2 = 25$.

Για $i = 3$ ($w_3 = 3$ και $v_3 = 9$)

w_i	$V[i]$
0	0
1	15
2	15
3	15
4	24
5	24
6	25
7	25
8	25

Για $w_i = 1$ και $w_i = 2$ επιλέγουμε το αντικείμενο 1 αφού δεν υπάρχει άλλο αντικείμενο με βάρος 1 ή 2.

Για w_i=3 και πάλι επιλέγουμε το αντικείμενο 1 καθώς $v_1 > v_3$.

Για $w_i = 4$ και $w_i = 5$ ο συνδυασμός των αντικειμένων 1 και 3 είναι αυτός που αποφέρει την μεγαλύτερη συνολική αξία $v_1 + v_3 = 24$

Στα $w_i = 6$, $w_i = 7$, και $w_i = 8$ ο βέλτιστος συνδυασμός επιτυγχάνεται με τα αντικείμενα 1 και 5 δηλαδή $v_1 + v_2 = 15 + 10 = 25$

Για $i = 4$ ($w_4 = 4$ και $v_4 = 5$)

w_i	$V[i]$
0	0
1	15
2	15
3	15
4	24
5	24
6	25
7	25
8	29

Σε αυτό το στάδιο έχουμε στην διάθεση μας και τα 4 αντικείμενα και παρατηρούμε ότι η διαφορά με το προηγούμενο στάδιο εντοπίζεται στο $w_i = 8$ δηλαδή στο διαθέσιμο βάρος που μπορούμε να συνδυάσουμε τα αντικείμενα 1,3,4 με συνολικό βάρος 8 και συνολική αξία 29.

Συνδυάζοντας όλα τα παραπάνω προκύπτει ο ακόλουθος πίνακας:

	0	1	2	3	4	5	6	7	8
{0}	0	0	0	0	0	0	0	0	0
{1}	0	15	15	15	15	15	15	15	15
{1,2}	0	15	15	15	15	15	25	25	25
{1,2,3}	0	15	15	15	24	24	25	25	25
{1,2,3,4}	0	15	15	15	24	24	25	25	29

Όπως είναι προφανές η βέλτιστη λύση βρίσκεται στην θέση του πίνακα $V[4,8]$ με βέλτιστο συνδυασμό αντικειμένων τα 1,3,4 και με συνολική αξία 29.

ΚΕΦΑΛΑΙΟ 5: ΤΟ ΠΡΟΒΛΗΜΑ ΤΩΝ ΑΡΙΘΜΩΝ FIBONACCI

5.1 Ορισμός προβλήματος αριθμών Fibonacci

Ο Leonardo Fibonacci γεννήθηκε γύρω στο 1170 και απεβίωσε γύρω στο 1250 στην Πίζα (σημερινή Ιταλία). Ταξίδεψε επανειλημμένα στην Ευρώπη και στην Νότια Αφρική. Συνέγραψε διάφορα κείμενα γύρω από τα μαθηματικά τα οποία μεταξύ άλλων εισήγαγαν στην Ευρώπη την Ινδο-αραβική σημειογραφία των αριθμών. Παρόλο που τα βιβλία του έπρεπε να αντιγραφούν με το χέρι, η κυκλοφορία τους ήταν ιδιαιτέρως ευρεία. Στο πιο γνωστό του βιβλίο με τίτλο Liber Abaci που εκδόθηκε το 1202, αναφέρεται στο παρακάτω πρόβλημα:

Βάζουμε ένα ζευγάρι από κουνέλια σε ένα μέρος που περιβάλλεται από όλες τις πλευρές από τοίχο. Πόσα ζευγάρια κουνέλια μπορούν να παραχθούν από το αρχικό ζευγάρι σε ένα έτος αν υποθέσουμε ότι κάθε μήνα το κάθε ζευγάρι γεννά ένα νέο ζευγάρι το οποίο γίνεται παραγωγικό από τον δεύτερο μήνα;

Σήμερα η επίλυση σε αυτό το πρόβλημα είναι γνωστή ως “Ακολουθία Fibonacci” ή “Αριθμοί Fibonacci” (Moler C. 2004). Πάνω στους Αριθμούς Fibonacci έχει βασιστεί μια μικρή μαθηματική βιομηχανία. Μια έρευνα στο διαδίκτυο και σε σχετική βιβλιογραφία θα αποφέρει στον ενδιαφερόμενο εκατοντάδες αποτελέσματα. Παρ’ όλα αυτά, αν ο Leonardo Fibonacci δεν είχε προσδιορίσει τον ένα μήνα ως το χρονικό διάστημα που το νεογέννητο ζεύγος κουνελιών γίνεται παραγωγικό, τότε πολύ απλά δεν θα υπήρχε μια ακολουθία με το όνομα του. Ο αριθμός των ζευγαριών απλά θα διπλασιαζόταν κάθε μήνα. Μετά από n μήνες θα υπήρχαν 2^n ζευγάρια κουνελιών.

Έστω ότι η f_n υποδηλώνει τον αριθμό των κουνελιών μετά από n μήνες. Το σημείο κλειδί εδώ είναι το γεγονός ότι ο αριθμός των κουνελιών στο τέλος ενός μήνα ισούται με τον αριθμό στην αρχή του μήνα συν τον αριθμό των γεννήσεων που προέκυψαν από τα παραγωγικά ζευγάρια:

$f_n = f_{n-1} + f_{n-2}$

Οι αρχικές συνθήκες είναι τέτοιες ώστε τον πρώτο μήνα υπάρχει ένα ζευγάρι κουνελιών και τον δεύτερο μήνα υπάρχουν δυο ζευγάρια:

$f_{(1)} = 1, f_{(2)} = 2$

Με αυτές τις αρχικές συνθήκες, η απάντηση στο αρχικό ερώτημα του Fibonacci σχετικά με τον πληθυσμό των κουνελιών με την πάροδο ενός έτους δίνεται από την $f_{(12)}$.

Η $f_{(12)}$παράγει τα παρακάτω:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233

Η απάντηση είναι 233 ζευγάρια κουνελιών. (θα ήταν 4096 ζευγάρια αν ο αριθμός διπλασιαζόταν κάθε μήνα για 12 μήνες).

5.2 Μοντελοποίηση προβλήματος αριθμών Fibonacci

Ας υποθέσουμε ότι μας ζητείται να βρούμε το ν-οστό στοιχείο της ακολουθίας Fibonacci. Παρακάτω ακολουθεί ο διάσημος αναδρομικός αλγόριθμος για την επίλυση του προβλήματος χωρίς χρήση δυναμικού προγραμματισμού (Cheng R. , http://www.ugrad.cs.ubc.ca/~cs490/sec202/notes/dp/DP%201.pdf, ανακτήθηκε 25 Απριλίου 2014):

$int\ fibonacci_{(int\ n)}$ {

$if\ (n == 1)\ return\ 1;$

$if\ (n == 2)\ return\ 1;$

$return\ fibonacci_{(n-1)} + fibonacci_{(n-2)};$

}

Παρ' όλο που ο παραπάνω αλγόριθμος είναι σωστός, είναι επίσης χαρακτηριστικά αργός. Για παράδειγμα αν θελήσουμε να βρούμε τον $fibonacci_{(6)}$ λόγω του αναδρομικού αλγορίθμου θα πρέπει να υπολογίσουμε τον $fibonacci_{(3)}$ τρείς φορές. Επιπροσθέτως, κάθε φορά που καλούμε τον $fibonacci_{(3)}$ θα υπολογίζονται οι $fibonacci_{(2)}$ και $fibonacci_{(1)}$.

Όπως γίνεται κατανοητό όλοι αυτοί οι επαναλαμβανόμενοι υπολογισμοί είναι περιττοί. Μπορούμε να τους εκμηδενίσουμε όμως με την χρήση πίνακα. Με αυτόν τον τρόπο ο υπολογισμός κάθε στοιχείου της ακολουθίας γίνεται ακριβώς μια φορά και αποθηκεύεται σε πίνακα. Έτσι λοιπόν κάθε φορά που καλούμε τον $fibonacci_{(3)}$ δεν χρειάζεται να υπολογίζουμε και τους $fibonacci_{(2)}$ και $fibonacci_{(1)}$, απλά τους αναζητούμε στον πίνακα που είναι ήδη αποθηκευμένοι.

Ακολουθεί ο αλγόριθμος για την επίλυση του προβλήματος των αριθμών Fibonacci με χρήση πίνακα (Cheng R. , http://www.ugrad.cs.ubc.ca/~cs490/sec202/notes/dp/DP%201.pdf, ανακτήθηκε 25 Απριλίου 2014):

// $στο\ fib_{[i]}$ θα αποθηκεύεται το αποτέλεσμα του $fibonacci_{(i)}$. Αρχικές τιμές 0.

$int[\,]\ fib\ =\ new\ fib_{[32]};$

$int\ fibonacci_{(int\ n)}$ {

$if\ (n\ =\ 1)\ return\ 1;$

$if\ (n\ =\ 2)\ return\ 1;$

$if\ (fib_{[n]}!\ =\ 0)\ return\ fib_{[n]};$ // ήδη υπολογισμένο $fibonacci_{(n)}$

$return\ fib_{[n]}\ =\ fibonacci_{(n-1)}\ +\ fibonacci_{(n-2)};$

Ο παραπάνω αλγόριθμος είναι στην ουσία δυναμικός προγραμματισμός για τους εξής λόγους:

1. Χρησιμοποιήσαμε την αναδρομική σχέση $fibonacci_{(n)} = fibonacci_{(n-1)} + fibonacci_{(n-2)}$. Για την ακρίβεια διαχωρίσαμε το πρόβλημα σε μικρότερα υπο-προβλήματα για την εύρεση των $fibonacci_{(n-1)}$ και $fibonacci_{(n-2)}$
2. Χαρακτηρίσαμε την λύση του προβλήματος της εύρεσης του ν-στού αριθμού $fibonacci$ με έναν και μόνο ακέραιο n. Αυτή είναι και η κατάσταση του προβλήματος.
3. Αφού αντιληφθήκαμε ότι οι επαναλαμβανόμενοι υπολογισμοί των ίδιων υποπροβλημάτων δεν είναι βέλτιστοι, κάναμε χρήση πίνακα.

5.3 Επίλυση προβλήματος αριθμών Fibonacci

Εφαρμόζοντας Δυναμικό Προγραμματισμό για το πρόβλημα Fibonacci συνειδητοποιούμε ότι υπάρχουν επικαλυπτόμενα υπο-προβλήματα και για αυτό τον λόγο πρέπει η σχεδίαση του αλγορίθμου επίλυσης να γίνει με τέτοιο τρόπο ώστε αυτά τα υπο-προβλήματα παρόλο που θα τα χρειαστούμε αρκετές φορές, να τα υπολογίσουμε ακριβώς μια φορά. Για να το καταφέρουμε αυτό μπορούμε να δημιουργήσουμε ένα πίνακα a μήκους n όπου το $a_{[i]}$ περιλαμβάνει το $fibonacci_{i+1}$. Στην συνεχεία γεμίζουμε τον πίνακα με την από-κάτω-προς-τα-πάνω προσέγγιση (http://martijn.van.steenbergen.nl/projects/DynProg.pdf, ανακτήθηκε 23 Μαΐου 2014):

$int\ fib_{(int\ n)}$ {

$if\ (n = 1)\ return\ 1;$

$if\ (n = 2)\ return\ 1;$

$int[\ ]a = new\ int[n];$

$a_{[0]} = 1;$

$a_{[1]} = 1;$

$for\ (int\ i = 2; i < n; i++)$

$a_{[i]} = a_{[i-1]} + a_{[i-2]};$

$return\ a_{[n-1]}$;

}

Όταν η επανάληψη for ολοκληρώνεται, τότε εμφανίζεται το τελευταίο στοιχείο του πίνακα που περιέχει το ζητούμενο $fibonacci_{(n)}$. Με αυτήν την τακτική το $fibonacci_{(2)}$ υπολογίζεται μόνο μια φορά σε αντίθεση με τις πέντε φορές στο προηγούμενο παράδειγμα. Αξίζει να τονίσουμε ότι επειδή τα στοιχεία του πίνακα υπολογίζονται μόνο μια φορά, ανακαλούνται πιο σπάνια. Για παράδειγμα το στοιχείο $a_{[1]}$ που περιέχει την τιμή για το $fibonacci_{(2)}$ το ανακαλούμε μόνο δυο φορές αντί για πέντε. Επίσης καλό είναι να γνωρίζουμε προκαταβολικά τις ακριβείς τιμές που θα χρειαστούμε για τον υπολογισμό της τελικής λύσης. Για το πρόβλημα των αριθμών Fibonacci αυτό είναι εύκολο γιατί χρειαζόμαστε όλες τις τιμές από το $fibonacci_{(1)}$ μέχρι και το $fibonacci_{(n)}$. Για άλλα προβλήματα αυτό μπορεί να είναι αρκετά πιο δύσκολο. Αν δεν μπορούμε να προβλέψουμε πόσες τιμές θα χρειαστεί να υπολογίσουμε, η επίλυση με τον δυναμικό προγραμματισμό ίσως οδηγήσει στον υπολογισμό περισσότερων τιμών από όσες χρειαζόμαστε με αποτέλεσμα τον αρνητικό αντίκτυπο σε υπολογιστικό χρόνο.

Με τον παραπάνω αλγόριθμο δυναμικού προγραμματισμού επιτυγχάνουμε πολυπλοκότητα χώρου $O(n)$ (για τον πίνακα) και πολυπλοκότητα χρόνου $O(n)$ (για να γεμίσει ο πίνακας από την επανάληψη for).

ΚΕΦΑΛΑΙΟ 6: ΕΠΙΛΥΣΗ ΠΡΟΒΛΗΜΑΤΟΣ ΕΘΝΙΚΟΥ ΑΥΤΟΚΙΝΗΤΟΔΡΟΜΟΥ

6.1 Ορισμός προβλήματος εθνικού αυτοκινητόδρομου

Μια εταιρεία εκμετάλλευσης ενός εθνικού αυτοκινητόδρομου θέλει να κατασκευάσει σταθμούς εξυπηρέτησης (ΣΕΑ) κατά μήκος του. Η εταιρεία έχει μια λίστα δυνατών χιλιομετρικών θέσεων $v_1, v_2, \dots, v_n$ κατά μήκος του συγκεκριμένου αυτοκινητόδρομου, αριθμημένες σύμφωνα με τη χιλιομετρική τους απόσταση από την αφετηρία του αυτοκινητόδρομου. Η μελέτη σκοπιμότητας καταδεικνύει ότι η εταιρεία αναμένει έσοδα $s_i > 0$ από την τοποθέτηση ενός ΣΕΑ στη θέση v_i. Σύμφωνα με την κείμενη νομοθεσία, δεν επιτρέπεται η κατασκευή δύο ΣΕΑ σε απόσταση μικρότερη των 30 χλμ. Την εταιρεία την ενδιαφέρει η κατασκευή ΣΕΑ σε ένα υποσύνολο των δυνατών θέσεων έτσι ώστε να μεγιστοποιήσει τα έσοδά της, χωρίς φυσικά να παραβιάζεται η νομοθεσία.

Για παράδειγμα, υποθέτουμε ότι ο αυτοκινητόδρομος έχει μήκος 150 χλμ, ότι υπάρχουν 10 δυνατές τοποθεσίες στις θέσεις $v_1, v_2, \dots, v_{10} = 10, 25, 40, 55, 65, 100, 110, 120, 130, 145$ και ότι τα αναμενόμενα έσοδα είναι $s_1, s_2, \dots, s_{10} = 1, 5, 7, 8, 10, 15, 2, 6, 3, 2$. Τότε, εφικτές λύσεις είναι τα υποσύνολα $\{10, 40, 100, 130\}$, $\{25, 55, 110, 145\}$, $\{25, 55, 100, 130\}$ και $\{25, 65, 100, 130\}$ με έσοδα $26, 17, 31$ και 33 αντίστοιχα. Το υποσύνολο $\{25, 65, 100, 130\}$ αποτελεί τη βέλτιστη λύση, αφού αποφέρει τα περισσότερα έσοδα.

6.2 Μοντελοποίηση προβλήματος εθνικού αυτοκινητόδρομου

Θα σχεδιάσουμε έναν αλγόριθμο δυναμικού προγραμματισμού ο οποίος, δεδομένου ενός στιγμιότυπου του παραπάνω προβλήματος, βρίσκει το ύψος των εσόδων της βέλτιστης λύσης. Η περιγραφή του αλγορίθμου θα περιλαμβάνει την αναδρομική σχέση που διέπει τον αλγόριθμο και συμπληρώνει τον πίνακα δυναμικού προγραμματισμού καθώς και τον χρόνο εκτέλεσης του.

Καταρχήν παρατηρούμε ότι αν η βέλτιστη λύση περιέχει τη θέση v_n τότε όλες οι θέσεις v_i , με $i < n$, και οι οποίες βρίσκονται σε απόσταση μικρότερη των 30 χλμ δεν μπορούν να συμπεριληφθούν στη βέλτιστη λύση. Διαφορετικά, αν η βέλτιστη λύση δεν περιέχει τη θέση v_n τότε είναι ίδια με τη βέλτιστη λύση για τις θέσεις $v_1, v_2, \dots, v_{n-1}$.

Ο ίδιος συλλογισμός ισχύει και για τη βέλτιστη λύση των πρώτων $v_1, v_2, \dots, v_i$ θέσεων. Έστω ότι $p_{(i)}$ συμβολίζει τη μέγιστη τιμή $j < i$, για την οποία η θέση v_j απέχει τουλάχιστον 30 χλμ από τη θέση v_i (αν δεν υπάρχει j με αυτή την ιδιότητα, τότε $p_{(i)} = 0$. Έστω επίσης ότι $A_{[i]}$ συμβολίζει τα έσοδα της βέλτιστης λύσης (βέλτιστου υποσυνόλου) των θέσεων $v_1, v_2, \dots, v_i$. Τότε, με βάση τον παραπάνω συλλογισμό, προκύπτει η αναδρομική σχέση

$$A_{[i]} = \max\left\{s_i + A_{[p_{(i)}]}, A_{[i-1]}\right\}$$

Μπορούμε να υπολογίσουμε τις τιμές του πινάκα δυναμικού προγραμματισμού A σε αυξανόμενη σειρά ως προς το $i = 1 \dots n$ με βάση την παραπάνω αναδρομή, θέτοντας ως αρχικές τιμές $A_{[0]} = 0$ και $A_{[1]} = s_1$. Το ύψος των εσόδων της βέλτιστης λύσης είναι η τιμή του $A_{[n]}$. Είναι επίσης προφανές ότι αν γνωρίζουμε για κάθε i τις τιμές $p_{(i)}$, τότε ο υπολογισμός του $A_{[i]}$ γίνεται σε σταθερό χρόνο και άρα ο πίνακας A μπορεί να υπολογισθεί σε χρόνο $O_{(n)}$.

Οι τιμές $p_{(i)}$ μπορούν να υπολογισθούν σε χρόνο $O_{(n)}$ ως εξής: Για κάθε θέση v_i ορίζουμε μια νέα θέση $w_i = v_i - 30$. Ο παρακάτω πίνακας υπολογίζει τις τιμές για τα $w_1 \dots w_{10}$ όπως αυτά προκύπτουν από τα $v_1 \dots v_{10}$.

$v_1 = 10$	$w_1 = v_1 - 30 = 10 - 30 = -20$
$v_2 = 25$	$w_2 = v_2 - 30 = 25 - 30 = -5$
$v_3 = 40$	$w_3 = v_3 - 30 = 40 - 30 = 10$
$v_4 = 55$	$w_4 = v_4 - 30 = 55 - 30 = 25$
$v_5 = 65$	$w_5 = v_5 - 30 = 65 - 30 = 35$
$v_6 = 100$	$w_6 = v_6 - 30 = 100 - 30 = 70$
$v_7 = 110$	$w_7 = v_7 - 30 = 110 - 30 = 80$
$v_8 = 120$	$w_8 = v_8 - 30 = 120 - 30 = 90$
$v_9 = 130$	$w_9 = v_9 - 30 = 130 - 30 = 100$
$v_{10} = 145$	$w_{10} = v_{10} - 30 = 145 - 30 = 115$

Στην συνέχεια δημιουργούμε δύο λίστες, μια για τα $v_1, v_2, \dots, v_n$ και μια για τα $w_1, w_2, \dots, w_n$.

$$v_i = [10, 25, 40, 55, 65, 100, 110, 120, 130, 145]$$

$$w_i = [-20, -5, 10, 25, 35, 70, 80, 90, 100, 115]$$

Συγχωνεύουμε τις λίστες $v_1, v_2, \dots, v_n$ και $w_1, w_2, \dots, w_n$ σε χρόνο $O_{(n)}$, δημιουργώντας μια νέα συγχωνευμένη λίστα. Σαρώνουμε αυτή τη νέα (συγχωνευμένη) λίστα. Όταν εξετάζουμε τη θέση w_i, γνωρίζουμε ότι όλες οι επόμενες μέχρι την v_i θέσεις δεν μπορούν να επιλεγούν, αφού απέχουν λιγότερο από 30 χλμ από την v_i. Επομένως, θέτουμε ως $p_{(i)}$ τη μεγαλύτερη τιμή $j < i$ από όλες τις θέσεις v_j που έχουμε ήδη σαρώσει. Είναι προφανές ότι ο υπολογισμός όλων των τιμών $p_{(i)}$ μπορεί να γίνει σε χρόνο $O_{(n)}$ και άρα ο συνολικός χρόνος του αλγορίθμου μας είναι $O_{(n)}$.

6.3 Επίλυση προβλήματος εθνικού αυτοκινητόδρομου

Παρατηρούμε ότι $p_{(1)} = 0, p_{(2)} = 0, p_{(3)} = 1, p_{(4)} = 2, p_{(5)} = 2, p_{(6)} = 5, p_{(7)} = 5, p_{(8)} = 5, p_{(9)} = 6, p_{(10)} = 7$.

Η εκτέλεση του αλγορίθμου στο παράδειγμα έχει ως εξής:

$$A_{[0]} = 0$$

$$A_{[1]} = 1$$

$$A_{[2]} = max\left\{5 + A_{[p_{(2)}]}, A_{[1]}\right\} = \max\{5 + 0,1\} = 5.$$

$$A_{[3]} = max\left\{7 + A_{[p_{(3)}]}, A_{[2]}\right\} = \max\{7 + 1,5\} = 8.$$

$$A_{[4]} = max\left\{8 + A_{[p_{(4)}]}, A_{[3]}\right\} = \max\{8 + 5,8\} = 13.$$

$$A_{[5]} = max\left\{10 + A_{[p_{(5)}]}, A_{[4]}\right\} = \max\{10 + 5,13\} = 15.$$

$$A_{[6]} = max\left\{15 + A_{[p_{(6)}]}, A_{[5]}\right\} = \max\{15 + 15,15\} = 30.$$

$$A_{[7]} = max\left\{2 + A_{[p_{(7)}]}, A_{[6]}\right\} = \max\{2 + 15,30\} = 30.$$

$$A_{[8]} = max\left\{6 + A_{[p_{(8)}]}, A_{[7]}\right\} = \max\{6 + 15,30\} = 30.$$

$$A_{[9]} = max\left\{3 + A_{[p_{(9)}]}, A_{[8]}\right\} = \max\{3 + 30,30\} = 33.$$

$$A_{[10]} = max\left\{2 + A_{[p_{(10)}]}, A_{[9]}\right\} = \max\{2 + 30,33\} = 33.$$

Έχοντας τον πίνακα δυναμικού προγραμματισμού A, μπορούμε να βρούμε τα στοιχεία του υποσυνόλου θέσεων της βέλτιστης λύσης με οπισθόδρομη ιχνηλάτηση των υπολογισμών από το $A_{[n]}$, ακολουθώντας κάθε φορά το αποτέλεσμα του τελεστή max. Δηλαδή, αρχίζοντας από τη θέση $A_{[n]}$ και αναλύοντας την αναδρομική σχέση, προσδιορίζουμε αν το μέγιστο προέρχεται από τη θέση $A_{[n-1]}$, ή από την $A_{[p_{(n)}]}$ και το έσοδο s_n. Ανάλογα με το αποτέλεσμα, συνεχίζουμε οπισθόδρομα είτε στη θέση $A_{[n-1]}$, είτε στην $A_{[p_{(n)}]}$. Με ανάλογο τρόπο, συνεχίζουμε προς τα πίσω μέχρι τη θέση $A_{[0]}$.

Η οπισθόδρομη ιχνηλάτηση στους υπολογισμούς του παραπάνω παραδείγματος μας δίνει ότι το υποσύνολο θέσεων της βέλτιστης λύσης αποτελείται από τα έσοδα:

Βέλτιστη λύση $A_{[9]}$ (αποτέλεσμα τελεστή max στο $A_{[10]}$),

s_9 (θέση v_9) και τη βέλτιστη λύση $A_{[p_{(9)}]} = A_{[6]}$ (αποτέλεσμα τελεστή max στο $A_{[9]}$),

s_6 (θέση v_6) και τη βέλτιστη λύση $A_{[p_{(6)}]} = A_{[5]}$ (αποτέλεσμα τελεστή max στο $A_{[6]}$),

s_5 (θέση v_5) και τη βέλτιστη λύση $A_{[p_{(5)}]} = A_{[2]}$ (αποτέλεσμα τελεστή max στο $A_{[5]}$),

s_2 (θέση v_2) και τη βέλτιστη λύση $A_{[p_{(2)}]} = A_{[0]}$ (αποτέλεσμα τελεστή max στο $A_{[2]}$).

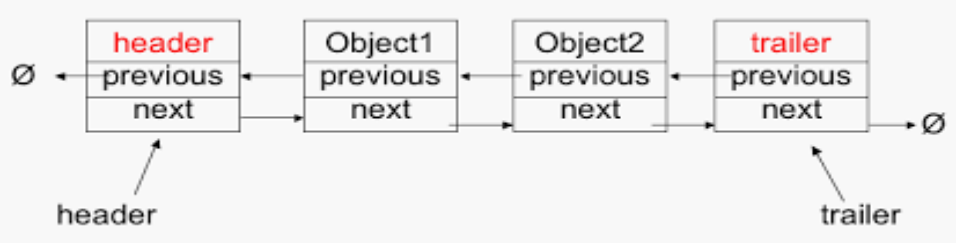
Δηλαδή, το υποσύνολο θέσεων της βέλτιστης λύσης είναι το $\{v_2, v_5, v_6, v_9\} = \{25, 65, 100, 130\}$.

6.4 Διπλά Συνδεδεμένες Λίστες

Η συνδεδεμένη λίστα είναι μια δομή δεδομένων που παρέχει μια απλή αλλά πολλές φορές κατάλληλη μέθοδο επίλυσης προβλημάτων. Η γλώσσα C (στην οποία είναι γραμμένος ο κώδικας του προβλήματος εθνικού αυτοκινητοδρόμου) δεν διαθέτει “έτοιμη” κάποια δομή δεδομένων για συνδεδεμένες λίστες, οπότε απαιτείται από τον χρήστη να προγραμματίσει την δική του.

Μια συνδεδεμένη λίστα αποτελείται από κόμβους που διατηρούν τα δεδομένα της λίστας. Κάθε συνδεδεμένη λίστα πρέπει να έχει τον αρχικό κόμβο που δείχνει προς τον πρώτο κόμβο δεδομένων. Κάθε κόμβος δεδομένων εμπεριέχει τα δεδομένα για αυτό το στοιχείο στην λίστα και έναν δείκτη που δείχνει προς τον επόμενο κόμβο της λίστας. (http://www.codeproject.com/Articles/641175/An-Introduction-to-Linked-Lists-in-C-Cplusplus)

Η διπλά συνδεδεμένη λίστα είναι μία πιο εξεζητημένη μορφή της “απλής” συνδεδεμένης λίστας. Κάθε κόμβος στην λίστα εμπεριέχει δυο δείκτες αντί για έναν. Ο ένας δείκτης δείχνει προς τον προηγούμενο κόμβο και ο άλλος προς τον επόμενο. Ο προηγούμενος δείκτης από τον πρώτο κόμβο και ο επόμενος δείκτης από τον τελευταίο κόμβο δείχνουν το μηδέν. Σε σύγκριση με τις συνδεδεμένες λίστες, οι διπλά συνδεδεμένες απαιτούν χειρισμό περισσότερων δεικτών αλλά ταυτόχρονα ζητούν λιγότερες πληροφορίες γιατί μπορούμε να χρησιμοποιήσουμε τον προηγούμενο δείκτη για να παρατηρήσουμε το προηγούμενο στοιχείο. Επίσης, οι διπλά συνδεδεμένες λίστες έχουν δυναμικό μέγεθος, το οποίο μπορεί να καθοριστεί μόνο κατά την στιγμή της εκτέλεσης. (http://www.thelearningpoint.net/computer-science/data-structures-doubly-linked-list-with-c-program-source-code)



Το πλεονέκτημα της διπλά συνδεδεμένης λίστας έναντι της “απλής” είναι ότι δεν χρειάζεται να παρακολουθούμε τον προηγούμενο κόμβο για την διάσχιση και πιο συγκεκριμένα δεν χρειάζεται καθόλου η διάσχιση ολόκληρης της λίστας για την εύρεση του προηγούμενου κόμβου.

Το μειονέκτημα είναι ότι απαιτείται ο χειρισμός περισσοτέρων δεικτών και η ενημέρωση περισσοτέρων συνδέσμων.

6.5 Σύντομη Περιγραφή Συναρτήσεων Προβλήματος

Ακολουθεί μια σύντομη περιγραφή των συναρτήσεων που χρησιμοποιήθηκαν για την εκτέλεση του προβλήματος εθνικού αυτοκινητοδρόμου:

1. Δήλωση Λίστας

Δήλωση κόμβου

```
typedef struct listNode {
    struct listNode * next;
    struct listNode * prev;
    int data;
} ListNode;
```

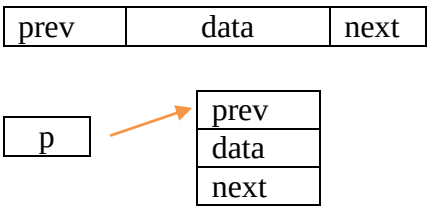
Για τον ορισμό ενός κόμβου μιας λίστας , χρησιμοποιούμε ως τύπο δεδομένων το ListNode, για παράδειγμα ListNode * p;

Ο δείκτης p αντιπροσωπεύει μια θέση μνήμης που αποτελείται από:

1) τον δείκτη προς τον επόμενο κόμβο

2) τον δείκτη προς τον προηγούμενο κόμβο

3) ακέραια δεδομένα



Χρησιμοποιώντας την συνάρτηση malloc έχουμε την δυνατότητα να δεσμεύσουμε χώρο μνήμης για την αποθήκευση ενός ListNode.

$P = (ListNode *)malloc(sizeof(ListNode));$

Δήλωση λίστας

```
typedef struct {
    int count;
    ListNode *first;
    ListNode *last;
} List;
```

Η δομή List αποτελείται από:

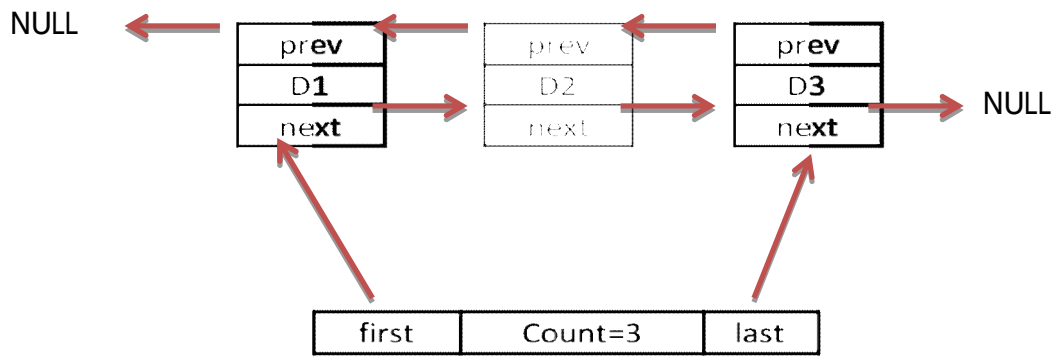
1) τον δείκτη προς τον πρώτο κόμβο της λίστας

2) τον δείκτη προς τον τελευταίο κόμβο της λίστας

3) την ακέραια μεταβλητή count που συμβολίζει το σύνολο των κόμβων

count
first
last

Σχήμα που αναπαριστά μια λίστα με 3 στοιχεία:



<u>Συνάρτηση που υπολογίζει το μέγιστο(max) ανάμεσα σε δυο ακέραιους a και b</u>

```
int max(int a, int b){
    return a > b? a: b;
}
```

2. Αρχικοποίηση Λίστας

```
int initList(List ** list){
        * list = (List *)malloc(sizeof(List));
        if ((* list) == NULL) return EXIT_FAILURE;
        (* list)-> first = NULL;
        (* list)-> last = NULL;
        (* list)-> count = 0;
        return EXIT_SUCCESS;
}
```

Την παραπάνω συνάρτηση την καλούμε στο main για την αρχικοποίηση των λιστών V,S,W,P,A,Solution.

```
List * V,* S,* W,* P,* A,* Solution;        //ορίζουμε ότι οι μεταβλητές
                                            //V,S,W,P,A,Solution είναι δείκτες προς LIST
```

3. Προσθήκη Στοιχείων στη Λίστα

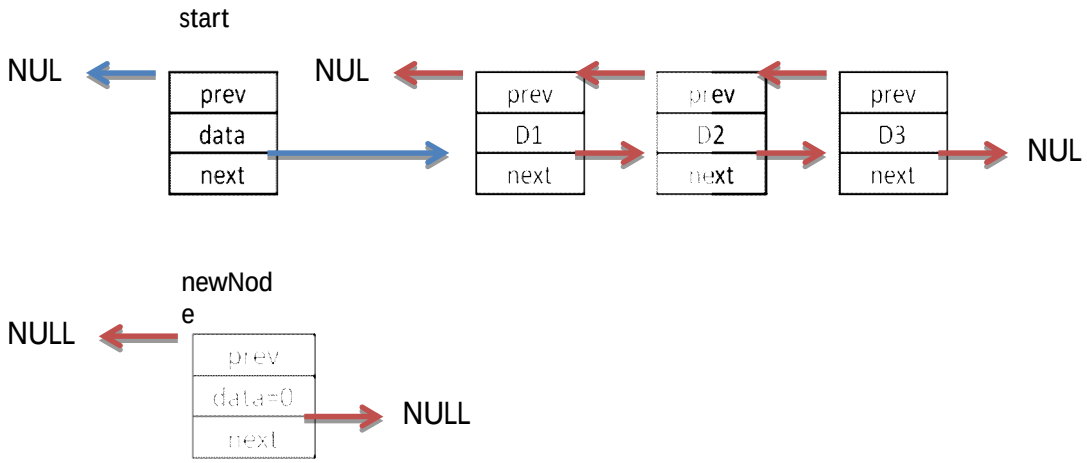
Προσθήκη στην αρχή της λίστας

Η συνάρτηση addAtStart δέχεται δυο ορίσματα. Το ένα είναι ένας δείκτης που δείχνει σε δείκτη προς ListNode (δηλαδή δείκτη που έχει τιμή τη διεύθυνση μνήμης ενός δείκτη προς κόμβο λίστας) και το δεύτερο είναι ένας ακέραιος.

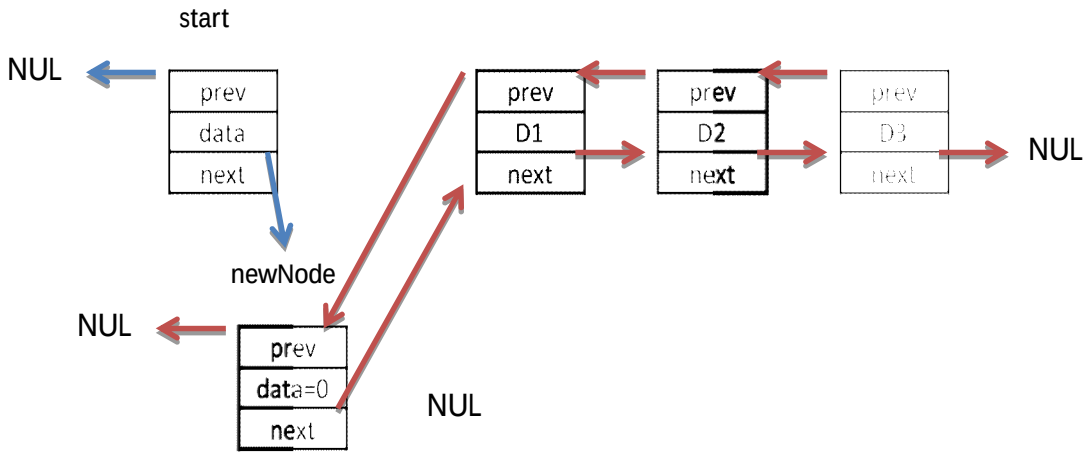
```
void addAtStart(ListNode ** start,int data) {
    ListNode * newNode;                          //ορίζουμε δείκτη προς ListNode με όνομα newNode
    newNode = (ListNode *)malloc(sizeof(ListNode)); //δέσμευση μνήμης για την newNode
    if (newNode == NULL){        //έλεγχος της επιτυχίας ή όχι της δέσμευσης χώρου στην μνήμη
        printf("Error allocating memory" );          //αν ανεπιτυχής εμφανίζει το μήνυμα
        exit(EXIT_FAILURE);                          //διακοπή εκτέλεσης προγράμματος
```

```
        }
    newNode-> data = data;          //η τιμή του newNode γίνεται ίση με την data που περάσαμε
                                                    //σαν όρισμα στη συνάρτηση
    newNode-> next = *start;        //ο δείκτης next του newNode δείχνει στη διεύθυνση start που
                                                    //είναι η αρχή της λίστας
    newNode-> prev = NULL;                  //ο δείκτης prev του newNode δείχνει στο NULL
    *start = newNode;                                   //ο newNode γίνεται η αρχή
}
```

Αρχικά:



Μετά την εκτέλεση της addAtStart:



Συνάρτηση που προσθέτει ένα κόμβο στην αρχή της λίστας. Χρησιμοποιεί την προηγούμενη συνάρτηση addAtStart(ListNode ** start, int data)

```
void addAtStartOfList(List *l, int data){      //έχει για ορίσματα τα δεδομένα και τη λίστα που θα
                                                                    //προσθέσουμε στην αρχή
    addAtStart(&(l-> first), data);     //καλούμε την προηγούμενη συνάρτηση και προσθέτει ένα
                                                                    //κόμβο στην αρχή
    l-> count++;        //αυξάνουμε το πλήθος των στοιχείων της λίστας (μεταβλητή count) κατά 1
}
```

Η συνάρτηση addAtStartOfList δέχεται δυο ορίσματα. Το ένα είναι ένας δείκτης που δείχνει σε List και το άλλο είναι ένας ακέραιος(data). Καλεί την συνάρτηση addAtStart και αυξάνει το πλήθος των στοιχείων της λίστας(count) κατά ένα.

Προσθήκη στο τέλος της λίστας

```
void addAtEndOfList(ListNode *ptr, int data)          //ορίσματα έχει έναν δείκτη προς ListNode
                                                                    //και έναν ακέραιο
{
    /* advances to the last node.*/
    while(ptr-> next! = NULL)                    //εφόσον ο δείκτης next του ptr δεν είναι NULL
    {
        ptr = ptr -> next;                                  //προχωράμε στον επόμενο κόμβο
    }                                                  //στο σημείο αυτό ολοκληρώνονται οι κόμβοι
    ptr-> next = (ListNode *)malloc(sizeof(ListNode));      //δέσμευση μνήμης για επόμενο
                                                                    //κόμβο
    (ptr-> next)-> prev = ptr;                    //θέτουμε σαν prev του επόμενου κόμβου την
```

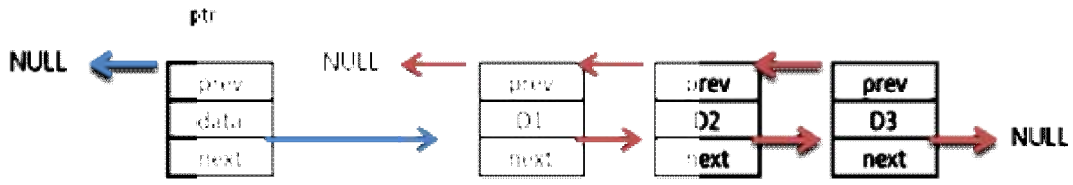
```
                                                        //τιμή του δείκτη ptr
    ptr = ptr−> next;                                   //ο δείκτης ptr δείχνει στον τελευταίο κόμβο
    ptr−> data = data;                                  //ενημερώνουμε την μεταβλητή data του ptr
    ptr−> next = NULL;                                  //ενημερώνουμε το next του ptr
}
```

Η συνάρτηση addAtEndOfList αποτελείται από δυο ορίσματα:

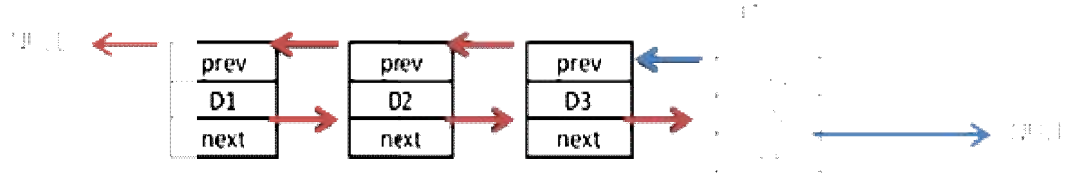
1)έναν δείκτη(pointer) προς ListNode

2)έναν ακέραιο

Αρχικά:



Στο τέλος:



Προσθήκη είτε στη αρχή είτε στο τέλος της λίστας

```
void addToList(List *l,int data){
        if (l->first == NULL){                                  //αν η λίστα είναι κενή
            addAtStartOfList(l,data);              //καλούμε την συνάρτηση addAtStartOfList
         }else{                                                 //διαφορετικά
            addAtEndOfList(l->first,data);         //καλούμε την συνάρτηση addAtEndOfList
            l->count++;                            //ενημερώνουμε το πλήθος(count) της λίστας
         }
         ListNode *ptr;                                         //χρήση ενός pointer προς ListNode
         ptr = (ListNode *)malloc(sizeof(ListNode));            //δέσμευση μνήμης για κόμβο
         ptr = l->first;                                        //αρχικά δείχνει στη αρχή της λίστας
         while(ptr->next != NULL)               //εφόσον υπάρχει επόμενο στοιχείο στην λίστα
             {
                    ptr = ptr->next;            //ο pointer δείχνει στο επόμενο στοιχείο
             }
         l->last = ptr;                         //ενημέρωση του τέλους (δείκτης last) της λίστας
 //free(ptr);                                                   //αποδέσμευση μνήμης για ptr
}
```

Η συνάρτηση addToList καλεί την addAtStartOfList ή την addAtEndOfList ανάλογα αν είναι κενή η λίστα ή αν έχει ήδη στοιχεία. Αν έχει ήδη στοιχεία, τότε προσθέτουμε στο τέλος της.

4. Εκτύπωση

Εκτύπωση όλων των κόμβων

```
void printRecursively(ListNode *ptr)
{           if(ptr == NULL){                                    //αν ο δείκτης ptr είναι κενός
                printf("The list is empty.\n")                  //εμφανίζεται ότι η λίστα είναι κενή
                return ;                                        //η συνάρτηση διακόπτεται
            }
            printf("%d  ",ptr-> data);                          //εμφανίζονται τα δεδομένα
              if(ptr-> next! = NULL)     //αν το επόμενο στοιχείο(δείκτης next) δεν είναι κενό
            {
                printRecursively(ptr-> next);           //αναδρομική κλήση της συνάρτησης
                                                        //θέτοντας ως όρισμα τον επόμενο κόμβο
            }
}
```

Η συνάρτηση καλείται αναδρομικά περνώντας κάθε φορά την τιμή που έχει ο δείκτης next του κόμβου και έχει ως όρισμα έναν δείκτη προς την δομή ListNode.

Εκτύπωση Λίστας

```
void printList(List **l)
{
    printRecursively((*l)-> first);          //καλεί την προηγούμενη συνάρτηση με όρισμα τον
                                             //δείκτη προς το πρώτο στοιχείο(first) της λίστας
    printf("\n");
}
```

Για παράδειγμα η κλήση για την λίστα V : $printList(\&V)$

Η εν λόγω συνάρτηση έχει ως όρισμα έναν δείκτη προς δείκτη προς την δομή List και καλεί την προηγούμενη αναδρομική συνάρτηση περνώντας ως όρισμα την τιμή του δείκτη first της λίστας.

5. Ανάγνωση του ονόματος αρχείου

```
char * readStr(char * file_name){
    printf("\nEnter the file name : ");
    gets(file_name);
    return file_name;
}
```

Η συνάρτηση readStr διαβάζει την μεταβλητή file_name και επιστρέφει πίνακα χαρακτήρων με το κείμενο που καταχώρησε ο χρήστης.

Η κλήση γίνεται ως εξής:

```
char fname[25];          //ορισμός fname me 25 σύμβολα
readStr(fname);          //κλήση της readStr με όρισμα την fname
```

Άνοιγμα αρχείου για ανάγνωση

```
FILE * openfile(FILE * fp,char * file_name ){
    fp = fopen(file_name,"r");
    if (fp == NULL){
        printf("Error opening the file: %s \n", file_name );
```

```
            exit(EXIT_FAILURE);

        }

        return fp;

}
```

Η συνάρτηση openfile έχει ως ορίσματα ένα δείκτη προς file και ένα κείμενο που αντιστοιχεί στο όνομα του αρχείου και επιστρέφει δείκτη προς το αρχείο. Όταν ο δείκτης προς το αρχείο που θα επιστρέψει είναι κενός(NULL), θα τερματίσει την εφαρμογή και θα εμφανίσει μήνυμα αδυναμίας ανοίγματος του αρχείου.

Η κλήση γίνεται ως εξής:

$FILE * fp;$ //ορίζουμε δείκτη fp προς FILE

$fp = openfile(fp, fname);$ //καλούμε την openfile με ορίσματα τον δείκτη fp και την μεταβλητή fname

6. Επιστροφή τιμής στοιχείου της λίστας σε συγκεκριμένη θέση

```
int getNodeData(List * l, int index){
    if (index > l-> count){  //αν η θέση στην λίστα είναι μεγαλύτερη από το πλήθος των στοιχείων
        printf("\nout of list bounds");                    //εμφανίζεται μήνυμα εκτός ορίων
            return 0 ;                                    //εμφανίζεται η τιμή μηδέν
     }
     ListNode * tmp;                                      //ορισμός προσωρινού δείκτη
     tmp = l-> first;                                     //ο δείκτης αρχικά δείχνει στο 1ο στοιχείο
     for (; index > 1; index --){   //όσο η θέση είναι μεγαλύτερη το ένα (η θέση μειώνεται κατά 1)
          tmp = tmp-> next;                               //ο δείκτης συνεχίζει στο επόμενο στοιχείο
      }
      return tmp-> data;    //επιστρέφει την τιμή που υπάρχει στην θέση που δείχνει ο δείκτης tmp
      free(tmp);                                          //αποδέσμευση μνήμης
      tmp = NULL;
}
```

Η συνάρτηση getNodeData έχει για ορίσματα την λίστα και ένα ακέραιο για την θέση στη λίστα και επιστρέφει την τιμή του στοιχείου στην συγκεκριμένη θέση.

Για παράδειγμα η getNodeData(V,2) θα εμφανίσει την τιμή του 2ου στοιχείου της λίστας V.

7. **Η συνάρτηση calcW υπολογισμού της λίστας W**

```
void calcW(List **W, List *V){

    ListNode * tmp;

     tmp = (ListNode *)malloc(sizeof(ListNode));

     tmp = V-> first;

     while (tmp! = NULL){

          int w = 0;

          w = tmp-> data − 30;

          addToList(*W, w);

          tmp = tmp-> next;

      }

     free(tmp);

}
```

Για τον υπολογισμό των τιμών wi = $vi - 30$ και για την δημιουργία της λίστας W, χρησιμοποιούμε την συνάρτηση calcW. Η συνάρτηση calcW δέχεται ως ορίσματα την λίστα V που περιέχει τις τιμές των v_i και έναν δείκτη προς δείκτη προς την λίστα W την οποία και αλλάζει. Για να διασχίσουμε όλα τα στοιχεία της λίστας V χρησιμοποιούμε ένα προσωρινό δείκτη προς ListNode. Για κάθε στοιχείο που δείχνει ο δείκτης tmp παίρνουμε την τιμή της μεταβλητής data του κόμβου της λίστας και αφαιρούμε 30. Στην συνέχεια με την βοήθεια της συνάρτησης addToList δημιουργούμε ένα νέο κόμβο στην λίστα W όπου προσθέτουμε την τιμή που υπολογίσαμε. Στην συνέχεια ο δείκτης tmp δείχνει στον επόμενο κόμβο. Όταν ο δείκτης tmp έχει τιμή NULL, δηλαδή όταν δεν υπάρχει επόμενος κόμβος στην λίστα, η επανάληψη διακόπτεται. Στο τέλος η μνήμη που δεσμεύσαμε για τον δείκτη tmp, αποδεσμεύεται.

Η κλήση έχει ως εξής:

$calcW(\&W, V);$ //με ορίσματα την διεύθυνση της λίστας W και την λίστα V

8. Η συνάρτηση CP υπολογισμού της λίστας P

```
void CP(List *P, List *V, List *W){
    int n = V-> count - 1;
    int i,j;
    j = 0;
    i = 0;
    addToList(P, 0);
    for (i = 1; i <= n; i++){

        j = getNodeData(P, i) + 1;
        while (getNodeData(V, j) <= getNodeData(W, i + 1))
            j++;
        addToList(P, j - 1);
    }
}
```

Η συνάρτηση CP έχει ως ορίσματα:

1)την λίστα V,

2)την λίστα W,

3)την λίστα P (την οποία και υπολογίζει).

Η κλήση της γίνεται ως εξής:

CP(P,V,W); //δέχεται ως ορίσματα την λίστα P, την λίστα V και την λίστα W

Η συνάρτηση CP υπολογίζει τις τιμές Pi, για κάθε i<n σύμφωνα με τον ακόλουθο αλγόριθμο:

```
p[1] = 0;
for(i = 1; i < n; i + +)
{
        j = p[i] + 1;
        while (V[j] <= W[i + 1])
                j++;
        p[i + 1] = j - 1;
}
```

9. Υπολογισμός της βέλτιστης αξίας Λίστας A

Η συνάρτηση CO

Υπολογισμός βέλτιστης αναδρομικότητας

```
int CO(List *A, List *S, List *P, int i){

    int Si, Pi;

    Si = getNodeData(S, i);

    Pi = getNodeData(P, i);

    if (i == 0){

        return 0;

    }else{

        return max(Si + CO(A, S, P, Pi), CO(A, S, P, i − 1));

    }

}
```

Η συνάρτηση CO έχει ως ορίσματα την λίστα S, την λίστα P, την λίστα A (που την υπολογίζει) και το i.

Επίσης υπολογίζει την τιμή Ai, αναδρομικά για κάθε i<n σύμφωνα με τον ακόλουθο αλγόριθμο:

```
CO(i){
        if (i==0){
                return 0;
        }else{
                return max(Si+CO(Pi),CO(i-1));
        }
}
```

Η συνάρτηση C_OPT

Υπολογισμός βέλτιστης αναδρομικότητας χρησιμοποιώντας την προηγούμενη συνάρτηση CO(List *A, List *S, List *P, int i) για όλους τους κόμβους στην λίστα

```
void C_OPT(List *A, List *S, List *P){

    int i, Ai;

    int n = S-> count;

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

        Ai = CO(A, S, P, i);

        addToList(A, Ai);

    }

}
```

Η συνάρτηση findOpt

```
void findOpt(List *A, List *S, List *P, int i){
    int Si, Pi, APi, Ai_1, j;

    if (i > 1){
        j = i − 1;
        Si = getNodeData(S, j);
        Pi = getNodeData(P, j);
        APi = getNodeData(A, Pi + 1);
        Ai_1 = getNodeData(A, i − 1);
        if (Si + APi > Ai_1){

            printf(" %d ", j);
            findOpt(A, S, P, Pi + 1);
        }else{
            findOpt(A, S, P, i − 1);
        }
    }
    else{
        return;
    }
}
```

Την καλούμε ως εξής:

findOpt(A, S, P, i); //με ορίσματα την λίστα A, την λίστα S, την λίστα P και το i(δείκτης στοιχείου στη λίστα A)

Υπολογίζει την τιμή Ai αναδρομικά για κάθε i<n σύμφωνα με τον ακόλουθο αλγόριθμο:

```
FindOpt (j) {
        if (j > 0){
            if (Ai + A(Pi )> Ai-1){
                εκτύπωσε το i
                FindOpt(Pi)
            }Else{
                FindOpt (i-1)
            }
         }
}
```

10. **Απελευθέρωση μνήμης**

Η συνάρτηση freeList δέχεται ως όρισμα ένα δείκτη προς δείκτη προς τον πρώτο κόμβο μιας λίστας. Καλεί αναδρομικά τον εαυτό της με όρισμα τον επόμενο κόμβο(next) του δείκτη first της λίστας τον οποίο και ελευθερώνει από την μνήμη χρησιμοποιώντας τον προσωρινό δείκτη tmp.

```
void freeList(ListNode ** first){

    ListNode * tmp;

    tmp = (ListNode *)malloc(sizeof(ListNode));

    tmp =* first;

    if (tmp == NULL) return;
```

```
        *first = (*first)-> next;
        free(tmp);
        tmp = NULL;
        freeList(&(*first));
}
```

Η κλήση της freeList γίνεται ως εξής:

```
freeList(&(V-> first));          //με όρισμα την διεύθυνση του δείκτη first της λίστας V
```

Ακολουθεί το τμήμα Main του κώδικα

```
int main(int argc, char *argv[]) {
        int a, b;
        List *V,*S,*W,*P,*A,*Solution;          //ορίζουμε ότι οι μεταβλητές V,S,W,P,A,
                                                //Solution είναι δείκτες προς LIST
        ListNode *tmp;                          //ορισμός ενός προσωρινού pointer
        tmp = (ListNode *)malloc(sizeof(ListNode));   //δεσμεύεται προσωρινός χώρος
                                                //μνήμης για νέο κόμβο
        initList(&V);                           //αρχικοποίηση λίστας V
        initList(&S);                           //αρχικοποίηση λίστας S
        initList(&W);                           //αρχικοποίηση λίστας W
        initList(&P);                           //αρχικοποίηση λίστας P
        initList(&A);                           //αρχικοποίηση λίστας A
        char fname[25];                         //ορισμός μεταβλητής fname με 25 σύμβολα
        readStr(fname);                 //διαβάζουμε στην μεταβλητή fname το όνομα του αρχείου
```

```
FILE * fp;                          //δηλώνουμε έναν δείκτη προς το αρχείο με όνομα fp
//File open and check if it is opened
fp = fopenfile(fname,"r");          //ανοίγουμε το αρχείο προς ανάγνωση
if (fp == NULL){                    //αν ο pointer fp είναι NULL
    printf("Error opening the file: %s \n", fname );   //εμφανίζεται μήνυμα ότι
                                                       //δεν άνοιξε επιτυχώς το αρχείο
    exit(EXIT_FAILURE);             //έξοδος από την εφαρμογή
 }
 //read file until EOF (End Of File)
      while(!feof(fp)){   //εφόσον δεν έχει έρθει το τέλος του αρχείου(End of File)
      fscanf(fp,"%d,%d",&a,&b);     //διαβάζουμε δεδομένα στις μεταβλητές a,b
            addToList(V,a);         //προσθέτουμε την τιμή της a στη λίστα V
            addToList(S,b);         //προσθέτουμε την τιμή της b στη λίστα S
   }
   //close file
   fclose(fp);                      //κλείσιμο του αρχείου
    printf("\n list V  count: %d",V-> count);          //εμφάνιση της λίστας V
    printf("\n  List V: \n");
    printList(&V);  //χρήση της συνάρτησης printList() για την εμφάνιση της λίστας V
    printf("\n  ********************************** \n");
    printf("\n list S  count: %d",S-> count);          //εμφάνιση της λίστας S
    printf("\n  List S: \n");
    printList(&S);  //χρήση της συνάρτησης printList() για την εμφάνιση της λίστας S
    printf("\n  ********************************** \n");
```

Υπολογισμός της λίστας W

```
calcW(&W, V);      //κλήσης της calcW για τον υπολογισμό της λίστας W (w=v-30)
                   //με ορίσματα την διεύθυνση της λίστας W και την λίστα V
printf("\n list W  count: %d", W-> count);          //εμφάνιση της λίστας W
printf("\n  List W: \n");
printList(&W);    //χρήση της συνάρτησης printList() για την εμφάνιση  λίστας W
```

Υπολογισμός της λίστας P

```
CP(P, V, W);              //κλήση συνάρτησης CP για τον υπολογισμό της  λίστας P
                          //με ορίσματα τις λίστες P,V,W
printf("\n  ********************************** \n");
printf("\n list P  count: %d", P-> count);          //εμφάνιση της λίστας P
printf("\n  List P: \n");
printList(&P);    //χρήση της συνάρτησης printList() για την εμφάνιση λίστας P
freeList(&(W-> first));          //κλήση της συνάρτησης freeList για την
                   //απελευθέρωση μνήμης της λίστας W που δεν χρειάζεται άλλο
```

Υπολογισμός της λίστας A

```
C_OPT(A, S, P);           //κλήση συνάρτησης C_OPT για τον υπολογισμό της
                          //λίστας A με ορίσματα τις λίστες A,S,P
printf("\n  ********************************** \n");
printf("\n list A  count: %d", A-> count);          //εμφάνιση της λίστας A
printf("\n  List A: \n");
printList(&A);    //χρήση της συνάρτησης printList() για την εμφάνιση λίστας A
```

Υπολογισμός και εμφάνιση της Λύσης

```
int i = A-> count;
printf("\nSolution: ");
findOpt(A, S, P, i);        //κλήση της συνάρτησης findOpt με ορίσματα τις λίστες
                            //υπολογίζει και εμφανίζει τη βέλτιστη λύση
```

Απελευθέρωση Μνήμης

```
freeList(&(V-> first));     //κλήση της συνάρτησης freeList για την
                            //απελευθέρωση μνήμης της λίστας V
freeList(&(S-> first));     //κλήση της συνάρτησης freeList για την
                            //απελευθέρωση μνήμης της λίστας S
freeList(&(P-> first));     //κλήση της συνάρτησης freeList για την
                            //απελευθέρωση μνήμης της λίστας P
freeList(&(A-> first));     //κλήση της συνάρτησης freeList για την
                            //απελευθέρωση μνήμης της λίστας A
printf("\n"); printf("\n");

system("PAUSE");
return 0;
}
```

ΒΙΒΛΙΟΓΡΑΦΙΚΕΣ ΑΝΑΦΟΡΕΣ

Levitin A. (2012*), Introduction to the Design and Analysis of Algorithms (3rd edition)*. USA: Pearson Education.

Σάββας Η. (2005), *Σημειώσεις για το μάθημα Αλγόριθμοι και Πολυπλοκότητα*. Λάρισα: ΤΕΙ Λάρισας.

Dasgupta S., Papadimitriou C. & Vazirani U. (2006), *Algorithms*. USA: McGraw-Hill.

Kleinberg J., Tardos E. (2006), Algorithm *Design.* USA: Pearson Education.

Dynamic Programming, (χ.χ.) Ανακτήθηκε 28 Αυγούστου, 2013 από το Dynamic Programming Wiki: http://en.wikipedia.org/wiki/Dynamic_programming

Εφαρμογές Βελτιστοποίησης και Επιχειρησιακής Έρευνας σε Προβλήματα Μηχανικών, (χ.χ.) Ανακτήθηκε 15 Οκτωβρίου, 2013 από το : http://users.teiath.gr/vmouss/ebooks/optimee/sections/section01_intro_i.html

Knapsack Problem, (χ.χ.) Ανακτήθηκε 25 Νοεμβρίου, 2013 από το Knapsack Problem Wiki: http://en.wikipedia.org/wiki/Knapsack_problem

Moler C. (2004), Numerical *Computing with MATLAB.* USA: Society for Industrial and Applied Mathematics.

Μποζάνης Π. (2006), *Αλγόριθμοι*. Ελλάδα: Εκδόσεις Τζιόλα.

Cormen T., Leiserson C. , Rivest R. & Stein C. (2009), *Introduction to Algorithms.* USA: MIT Press.

Ζάχος Ε. , *Αλγόριθμοι και Πολυπλοκότητα (2005).* Ελλάδα: ΕΜΠ

Δημητρίου Τ. (2001), *Παράλληλοι Αλγόριθμοι και Software.* Ελλάδα.

Leiss L. (2007), *A Programmer's Companion to Algorithm Analysis*. USA: Chapman & Hall/CRC.