ΤΕΙ ΚΑΛΑΜΑΤΑΣ
ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ

ΤΜΗΜΑ: ΤΕΧΝΟΛΟΓΙΑΣ
ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΑΝΑΛΥΣΗ ΚΑΙ ΣΥΓΚΡΙΣΗ ΑΛΓΟΡΙΘΜΩΝ ΤΑΞΙΝΟΜΗΣΗΣ

Επιβλέπων καθηγητής: Καραγιώργος Γρηγόρης

Φοιτήτρια: Καραγιάννη Κανέλλα Α.Μ. 2006035

Σπάρτη 2011

ΤΕΙ ΚΑΛΑΜΑΤΑΣ
ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ

ΤΜΗΜΑ: ΤΕΧΝΟΛΟΓΙΑΣ
ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΑΝΑΛΥΣΗ ΚΑΙ ΣΥΓΚΡΙΣΗ ΑΛΓΟΡΙΘΜΩΝ ΤΑΞΙΝΟΜΗΣΗΣ

Επιβλέπων καθηγητής: Καραγιώργος Γρηγόρης

Φοιτήτρια: Καραγιάννη Κανέλλα Α.Μ. 2006035

Σπάρτη 2011

Περιεχόμενα

1 ΠΕΡΙΛΗΨΗ

Στην παρούσα εργασία θα ασχοληθούμε με τους αλγορίθμους ταξινόμησης. Θα γίνει μια γενική εισαγωγή για τους αλγόριθμους, από που προήλθε η λέξη και πως εφαρμόζεται σήμερα στους ηλεκτρονικούς υπολογιστές. Στη συνέχεια θα ξεκινήσουμε την ανάλυση των αλγορίθμων ταξινόμησης όπως είναι οι Bubble Sort, Insert Sort, Select Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort και Shaker Sort. Θα αναφέρουμε βήμα - Βήμα τη λειτουργία τους, τα πλεονεκτήματα και τα μειονεκτήματα που ενδεχομένος να έχουν και θα συντάξουμε πηγαίο κώδικα για το κάθε αλγόριθμο σε γλώσσα C. Τέλος, θα γίνει σύγκριση όλων των αλγορίθμων που αναφέρθηκαν παραπάνω όπου θα διαπιστώσουμε ποιος αλγόριθμος είναι ο πιο αποτελεσματικός και ποιος αλγόριθμος είναι αναποτελεσματικός. Η σύνταξη της εργασίας πραγματοποιήθηκε σε λειτουργικό σύστημα Linux στο πρόγραμμα Latex, όπως απαιτούσε ο επιβλέπων καθηγητής της εργασίας.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Αλγόριθμος, πολυπλοκότητα, Bubble Sort, Select sort, Insert Sort, Heap Sort, Merge Sort, Quick Sort, Radix Sort, Shaker Sort, διαίρει και βασίλευε, σωρός.

2 ΕΙΣΑΓΩΓΗ

Για την ιστορία η λέξη αλγόριθμος προέρχεται από τον Πέρση μαθηματικό του 8ου αιώνα μ.Χ. Αλ.Χουαρίζμι, ο οποίος έγραφε συστηματικές τυποποιημένες λύσεις αλγεβρικών προβλημάτων σε διάφορα συγγράμματά του. Όλες οι τυποποιημένες οδηγίες άρχιζαν με την φράση «Ο αλγόριθμος λέει...», έτσι η λέξη αλγόριθμος καθιερώθηκε αργά τα επόμενα χίλια χρόνια με την έννοια «συστηματική διαδικασία αριθμητικών χειρισμών». Τη σημερινή της σημασία την οφείλει στη γρήγορη ανάπτυξη των ηλεκτρονικών υπολογιστών στα μέσα του 20ου αιώνα. Γενικά η έννοια ενός αλγόριθμου χρησιμοποιείται για να δηλώσει μεθόδους που εφαρμόζονται για την επίλυση προβλημάτων. Ωστόσο υπάρχει ένας πιο αυστηρός ορισμός της έννοιας αυτής που είναι ο εξής:
Αλγόριθμος είναι μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος. Κάθε αλγόριθμος απαραίτητα ικανοποιεί τα επόμενα κριτήρια:

1) Είσοδος (input). Καμία, μία ή και περισσότερες τιμές δεδομένων πρέπει να δίνονται ως είσοδοι στον αλγόριθμο. Η περίπτωση που δεν δίνονται τιμές δεδομένων εμφανίζεται όταν ο αλγόριθμος δημιουργεί και επεξεργάζεται κάποιες πρωτογενείς τιμές με τη βοήθεια των συναρτήσεων παραγωγής τυχαίων αριθμών, ή με την βοήθεια άλλων απλών εντολών.

2) Έξοδος (output). Ο αλγόριθμος πρέπει να δημιουργεί τουλάχιστον μία τιμή δεδομένων ως αποτέλεσμα προς το χρήστη ή προς έναν άλλο αλγόριθμο.

3) Καθοριστικότητα (definiteness). Κάθε εντολή πρέπει να καθορίζεται χωρίς καμία αμφιβολία για τον τρόπο εκτέλεσης της. Λόγου χάριν, μία εντολή διαίρεσης πρέπει να θεωρεί και την περίπτωση, όπου ο διαιρέτης λαμβάνει τη μηδενική τιμή.

4) Περατότητα (finiteness). Ο αλγόριθμος να τελειώνει μετά από πεπερασμένα βήματα εκτέλεσης των εντολών του. Μία διαδικασία που δεν τελειώνει μετά από ένα συγκεκριμένο αριθμό βημάτων δεν αποτελεί αλγόριθμο, αλλά λέγεται απλά υπολογιστική διαδικασία (computational procedure).

Η έννοια του αλγορίθμου γίνεται ευκολότερα αντιληπτή με το παρακάτω παράδειγμα:

Αν κάποιος επιθυμεί να γευματίσει θα πρέπει να εκτελέσει κάποια συγκεκριμένα βήματα: να συγκεντρώσει τα υλικά, να προετοιμάσει τα σκεύη μαγειρικής, να παρασκευάσει το φαγητό, να στρώσει το τραπέζι, να ετοιμάσει τη σαλάτα, να γευματίσει, να καθαρίσει το τραπέζι και να πλύνει τα πιάτα. Προφανώς, η προηγούμενη αλληλουχία οδηγεί στο επιθυμητό αποτέλεσμα. Δεν είναι όμως η μοναδική για την επίτευξη του σκοπού, αφού μπορεί να αλλάξει η σειρά των βημάτων (π.χ. πρώτα να ετοιμάσει τη σαλάτα και μετά να στρώσει το τραπέζι). Ωστόσο το νόημα είναι πως η κατάτμηση μιας σύνθετης εργασίας σε διακριτά βήματα που εκτελούνται διαδοχικά, είναι ο πιο πρακτικός τρόπος επίλυσης πολλών προβλημάτων.

3 ΤΟ ΑΝΤΙΚΕΙΜΕΝΟ ΤΗΣ ΕΡΓΑΣΙΑΣ

Αντικείμενο της παρούσας εργασίας είναι να πραγματοποιηθεί μια παρουσίαση των διαφόρων αλγορίθμων ταξινόμησης που υπάρχουν και να γίνει ανάλυση και σύγκριση των διαφορετικών τύπων τους.

Πιο συγκεκριμένα θα ασχοληθούμε με τους αλγόριθμους Bubble Sort, Merge Sort, Quick Sort, Select Sort, Insert Sort, και Heap Sort. Ακόμα αναφορικά θα ασχοληθούμε με τους Radix Sort και Shaker Sort.

Στόχοι της εργασίας είναι να κατανοήσουμε τη λειτουργία του κάθε αλγόριθμου, όπου η ανάπτυξή τους θα γίνει στη συνέχεια, και να διατυπώσουμε τα πλεονεκτήματα και τα μειονεκτήματα που κάνουν τον κάθε ένα ξεχωριστό. Στο τέλος της εργασίας θα πρέπει να γνωρίζουμε ποιος αλγόριθμος είναι ταχύτερος και ποιος χρονοβόρος.

4 ΑΛΓΟΡΙΘΜΟΙ ΤΑΞΙΝΟΜΗΣΗΣ

Οι αλγόριθμοι ταξινόμησης είναι αλγόριθμοι που τοποθετούν τα στοιχεία μίας λίστας με συγκεκριμένη σειρά, από τις οποίες οι πιο γνωστές είναι η αριθμητική και η λεξικογραφική σειρά. Η ταξινόμησή τους στην πληροφορική γίνεται με βάση διάφορων κριτηρίων όπως :

1) Την υπολογιστική πολυπλοκότητα των συγκρίσεων

2) Την υπολογιστική πολυπλοκότητα των ανταλλαγών

3) Την επαναληπτικότητα

4) Και την σταθερότητα

Όσον αφορά την ταξινομημένη λίστα που προκύπτει πρέπει να τηρούνται δύο κανόνες:

•Τα στοιχεία της λίστας πρέπει να είναι τοποθετημένα σε αύξουσα σειρά.

•Το αποτέλεσμα να περιέχει όλα τα στοιχεία της αρχικής λίστας, μόνο που θα είναι σε διαφορετική σειρά.

Οι αλγόριθμοι ταξινόμησης που χρησιμοποιούνται στην πληροφορική ταξινομούνται με βάση :

•Την υπολογιστική πολυπλοκότητα (worst, average and best behaviour) των συγκρίσεων των στοιχείων από την άποψη του μεγέθους της λίστας(n). Για χαρακτηριστικούς αλγόριθμους ταξινόμησης η καλή συμπεριφορά είναι O(n

log n) και η κακή συμπεριφορά είναι $\Omega(n^2)$. Η ιδανική συμπεριφορά για μία ταξινόμηση είναι O(n). Αλγόριθμοι ταξινόμησης οι οποίοι χρησιμοποιούν μόνο ένα αφηρημένο κλειδί για λειτουργίες σύγκρισης χρειάζονται πάντα τουλάχιστον $\Omega(n \log n)$ συγκρίσεις κατά μέσον όρο.

•Την υπολογιστική πολυπλοκότητα των ανταλλαγών (για "in place" αλγορίθμους).

•Την χρήση μνήμης και άλλων υπολογιστών πόρων. Ειδικότερα, μερικοί αλγόριθμοι ταξινόμησης are "in place",έτσι ώστε μόνο O(1) ή O(log n) μνήμη απαιτείται πέρα από τα στοιχεία που ταξινομούνται, ενώ άλλοι πρέπει να δημιουργήσουν τις βοηθητικές θέσεις για τα στοιχεία που αποθηκεύονται προσωρινά.

5 ΤΕΧΝΙΚΕΣ

[2] Πριν αρχίσουμε την εξέταση συγκεκριμένων αλγορίθμων, ας κάνουμε ένα βήμα πίσω και ας εξετάσουμε το πρόβλημα της ταξινόμησης σφαιρικότερα. Θέλουμε να τοποθετήσουμε n πράγματα στη σειρά, ποιες είναι οι γενικές τεχνικές που θα κάνουν κάτι τέτοιο; Στο μοντέλο μας υπάρχουν μόνο δύο πράγματα που μπορούμε να κάνουμε σε στοιχεία: να τα συγκρίνουμε ή να τα μετακινήσουμε. Υποθέστε ότι διαιρούμε την είσοδο σε δύο τμήματα. Υπάρχουν μόνο τέσσερις ιδιότητες που μπορεί να έχει αυτή η διαίρεση και από τις τέσσερις μόνο οι δύο είναι οι σημαντικές. Πρώτον μπορεί να υπάρχει ή να μην υπάρχει κατάταξη μεταξύ δύο τμημάτων - όλα τα στοιχεία του ενός τμήματος μπορεί να

είναι μικρότερα από όλα τα στοιχεία του άλλου, ή όχι. Δεύτερον, μπορεί να υπάρχει ή να μην υπάρχει κατάταξη μεταξύ των στοιχείων κάθε τμήματος - όλα τα στοιχεία σε ένα τμήμα μπορεί να είναι στη σειρά ή όχι. Συνεπώς υπάρχουν έξι πιθανά είδη κατάταξης (οι άλλες δύο ιδιότητες είναι ότι τα δύο τμήματα μπορεί να έχουν διαφορετικά μεγέθη και ότι τα στοιχεία των δύο τμημάτων μπορεί να είναι γειτονικά στη λίστα ή όχι. Κάθε επιλογή διατμηματικής κατάταξης, ενδοτμηματικής κατάταξης, μεγέθους και γειτνίασης οδηγεί σε έναν αλγόριθμο ταξινόμησης). Από αυτές μόνο τέσσερις είναι σημαντικές, οι άλλες δύο δεν δίνουν αλγορίθμους ταξινόμησης. (Να σημειωθεί ότι επειδή εξετάζουμε μόνο την κατάταξη, ένας αλγόριθμος μπορεί να ανήκει σε περισσότερες από μία ομάδες ανάλογα με τα μεγέθη των υπολογιστών και με το αν τα στοιχεία μίας υπολίστας είναι γειτονικά.)

Οι τέσσερις σημαντικές περιπτώσεις κατάταξης είναι:

•Δεν υπάρχει κατάταξη μεταξύ των τμημάτων:

–Το ένα τμήμα είναι ταξινομημένο. Αυτή είναι η **ταξινόμηση με εισαγωγή** (insert sort). Παράδειγμα είναι η γραμμική ταξινόμηση με εισαγωγή, η δυαδική ταξινόμηση με εισαγωγή και η ταξινόμηση με εισαγωγή με βήμα k.

–Και τα δύο είναι ταξινομημένα. Αυτή είναι η **ταξινόμηση με συγχώνευση** (merge sort). Ένα παράδειγμα είναι η ταξινόμηση με συγχώνευση.

•Υπάρχει κατάταξη μεταξύ των τμημάτων:

–Κανένα από τα τμήματα δεν είναι ταξινομημένο. Αυτή είναι η **ταξινόμηση**

με διαίρεση (split sort).Ένα παράδειγμα είναι η γρήγορη ταξινόμηση (quick sort).

–Ένα τμήμα είναι ταξινομημένο. Αυτή είναι η **ταξινόμηση με επιλογή** (select sort). Παράδειγμα είναι η ταξινόμηση φυσαλίδας (bubble sort), η γραμμική ταξινόμηση με επιλογή και η ταξινόμηση με σωρό (heap sort).

Καθεμία από τις τέσσερις αυτές τεχνικές ταξινόμησης πήρε το όνομά της από το πιο δαπανηρό στάδιο της ταξινόμησης. Στη ταξινόμηση με συγχώνευση, η συγχώνευση "κοστίζει" περισσότερο από τη διαίρεση, στην ταξινόμηση με διαίρεση, η διαίρεση κοστίζει περισσότερο από της συγχώνευσης σε μια ταξινόμηση με εισαγωγή, η εισαγωγή κοστίζει περισσοτερο από την επιλογή και σε μια ταξινόμηση με επιλογή, η επιλογή κοστίζει περισσότερο από την εισαγωγή. Οι τεχνικές της ταξινόμησης με συγχώνευση και της ταξινόμησης με διαίρεση υποδεικνύουν αναδρομικούς αλγόριθμους, δεδομένου ότι τα δύο υποπροβλήματα είναι όμοια. Οι τεχνικές της ταξινόμησης με εισαγωγή και της ταξινόμησης με επιλογή υποδεικνύουν επαναληπτικούς αλγορίθμους, δεδομένου ότι τα δύο υποπροβλήματα είναι διαφορετικά. Σε μία ταξινόμηση με εισαγωγή, τα στοιχεία στην ταξινομημένη υπολίστα δεν έχουν κάποια ιδιαίτερη κατάταξη, άρα ένας τρόπος να κάνουμε ταξινόμηση με εισαγωγή είναι να επιλέγουμε κατ' επανάληψη ένα τυχαίο στοιχείο της μη ταξινομημένης υπολίστας που έχει απομείνει και να το εισάγουμε στην ήδη ταξινομημένη υπολίστα. Σε μία ταξινόμηση με επιλογή, τα στοιχεία στην ταξινομημένη υπολίστα έχουν μια ιδιαίτερη κατάταξη - οποιαδήποτε κατάταξη μας κάνει αλλά η μεγαλύτερη (ή η μικρότερη) είναι πιο

εύκολο να βρεθεί - άρα ένας τρόπος να κάνουμε ταξινόμηση με επιλογή είναι να επιλέγουμε κατ' επανάληψη το μεγαλύτερο στοιχείο από την μη ταξινομημένη υπολίστα που έχει απομείνει και να το εισάγουμε στην ήδη ταξινομημένη υπολίστα.

Υπάρχουν και άλλοι τρόποι να ταξινομήσουμε πράγματα.
Για παράδειγμα, μπορούμε να συγκρίνουμε κάθε ζεύγος στοιχείων, και μετά να διατάξουμε τα στοιχεία με βάση τον αριθμό των στοιχείων από τα οποία είναι μικρότερα. Αυτή είναι η **ταξινόμηση με μέτρηση** (count sort). Δεδομένου ότι μία ταξινόμηση με μέτρηση συγκρίνει κάθε ζεύγος στοιχείων, είναι $\Theta(n^2)$. Ένας άλλος τρόπος να ταξινομήσουμε πράγματα είναι να αντιμεταθέσουμε κάθε δύο στοιχεία τα οποία δεν είναι στη σωστή σειρά το ένα με το άλλο. Αυτή είναι **η ταξινόμηση με αντιμετάθεση** (swap sort). Ακόμα θα μπορούσαμε να εγκαταλείψουμε εντελώς το μοντέλο συγκρίσεων - αντιμεταθέσεων και να χρησιμοποιήσουμε μία ειδική ιδιότητα των στοιχείων της εισόδου. Για παράδειγμα, για να ταξινομήσουν πορτοκάλια ως προς το μέγεθός τους, οι συσκευαστές φρούτων τα κυλούν μεταξύ δύο σωληνών που αποκλίνουν σιγά - σιγά και κάτω από τους οποίους υπάρχει μία σειρά κουτιών. Κάθε πορτοκάλι συνεχίζει να κυλάει έως ότου οι σωλήνες αποκλίνουν αρκετά ώστε να πέσει στο κουτί που βρίσκεται από κάτω. Με αυτόν τον τρόπο ταξινομούνται πράγματα στη βιομηχανία ως προς το μέγεθός τους όταν αυτά τα πράγματα μπορούν να κυλούν. Όπως καταλαβαίνετε, η παράκαμψη των κανόνων μπορεί να μας προμηθεύσει με πραγματικά αποτελεσματικές λύσεις, ωστόσο, επειδή εκμεταλλεύεται κάτι άλλο

και όχι τις συγκρίσεις, δε θα δουλέψει για όλες τις εισόδους. Δε θα μπορούσαμε να ταξινομήσουμε τα πορτοκάλια τόσο εύκολα χωρίς τη βαρύτητα- θα έπρεπε να τη μιμηθούμε με την αναρρόφηση ή κάτι παρόμοιο. Στο μοντέλο συγκρίσεων - αντιμεταθέσεων, άλλες τεχνικές ταξινόμησης έχουν λιγότερες απαιτήσεις από την ενδιάμεση ταξινόμηση: στην κατηγοριοποίησή μας είτε ταξινομούμε είτε δεν ταξινομούμε κάθε τμήμα, μπορούμε όμως να διαιρέσουμε κάθε τμήμα σε περαιτέρω τμήματα και να κάνουμε το ίδιο. Γενικά, στο μοντέλο συγκρίσεων - αντιμεταθέσεων, οι ταξινομήσεις με μέτρηση και με αντιμετάθεση είναι οι λιγότερο αποδοτικές, οι ταξινομήσεις με εισαγωγή και με επιλογή είναι μετρίως αποδοτικές και οι ταξινομήσεις με συγχώνευση και με διαίρεση είναι οι περισσότερο αποδοτικές. Ένα προτέρημα της ταξινόμησης με εισαγωγή ως προς την ταξινόμηση με επιλογή είναι ότι μπορούμε να τη χρησιμοποιήσουμε on-line: μπορούμε να ξεκινήσουμε την ταξινόμηση ακόμα και χωρίς να έχουμε όλα τα στοιχεία. Ένα προτέρημα της ταξινόμησης με επιλογή ως προς την ταξινόμηση με εισαγωγή είναι ότι μπορούμε πάντα τα βρούμε τα i μεγαλύτερα στοιχεία ακόμα και πριν το τέλος της ταξινόμησης.

6 ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ

6.1 BUBBLE SORT (ΤΑΞΙΝΟΜΗΣΗ ΦΥΣΑΛΙΔΑΣ)

Η ταξινόμηση φυσαλίδας είναι η παλαιότερη και απλούστερη ταξινόμηση σε χρήση. Δυστυχώς, είναι επίσης ο πιο αργός. Το είδος της φυσαλίδας λειτουργεί συγκρίνοντας κάθε στοιχείο στη λίστα με το στοιχείο δίπλα του, και εναλλάσσεται, εφόσον απαιτείται. Ο αλγόριθμος επαναλαμβάνει αυτή τη διαδικασία μέχρι να κάνει ένα πέρασμα σε όλη τη διαδρομή του καταλόγου χωρίς αλλάζουν οποιαδήποτε στοιχεία (με άλλα λόγια, όλα τα στοιχεία βρίσκονται στη σωστή σειρά). Αυτό προκαλεί μεγαλύτερες τιμές στο τέλος της λίστας, ενώ μικρότερες τιμές "βυθίζονται"προς την αρχή της λίστας. Η ταξινόμηση φυσαλίδας θεωρείται γενικά ως ο πλέον αναποτελεσματικός αλγόριθμος διαλογής στην κοινή χρήση. Στη καλύτερη περίπτωση (που ο κατάλογος είναι ήδη ταξινομημένος), το είδος της φυσαλίδας μπορεί να απευθύνεται σε σταθερά O (n) βαθμό πολυπλοκότητας. Γενικά, η υπόθεση αυτή αποτελεί ένα αβυσσαλέο χάσμα O (n^2). Ενώ παρακάτω θα δούμε ότι η εισαγωγή (insertion) , η επιλογή (selection) και το κέλυφος (shell) επίσης έχουν O (n^2) πολυπλοκότητα, αλλά είναι πολύ πιο αποτελεσματικές από το είδος της φυσαλίδας (bubble).

Αν και η ταξινόμηση φυσαλίδας είναι ένα από τα απλούστερα είδη αλγορίθμων ταξινόμησης για να κατανοήθουν και να εφαρμόσθουν, με πολυπλοκότητα όμως O (n^2) σημαίνει ότι είναι πάρα πολύ αναποτελεσματική για χρήση σε καταλόγους που έχουν περισσότερα από μερικά στοιχεία. Ακόμη και μεταξύ των απλών αλγόριθμων ταξινόμησης ,όπως ταξινόμηση με εισαγωγή, συνήθως είναι

πολύ πιο αποτελεσματικοί, εάν τα στοιχεία είναι ήδη σχεδόν σε ταξινομημένη σειρά. Λόγω της απλότητάς του, χρησιμοποιείται συχνά για να εισαχθεί η έννοια ενός αλγορίθμου, σε φοιτητές που βρίσκονται στην εισαγωγική επιστήμη των υπολογιστών. Εντούτοις, μερικοί ερευνητές , όπως ο Όουεν Αστραχάν, έχουν δυσφημήσει την ταξινόμηση της φυσαλίδας συνιστώντας ότι πλέον δεν πρέπει να διδάσκεται στην εκπαίδευση της επιστήμης των υπολογιστών. Τα μοναδικά **πλεονεκτήματά** στην ταξινόμηση φυσαλίδας είναι η απλότητά της και η ευκολία εφαρμογής της. [2] Άραγε είναι εγκυημένο ότι αυτή η διαδικασία θα ταξινομήσει μια λίστα; Αν, μετά από κάποια εξέταση της εισόδου, δεν υπάρχει ζεύγος γειτωνικών στοιχείων τα οποία να μην είναι στη σειρά, η λίστα είναι ταξινομημένη. Επίσης κάθε φορά που συναντάμε το μεγαλύτερο από τα στοιχεία που δεν είναι ακόμα ταξινομημένα, το προωθούμε με αντιμεταθέσεις προς τα πάνω στη λίστα, δεδομένου ότι, ανεξάρτητα από το που βρίσκεται, αυτό το στοιχείο δεν είναι ποτέ στη σειρά ως προς το γείτονά του. Άρα ο αλγόριθμος Bubble Sort, όχι μόνο μειώνει την αταξία, αλλά και μετά από κάθε ανίχνευση της εισόδου τοποθετεί το επόμενο μεγαλύτερο στοιχείο στη τελική του θέση. Άρα ο Bubble Sort είναι στη πραγματικότητα μια συγκεκαλυμμένη ταξινόμηση με επιλογή. Η i- οστή ανίχνευση της εισόδου κοστίζει μέχρι n-i συγκρίσεις και υπάρχουν n-1 διαφορετικές ανιχνεύσεις της εισόδου, άρα ο Bubble Sort μπορεί να χρησιμοποιήσει μέχρι:

$\sum_{i=1}^{n-1}(n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$

συγκρίσεις και αντιμεταθέσεις.

Μπορούμε να βελτιώσουμε αυτόν τον αλγόριθμο αλλά δεν αξίζει τον κόπο

να το κάνουμε. Ο Bubble sort δεν είναι ο καλύτερος από οποιονδήποτε άλλο αλγόριθμο ταξινόμησης που θα αναφέρω παρακάτω. Και παρόλο που είναι εύκολο να γραφτεί σε πρόγραμμα, καλός είναι μόνο όταν η είσοδος είναι σχεδόν ταξινομημένη από την αρχή.

6.1.1 ΒΗΜΑ-ΒΗΜΑ Η ΛΕΙΤΟΥΡΓΙΑ ΤΗΣ

Ας πάρουμε με τη σειρά τους αριθμούς "5 1 4 2 8", και ας ταξινομήσουμε τον πίνακα από το μικρότερο αριθμό στο μεγαλύτερο αριθμό, χρησιμοποιώντας τον αλγόριθμο. Σε κάθε βήμα οι έντονοι αριθμοί είναι και αυτοί που συγκρίνονται.

Πρώτη Φάση:

Ξεκινάμε συγκρίνοντας δύο δύο τα στοιχεία μας.

(**5 1** 4 2 8)→(**1 5** 4 2 8)

(1 **5 4** 2 8)→(1 **4 5** 2 8)

(1 4 **5 2** 8)→(1 4 **2 5** 8)

(1 4 2 **5 8**)→(1 4 2 **5 8**)

Δεύτερη Φάση:

(**1 4** 2 5 8)→(**1 4** 2 5 8)

(1 **4 2** 5 8)→(1 **2 4** 5 8)

(1 2 **4 5** 8)→(1 2 **4 5** 8)

(1 2 4 **5 8**)→(1 2 4 **5 8**)

Τώρα, η σειρά είναι ήδη ταξινομημένη, αλλά ο αλγόριθμος μας δεν ξέρει αν έχει ολοκληρωθεί, χρειάζεται να γίνει άλλη μία επαλήθευση για να καταλάβει εάν είναι ταξινομημένο.

Τρίτη Φάση:

(**1 2** 4 5 8)→(**1 2** 4 5 8)

(1 **2 4** 5 8)→(1 **2 4** 5 8)

(1 2 **4 5** 8)→(1 2 **4 5** 8)

(1 2 4 **5 8**)→(1 2 4 **5 8**)

Τέλος, ο πίνακας είναι ταξινομημένος, και ο αλγόριθμος τερμάτησε.

6.1.2 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
[5] bubbleSort (p, plithos)
    char *p;
    int plithos;
    {
    int a, b;
    char x;

    for (a=1; a<plithos; ++a)
    for (b= plithos-1; b>=a; –b)
    {
    if (p[b-1]>p[b])
    {
    x=p [b-1]; p[b-1]=p[b]; p[b]=x;
    }
    }
    }
```

6.2 INSERT SORT (ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΕΙΣΑΓΩΓΗ)

Σε μία ταξινόμηση με εισαγωγή κατασκευάζουμε σταδιακά μία ταξινομημένη υπολίστα σε ένα συνεχόμενο τμήμα της λίστας. Μετά τοποθετούμε κάθε νέο στοιχείο στη σωστή του θέση ως προς την ήδη ταξινομημένη υπολίστα. Οι τρόποι που αποφασίζουμε να επιλέξουμε το επόμενο στοιχείο και να αναζητήσουμε τη σωστή σχετική θέση του στοιχείου αυτού, καθορίζουν τους διαφορετικούς τύπους των ταξινομήσεων με εισαγωγή (γραμμική ταξινόμηση με εισαγωγή και δυαδική ταξινόμηση με εισαγωγή).

6.2.1 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
void insertion_sort(int a[], int length)
    {
    int i;
    for (i=0; i ¡ length; i++)
    {
    int j, v = a[i];
    for (j = i - 1; j >= 0; j )
    {
    if (a[j] <= v) break;
```

```
a[j + 1] = a[j];
}
a[j + 1] = v;
}
}
```

6.2.2 ΓΡΑΜΜΙΚΗ ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΕΙΣΑΓΩΓΗ

[2] Ο πιο απλός τρόπος για να βρούμε τη σχετική θέση του επόμενου μη τοποθετημένου στοιχείου είναι να χρησιμοποιήσουμε γραμμική αναζήτηση. Στο i -οστό βήμα, εισάγουμε το επόμενο στοιχείο στη σωστή του θέση ως προς τα ήδη ταξινομημένα i -1 στοιχεία. Αυτό κοστίζει το πολύ i -1 συγκρίσεις. Επαναλαμβάνουμε για i από το 2 έως και το n. Αυτή είναι η *γραμμική ταξινόμηση με εισαγωγή*.

Η i-οστή ανίχνευση της εισόδου κοστίζει μέχρι i -1 συγκρίσεις και αντιμεταθέσεις και υπάρχουν n -1 ανιχνεύσεις. Συνεπώς, ο INSERT_SORT χρησιμοποιεί μέχρι

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

συγκρίσεις και αντιμεταθέσεις. (Ο αριθμός των αντιμεταθέσεων είναι ίδιος με τον αριθμό των συγκρίσεων τόσο στη μέση όσο και στη χειρότερη περίπτω-

ση. Άρα δεν έχουμε πραγματική βελτίωση σε σχέση με τον αλγόριθμο BUBBLE_SORT).

Η συνάρτηση, η οποία ακολουθεί αναζητεί μέσα σε έναν μονοδιάστατο πίνακα τύπου char γνωστού μήκους ένα στοιχείο, το οποίο πρέπει να συμπίπτει με το δοθέν κλειδί:

```
sequential (p. plithos, kleidi)
char *p;
int plithos;
char kleidi;
{
int i;

for (i=0; i<plithos; ++i)
if (kleidi== p[i]) return(i);

return(-1); /* δεν υπάρχει στοιχείο ίδιο με το κλειδί */
}
```

6.2.3 ΔΥΑΔΙΚΗ ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΕΙΣΑΓΩΓΗ

[2] Αντί να χρησιμοποιήσουμε γραμμική αναζήτηση για την εισαγωγή, γιατί να μην χρησιμοποιήσουμε δυαδική αναζήτηση· Η διαδική αναζήτηση απαιτεί μέχρι [lg(i+1)] συγκρίσεις για να εισάγει το i+1 - οστό στοιχείο, άρα ο μέγιστος αριθμός συγκρίσεων της δυαδικής ταξινόμησης με εισαγωγή είναι:

$$\sum_{i=0}^{n-1}[lg(i+1)] = \sum_{i=1}^{n}[lgi] = n[lgn] - 2^{[lgn]} + 1 = O(nlgn)$$

Υπάρχει μεγάλη διαφορά μεταξύ του n^2 και του n lg n, το lg n αυξάνεται τόσο αργά σε σύγκριση με το n ώστε το n lg n είναι "σχεδόν γραμμικό". Για παράδειγμα εάν το n είναι ένα εκατομμύριο το n lg n είναι περίπου είκοσι εκατομμύρια, αλλά το n^2 είναι ένα τρισεκατομμύριο. Εάν το n είναι ένα εκατομμύριο και μια σύγκριση χρειάζεται ένα μικρό δευτερόλεπτο, τότε n lg n συγκρίσεις χρειάζονται περίπου είκοσι δευτερόλεπτα, αλλά οι n^2 συγκρίσεις χρειάζονται περισσότερο από εντεκάμιση μέρες.

Αυτή είναι η μεγάλη βελτίωση στο χειρότερο αριθμό συγκρίσεων, αλλά τι γίνεται με τις αντιμεταθέσεις· Δεδομένου ότι ταξινομούμε μέσα σε έναν πίνακα, μετά τη χρησιμοποίηση της δυαδικής αναζήτησης για να βρούμε τη σωστή θέση του i -οστού στοιχείου, θα πρέπει να μετατοπίσουμε τα i -1 στοιχεία για να του κάνουμε χώρο. Άρα η ταξινόμηση εξακολουθεί να χρειάζεται $\Omega(n^2)$ αντιμεταθέσεις στη χειρότερη περίπτωση. Αυτό μπορεί να μειωθεί με τη χρήση

συμπληρωματικών δεικτών για να παρακολουθούμε τις σωστές σχετικές θέσεις κάθε στοιχείου, αυτό όμως είναι δαπανηρό γιατί αυτές οι θέσεις μπορεί να αλλάζουν μετά από κάθε εισαγωγή.

Το ακόλουθο παράδειγμα απεικονίζει τη μέθοδο της δυαδικής ταξινόμησης όταν αναζητείται το κλειδί Τ:

Α Α Α Β Γ Γ Γ Δ Κ Ν Π Ρ Ρ Τ Χ Χ Υ

/

Ν Π Ρ Ρ Τ Χ Χ Υ

/

Τ Χ Χ Υ

/

Τ

Ακολουθεί η συνάρτηση δυαδικής αναζήτησης:

```
binary(p, plithos, kleidi)
char *p;
int plithos;
char kleidi:
{
```

```
int low. high. mid;

low=0; high=plithos-1;

while (low<=high)
{
mid=(low+high)/2;

if (kleidi<p[mid]) high=mid-1;
else if (kleidi>p[mid]) low=mid+1;
else return(mid);
}
return(-1);
}
```

6.2.4 ΠΛΕΟΝΕΚΤΗΜΑΤΑ INSERT SORT

Αυτός ο απλός αλγόριθμος ταξινόμησης έχει τα εξής πλεονεκτήματα:

- Απλή εφαρμογή.

- Αποτελεσματικός σε πολύ μικρά σύνολα δεδομένων.

∘Προσαρμοστικός για σύνολα δεδομένων που είναι ήδη ταξινομημένα:όπου η πολυπλοκότητα του χρόνου είναι O (n + d), όπου d είναι ο αριθμός των αναστροφών.

∘Πιο αποτελεσματικός στην πράξη δηλαδή O (n^2), από τους περισσότερους αλγόριθμους όπως την ταξινόμηση επιλογής ή την ταξινόμηση φυσαλίδας που έχουν O (n).

∘Σταθερός , δηλαδή δεν αλλάζει τη σχετική σειρά των στοιχείων.

6.2.5 Η ΛΕΙΤΟΥΡΓΙΑ ΤΟΥ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΠΑΡΑΔΕΙΓΜΑ

Ο παρακάτω πίνακας δείχνει τα βήματα για την ταξινόμηση της ακολουθίας 5 7 0 3 4 2 6 1. Στην αριστερή πλευρά της ακολουθίας εμφανίζεται το ταξινομημένο μέρος με έντονα γράμματα. Για χάθε επανάληψη, ο αριθμός των θέσεων του εισαχθέντος στοιχείου έχει μετακινηθεί εντός παρενθέσεως. Συνολικά το ποσό αυτό ανέρχεται σε 17 βήματα.

5 7 0 3 4 2 6 1 (0)

5 7 0 3 4 2 6 1 (0)

0 5 7 3 4 2 6 1 (2)

0 3 5 7 4 2 6 1 (2)

0 3 4 5 7 2 6 1 (2)

0 2 3 4 5 7 6 1 (4)

0 2 3 4 5 6 7 1 (1)

0 1 2 3 4 5 6 7 (6)

6.3 SELECT SORT (ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΕΠΙΛΟΓΗ)

Η Selection Sort είναι ένας αλγόριθμος διαλογής. Έχει O (n^2) πολυπλοκότητα, γεγονός που τον καθιστά αναποτελεσματικό σε μεγάλες λίστες και γενικά εκτελεί χειρότερα σε σχέση με την ανάλογη ταξινόμηση με εισαγωγή. Χαρακτηρίζεται για την απλότητά του, και επίσης την επίδοσή του όπου είναι καλύτερη, σε σύγκριση με άλλους πολύπλοκους αλγόριθμους, ιδίως όταν η βοηθητική μνήμη είναι περιορισμένη.

Ας εξετάσουμε λεπτομερέστερα την τεχνική της ταξινόμησης με επιλογή. Σε κάθε βήμα βρίσκουμε το μεγαλύτερο από τα υπόλοιπα μη διατεταγμένα στοιχεία. Αλλά μετά από κάθε ανίχνευση της εισόδου, χαραμίζουμε πολλές χρήσιμες πληροφορίες αφού η επόμενη ανίχνευση αφού η επόμενη ανίχνευση δε ξεκινά με καμία πληροφορία σχετικά με το δεύτερο μεγαλύτερο στοιχείο (το οποίο είναι

τώρα το μεγαλύτερο από τα υπόλοιπα στοιχεία). Εάν είμαστε προσεκτικοί μετά την εύρεση του μεγαλύτερου στοιχείου, το δεύτερπο μεγαλύτερο μπορεί να είναι μόνο ένα από τα [lg n] στοιχεία, και μπορούμε να προσδιορίσουμε αυτά τα στοιχεία από το πρώτο "διάβασμα" της εισόδου. Άρα ας κρατήσουμε πληροφορίες για τη σχετική διάταξη των στοιχείων όταν βρίσκουμε το μεγαλύτερο και στη συνέχεια, ας χρησιμοποιήσουμε αυτήν την πληροφορία όταν βρίσκουμε το δεύτερο καλύτερο στον επόμενο γύρο, και ούτο καθεξής. Για να το κάνουμε αυτό χρειαζόμαστε μια δομή η οποία θα μας επιτρέψει να κρατάμε πληροφορία για τη διάταξη των στοιχείων.

[2]

Με αυτήν την τεχνική αντί να ασχολούμαστε με ένα αυθαίρετο στοιχείο, βρίσκουμε το μεγαλύτερο στοιχείο και το βάζουμε στη σωστή του θέση. Στη συνέχεια επαναλαμβάνουμε τη διαδικασία με τα στοιχεία που έχουν απομείνει. Ένα **πλεονέκτημα** μιας ταξινόμησης με επιλογή ως προς μια ταξινόμηση με εισαγωγή είναι ότι όταν βρούμε μία θέση για ένα στοιχείο, αυτή θα είναι και η τελική του θέση. Ένα **μειονέκτημα** μιας ταξινόμησης με επιλογή ως προς μία ταξινόμηση με εισαγωγή είναι ότι δεν οφείλεται από εισόδους που είναι μερικώς διατεταγμένες. Επίσης, μια ταξινόμηση με επιλογή πρέπει να γίνει off-line (μπορεί μόνο όταν έχουμε όλα τα στοιχεία).

Έστω η i- οστή επανάληψη του αλγορίθμου. Σε αυτό το σημείο, τα μεγαλύτερα i-1 στοιχεία είναι ήδη με τη σωστή σειρά στις τελευταίες i-1 θέσεις στη

λίστα. Ο Select Sort βρίσκει ξανά και ξανά το μεγαλύτερο από τα στοιχεία που έχουν απομείνει έως ότου δεν έχει απομείνει πια κανένα. Η i- οστή επανάληψη κοστίζει μέχρι n-i συγκρίσεις, συνεπώς ο αλγόριθμος αυτός κοστίζει μέχρι

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = n(n-1)/2 = O(n^2)$$

συγκρίσεις. Ωστόσο ο Select sort χρησιμοποιεί μόνο O(n) αντιμεταθέσεις. Οι μέσοι αριθμοί συγκρίσεων και αντιμεταθέσεων είναι ίσοι με αυτούς της χειρότερης περίπτωσης.

Όπως και με τον insert sort, ένας λόγος για να χρησιμοποιήσουμε τον select sort είναι ότι είναι εύκολο να γραφτεί σε πρόγραμμα. Επίσης, και οι δύο αλγόριθμοι είναι τόσο εύκολοι που έχουν χαμηλό γενικό κόστος, άρα και οι δύο είναι αποδοτικοί για μικρά n.

6.3.1 ΛΕΙΤΟΥΡΓΙΑ ΑΛΓΟΡΙΘΜΟΥ

Ο αλγόριθμος λειτουργεί ως εξής:

1) Βρίσκουμε την ελάχιστη τιμή στη λίστα

2) Την ανταλλάσουμε με την τιμή στην πρώτη θέση

3) Επαναλαμβάνουμε τα δύο βήματα για το υπόλοιπο της λίστας, ξεκινώντας

από την δεύτερη θέση και την προώθηση κάθε φορά.

Ουσιαστικά, ο κατάλογος χωρίζεται σε δύο μέρη:

1) Στην δευτερεύουσα λίστα των στοιχείων που είναι ήδη ταξινομημένα και

2) Στην δευτερεύουσα λίστα των στοιχείων που απομένουν για ταξινόμηση, καταλαμβάνοντας το υπόλοιπο της συστοιχίας.

Αυτό είναι ένα παράδειγμα του αλγορίθμου ταξινόμησης με πέντε στοιχεία:

64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

11 12 22 25 64

Η Ταξινόμηση με επιλογή μπορεί επίσης να χρησιμοποιηθεί για τις δομές με λίστα που προσθέτουν και αφαιρούν αποτελεσματικά, όπως μια συνδεδεμένη λίστα. Σε αυτή την περίπτωση είναι πιο σύνηθες να αφαιρέσουμε το ελάχιστο στοιχείο από το υπόλοιπο της λίστας, και στη συνέχεια να το τοποθετήστε στο τέλος των ταξινομημένων στοιχείων μέχρι τώρα. Για παράδειγμα:

64 25 12 22 11

11 64 25 12 22

11 12 64 25 22

11 12 22 64 25

11 12 22 25 64

6.3.2 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

[5]

```
int iPos; int iMin;
for (iPos = 0; iPos <n; iPos++)
{
iMin = iPos;
for (i = iPos+1; i <n; i++)
{
if (a[i] <a[iMin])
{
iMin = i;
}
}
if ( iMin != iPos )
{
```

```
swap(a, iPos, iMin);
}
}
```

6.4 HEAP SORT (ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΣΩΡΟ)

[1] Ένας άλλος γνωστός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος ταξινόμησης με σωρό (heap sort), ο οποίος αποτελεί μια βελτιωμένη παραλλαγή του αλγόριθμου επιλογής, που περιγράφηκε προηγουμένως. Όπως και ο αλγόριθμος επιλογής, έτσι και αυτός ο αλγόριθμος δουλεύει ψάχνοντας το μεγαλύτερο (ή μικρότερο) στοιχείο της λίστας, τοποθετώντας το στο τέλος (ή στην αρχή) και συνεχίζει με το υπόλοιπο της λίστας. Η διαφορά είναι, όμως, ότι ο αλγόριθμος heapsort εκτελεί αυτή τη διαδικασία πιο αποτελεσματικά χρησιμοποιώντας ένα τύπο δεδομένων που ονομάζεται heap, που ουσιαστικά είναι ένας ειδικός τύπος δυαδικού δέντρου. Μόλις τα στοιχεία της λίστας σχηματίσουν το heap, τότε η ρίζα του δέντρου είναι το μεγαλύτερο στοιχείο. Τότε αφαιρείται και τοποθετείται στο τέλος της λίστας και σχηματίζεται ξανά το heap με αποτέλεσμα η ρίζα του δέντρου να είναι πάλι το μεγαλύτερο στοιχείο. Χρησιμοποιώντας το heap για να βρεθεί το μεγαλύτερο στοιχείο της λίστας απαιτείται O(log n) χρόνος αντί για O(n) που χρειάζεται για μια σειριακή σάρωση στον απλό αλγόριθμο επιλογής. Αυτό επιτρέπει στον heapsort να εκτελείται σε χρόνο O(n log n).

Η σωρός είναι μια δενδρική δομή και χρησιμοποιείται για τη δημιουργία ου-

ρών προτεραιότητας (priority queues). Η ρίζα του δέντρου περιλαμβάνει το μικρότερο-μεγαλύτερο στοιχείο του αναλόγως αν έχουμε σωρό ελαχίστων ή μεγίστων. Τα επόμενα δύο στοιχεία του δένδρου είναι τα παιδιά του. Γενικότερα αν ο πατέρας είναι στη θέση i τα παιδιά του θα είναι στην θέση 2i(αριστερό παιδί) και 2i+1(δεξί παιδί) αντίστοιχα. Αν i η θέση ενός παιδιού i/2 είναι η θέση του πατέρα του. Κάθε σωρός με ν στοιχεία έχει ύψος log 2 n. Ο σωρός, όπως και τα δέντρα γενικότερα, μπορεί να υλοποιηθεί με πίνακα, στον οποίο εισάγονται τα κλειδιά του σωρού από αριστερά προς τα δεξιά και από πάνω προς τα κάτω.

Υπάρχουν δύο είδη σωρών:

•Οι σωροί μέγιστων (maxheap).

•Και οι σωροί ελαχίστων (minheap).

Οι βασικές λειτουργίες ενός σωρού είναι:

•Η εισαγωγή (Insert) ενός στοιχείου στον σωρό.

•Η διαγραφή (Delete) ενός στοιχείου από τον σωρό.

6.4.1 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
// array of integers to hold values
    private int[] a = new int[100];

    // number of elements in array
    private int x;

    // Heap Sort Algorithm
    public void sortArray()
    {
    int i;
    int temp;

    for( i = (x/2)-1; i >= 0; i- )
    {
    siftDown( i, x );
    }

    for( i = x-1; i >= 1; i- )
    {
    temp = a[0];
    a[0] = a[i];
```

```
a[i] = temp;
siftDown( 0, i-1 );
}
}

public void siftDown( int root, int bottom )
{
bool done = false;
int maxChild;
int temp;

while( (root*2 <= bottom) && (!done) )
{
if( root*2 == bottom )
maxChild = root * 2;
else if( a[root * 2] >a[root * 2 + 1] )
maxChild = root * 2;
else
maxChild = root * 2 + 1;

if( a[root] <a[maxChild] )
{
temp = a[root];
```

```
a[root] = a[maxChild];
a[maxChild] = temp;
root = maxChild;
}
else
{
done = true;
}
}
}
```

Το είδος της σωρού δεν απαιτεί μαζική αναδρομή ή πολλαπλές συστοιχίες στην εργασία. Αυτό το καθιστά μια ελκυστική επιλογή για τα πολύ μεγάλα σύνολα δεδομένων με εκατομμύρια αντικείμενα. Η λειτουργία του αλγορίθμου αρχίζει με την οικοδόμηση ενός σωρού από ένα σύνολο δεδομένων και με την έπειτα αφαίρεση του μεγαλύτερου στοιχείου και τη διάθεσή του στο τέλος του ταξινομημένου πίνακα. Αφού αφαιρέθηκε το μεγαλύτερο στοιχείο από τη σωρό, αυτό απομακρύνεται και τοποθετείτε στην επόμενη ανοιχτή θέση ξεκινώντας από το τέλος του ταξινομημένου πίνακα. Αυτό επαναλαμβάνεται μέχρις ότου να μην υπάρχουν αντικείμενα που έχουν μείνει στο σωρό και ο πίνακας να είναι πλήρης.

Σε μια τέτοια εφαρμογή απαιτούνται δύο πίνακες, ένας για να τα στοιχεία

της σωρού και ένας για τα ταξινομημένα στοιχεία. Κάθε φορά που ένα στοιχείο έχει αφαιρεθεί από τη σωρό, απελευθερώνετε ένα διάστημα στο τέλος του πίνακα όπου το παραληφθές στοιχείο μπορεί να τοποθετηθεί μέσα σε αυτόν.

6.5 MERGE SORT (ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΣΥΓΧΩΝΕΥΣΗ)

[2] *Η τέχνη του μηχανικού χαρακτηρίζεται από το γεγονός ότι οι μηχανικοί δεν ικανοποιούνται με την επίτευξη μίας οποιασδήποτε λύσης. Οι μηχανικοί αναζητούν την καλύτερη λύση με τη χρήση καθορισμένων όρων, υπό γνωστούς περιορισμούς, και κάνοντας τους συμβιβασμούς που επιβάλλονται από το γεγονός ότι δουλεύουν στον πραγματικό κόσμο.*

E. Yourdon & L. Constantine, *Structure Design*

Ο αλγόριθμος συγχώνευσης (merge sort) έχει ως **πλεονέκτημα** την ευκολία να παρουσιάζει τη συνένωση των ήδη ταξινομημένων λιστών σε μια νέα ταξινομημένη λίστα. Αρχίζει συγκρίνοντας κάθε ζευγάρι στοιχείων της λίστας (δηλαδή το πρώτο με το δεύτερο, το τρίτο με το τέταρτο και ούτω καθεξής) και εναλλάσσει τα στοιχεία αν το πρώτο είναι μεγαλύτερο του δεύτερου. Στη συνέχεια συνενώνει τα ζευγάρια των ταξινομημένων λιστών των δύο στοιχείων σε ταξινομημένες λίστες των τεσσάρων στοιχείων, στη συνέχεια των οχτώ

και ούτω καθεξής, μέχρι να συνενωθούν δύο λίστες στην τελική ταξινομημένη λίστα. Ο εν λόγω αλγόριθμος έχει θεωρητική πολυπλοκότητα O(n log n).

Γενικά, έστω ότι θέλουμε να εισάγουμε m στοιχεία σε μια ταξινομημένη λίστα μεγέθους n , αντί να αντιμετωπίσουμε καθένα από τα m στοιχεία ανεξάρτητα , πρώτα τα ταξινομούμε και μετά τα εισάγουμε με τη σειρά, γεγονός που μας επιτρέπει να εκμεταλλευτούμε την εξοικονόμηση για μελλοντικές εισαγωγές. Διαισθητικά αυτό θα πρέπει να δουλεύει βέλτιστα όταν m=n. Έτσι, ας διαιρέσουμε το πρόβλημα σε δύο μισά, ας ταξινομήσουμε τα μισά, και μετά ας συγχωνεύσουμε τα ταξινομημένα μισά.

Το χειρότερο κόστος αυτού του αλγορίθμου είναι:

$$f(n) = \begin{cases} 0 & n \leq 1 \\ f([n/2]) + f([n/2]) + n - 1 & n > 1 \end{cases}$$

6.5.1 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
mergesort(int a[], int low, int high)
    {
    int mid;
    if(low <high)
    {
    mid=(low+high)/2;
    mergesort(a,low,mid);
    mergesort(a,mid+1,high);
    merge(a,low,high,mid);
    }
    return 0;
    }


    merge(int a[], int low, int high, int mid)
    {
    int i, j, k, c[50];
    i=low;
    j=mid+1;
    k=low;
    while((i<=mid) & & (j<=high))
```

```
{
if(a[i]<a[j])
{
c[k]=a[i];
k++;
i++;
}
else
{
c[k]=a[j];
k++;
j++;
}
}
while(i<=mid)
{
c[k]=a[i];
k++;
i++;
}
while(j<=high)
{
c[k]=a[j];
```

```
k++;
j++;
}
for(i=low;i<k;i++)
{
a[i]=c[i];
}
}
```

6.5.2 ΛΥΣΗ ΤΟΥ ΚΩΔΙΚΑ

Ας υποθέσουμε ότι έχουμε τους αριθμούς 3 5 2 6 8

Άρα με βάση το παραπάνω κώδικα έχουμε το εξής αποτέλεσμα:

a[] = 3, 5, 2, 6, 8

low = 0; high = 4;

mergesort(a, 0, 4)

mid = 0 + 4 / 2 = 2

mergesort(a, 0, 2)

mergesort(a, 3, 4)

merge(a, 0, 4, 2)

6.6 QUICK SORT (ΤΑΞΙΝΟΜΗΣΗ ΜΕ ΔΙΑΙΡΕΣΗ Ή ΓΡΗΓΟΡΗ ΤΑΞΙΝΟΜΗΣΗ)

Ιστορικά, ο αλγόριθμος Quicksort αναπτύχθηκε το 1960 από τον Tony Hoare που βρέθηκε ως επισκέπτης φοιτητής στο Κρατικό Πανεπιστήμιο της Μόσχας. Εκείνη την εποχή, εργάστηκε για την αυτόματη μετάφραση ενός σχεδίου για το Εθνικό Εργαστήριο Φυσικής. Έτσι ανέπτυξε τον αλγόριθμο αυτό για να ταξινομήσουν τις λέξεις που έιναι για μετάφραση, για μεγαλύτερη ευκολία.

Ο αλγόριθμος quick sort είναι ένα παράδειγμα αλγορίθμου "διαίρει και βασίλευε"που στηρίζεται σε μια λειτουργία διαχωρισμού (partition). Για το διαχωρισμό ενός πίνακα, επιλέγουμε ένα στοιχείο, επονομαζόμενο ως βήμα (pivot), και μετακινούμε όλα τα μικρότερα στοιχεία πριν το στοιχείο βήμα και τα μεγαλύτερα μετά από αυτό. Η λειτουργία αυτή μπορεί να γίνει αποτελεσματικά σε γραμμικό χρόνο. Στη συνέχεια επαναληπτικά ταξινομούμε τις μικρότερες και μεγαλύτερες υπολίστες. Ο εν λόγω αλγόριθμος είναι ένας από τους ταχύτερους αλγόριθμους ταξινόμησης και μάλιστα απαιτεί O(log n) μνήμη. Το πιο περίπλοκο θέμα σε αυτόν τον αλγόριθμο είναι η επιλογή του στοιχείου βήματος. Λανθασμένες επιλογές αυτού του στοιχείου μπορούν να κοστίσουν σε δραματική μείωση της επίδοσης σε $O(n^2)$. Αν, όμως, σε κάθε βήμα διαλέγουμε το μεσαίο στοιχείο της λίστας τότε ο αλγόριθμος εκτελείται σε O(n log n).

Το **μειονέκτημα** της έκδοσης αυτής, είναι ότι απαιτεί πάνω από O (n)

επιπλέον αποθηκευτικό χώρο, όπως και στη Merge Sort. Επίσης, τα πρόσθετα κονδύλια μνήμης που απαιτούνται μπορεί να είναι δραστικά με ταχύτητα πρόσκρουσης στις επιδόσεις της λανθάνουσας μνήμης. Έτσι υπάρχει πλέον μια πιο σύνθετη έκδοση που χρησιμοποιεί έναν αλγόριθμο κατάτμησης και ο οποίος μπορεί να επιτύχει την πλήρη ταξινόμηση, χρησιμοποιώντας O (log n) μνήμη.

Για να βρούμε το μέσο κόστος του αλγορίθμου, θα υποθέσουμε ότι υπάρχουν n στοιχεία στ ο τμήμα της λίστας. Η παρακάτω αναδρομική σχέση εκφράζει το μέσο αριθμό συγκρίσεων του Quicksort.

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (f(i) + f(n-i-1)) & n > 1 \end{cases}$$

6.6.1 Η ΤΕΧΝΙΚΗ "διαίρει και βασίλευε".

Στην επιστήμη των υπολογιστών ,η τεχνική "διαίρει και βασίλευε"(divide and conquer, D & C) είναι ένα σημαντικό σχέδιο για έναν αλγόριθμο, γιατί βασίζεται σε πολλαπλά τμήματα αναδρομής. Ένας αλγόριθμος "διαίρει και βασίλευε"λειτουργεί με την αναδρομική διάσπαση ενός προβλήματος σε δύο ή

περισσότερα προβλήματα του ίδιου τύπου, μέχρις ότου γίνει αρκετά απλό και κατανοητό και να μπορεί να λυθεί άμεσα. Στη συνέχεια οι λύσεις που βρίσκουμε στα επιμέρους προβλήματα, συνδυάζονται για να δώσουν τη λύση στο αρχικό πρόβλημα.

Αυτή η τεχνική είναι η βάση στους αποδοτικούς αλγόριθμους όπως στον quick sort και στον merge sort.

Η ορθότητα του αλγόριθμου "διαίρει και βασίλευε" αποδεικνύεται συνήθως με μαθηματική επαγωγή και το κόστος του καθορίζεται συχνά από την εύκολη ή δύσκολη επίλυση των αναδρομικών σχέσεων.

6.6.2 ΒΗΜΑΤΑ ΤΑΞΙΝΟΜΗΣΗΣ

Όπως προανέφερα ο Quick Sort είναι αλγόριθμος "διαίρει και βασίλευε". Διαιρεί πρώτα μία μεγάλη λίστα σε δύο μικρότερους- επιμέρους πίνακες: ο ένας με τα μικρά στοιχεία και ο άλλος με τα μεγαλύτερα στοιχεία. Και στη συνέχεια κατ' επανάληψη ταξινομεί τους υποκαταλόγους. Τα βήματα για την ταξινόμηση των στοιχείων είναι τα εξής:

1) Επιλέγουμε ένα στοιχείο από τη λίστα, και το ονομάζουμε άξονα.

2) Κάνουμε αναδιάταξη του πίνακα, έτσι ώστε τα στοιχεία που είναι μικρότερα από τον άξονα έρχονται πριν από αυτόν και τα στοιχεία που είναι μεγαλύτερα από τον άξονα πάνε μετά από αυτόν. Όταν ολοκληρωθεί η κατάτμηση των στοιχείων , ο άξονας βρίσκεται στην τελική του θέση. Αυτό ονομάζεται λειτουργία διαχωρισμού.

3) Τέλος, αναδρομικά ταξινομούμε τον υποκατάλογο με τα μικρότερα στοιχεία και τον υποκατάλογο με τα μεγαλύτερα στοιχεία.

Η βασική περίπτωση της αναδρομής, είναι κατάλογοι με μέγεθος μηδέν ή ένα, οι οποίοι δεν χρειάζεται να διευθετηθούν.

Κάθε αναδρομική κλήση στη Quicksort λειτουργία, μειώνει το μέγεθος του πίνακα που είναι ταξινομημένος τουλάχιστον κατά ένα στοιχείο, αφού σε κάθε επίκληση τοποθετείται αμέσως στη θέση του τελικού. Ως εκ τούτου, αυτός ο αλγόριθμος είναι εγγυημένο ότι για να τερματίσει θα πρέπει να είναι μετά από πολλές αναδρομικές κλήσεις. Ωστόσο, δεδομένου ότι επαναπροσδιορίζει την κατάτμηση των στοιχείων, αυτή η έκδοση της "γρήγορης" ταξινόμησης δεν είναι ένα σταθερό είδος.

6.6.3 ΠΑΡΑΛΛΑΓΕΣ ΤΗΣ QUICK SORT

Υπάρχουν τρεις γνωστές παραλλαγές του αλγόριθμου Quicksort:

•**Ισορροπημένη (Balance) Quicksort:** Επιλέγουμε έναν άξονα, ο οποίος να αντιπροσωπεύει τα στοιχεία που πρέπει να ταξινομηθούν , και στη συνέχεια να ακολουθήσουν το κανονικό αλγόριθμο Quicksort.

•**Εξωτερική (External) Quicksort:** Είναι ίδια με την τακτική της γρήγορης ταξινόμησης, μόνο που ο άξονας περιστροφής του, αντικαθίσταται από μια ζώνη. Πρώτον, διαβάζουμε το πρώτο και το τελευταίο στοιχείο και τα ταξινομούμε σε μια προσωρινή μνήμη. Μετά διαβάζουμε το επόμενο στοιχείο από την αρχή ή το τέλος της ισορροπημνεης λίστας. Εάν το επόμενο στοιχείο είναι μικρότερο από αυτό που βρίσκεται στην προσωρινή μνήμη ως μικρό, τότε το γράφουμε στην αρχή (εάν το στοιχείο είναι μεγαλύτερο από το μέγιστο, τότε το γράφω στο τέλος). Διαφορετικά, γράφουμε το μεγαλύτερο ή το μικρότερο στη μνήμη για να θέσει το επόμενο στοιχείο στη προσωρινή μνήμη. Τέλος, ταξινομήσαμε αναδρομικά το μικρότερο διαμέρισμα και γκρουπάραμε τα υπόλοιπα διαμερίσματα στο βρόγχο.

•**Quick Sort-Radix sort:** Είναι ένας συνδυασμός radix sort και Quick-sort. Επιλέγουμε ένα στοιχείο (άξονα) από τον πίνακα και το λαμβάνουμε υπόψη μας ως τον πρώτο χαρακτήρα (κλειδί) της ακολουθίας. Ο διαχωρισμός των υπόλοιπων στοιχείων γίνεται σε τρία μέρη: 1) Σε εκείνα τα οποία είναι μικρότερα από τον χαρακτήρα "κλειδί", 2) σε αυτά που είναι ίσα με τον πρώτο χαρακτήρα και 3) σε αυτά που είναι μεγαλύτερα από το χαρακτήρα. Έπειτα ταξινομούμε τα "μικρότερα"και τα "μεγαλύτερα"στοιχεία πριν από το χαρακτήρα, και τα στοιχεία που είναι "ίσα"τα τα ξινομούμε μετά τον χαρακτήρα (κλειδί).

6.6.4 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
void quickSort( int[], int, int);
    int partition( int[], int, int);


    void main()
    {
    int a[ ] =  7, 12, 1, -2, 0, 15, 4, 11, 9;
    int i;
    printf(" Unsorted array is: ");
    for(i = 0; i <9; ++i)
    printf(" %d ", a[i]);
```

```
quickSort( a, 0, 8);
printf("Sorted array is: ");
for(i = 0; i <9; ++i)
printf(" %d ", a[i]);
}

void quickSort( int a[], int l, int r)
}

int j;
if( l <r )
{
// divide and conquer
j = partition( a, l, r);
quickSort( a, l, j-1);
quickSort( a, j+1, r);
}
}
int partition( int a[], int l, int r)
{
int pivot, i, j, t;
pivot = a[l];
i = l;
```

```
j = r+1;

while( 1)
{
do ++i;
while( a[i] <= pivot && i <= r );
do –j;
while( a[j] >pivot );
if( i >= j )
break;
t = a[i];
a[i] = a[j];
a[j] = t;
}
t = a[l];
a[l] = a[j];
a[j] = t;
return j:
}
```

6.7 RADIX SORT (ΜΗ ΣΥΓΚΡΙΤΙΚΟΣ ΑΛΓΟΡΙΘΜΟΣ ΤΑΞΙΝΟΜΗΣΗΣ

[1] Ένας άλλος γνωστός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος "radix sort". Ο συγκεκριμένος αλγόριθμος ταξινομεί μία λίστα αριθμών μεταχειριζόμενος τους αριθμούς σαν δυαδικούς. Υπάρχουν δύο παραλλαγές αυτού του τύπου του αλγορίθμου:

1) Η πρώτη παραλλαγή ταξινομεί τους αριθμούς αρχίζοντας από το πιο σημαντικό ψηφίο.

2) Και η δεύτερη παραλλαγή από το λιγότερο σημαντικό ψηφίο.

Έτσι για παράδειγμα, η πρώτη παραλλαγή του αλγορίθμου ταξινομεί τους αριθμούς σύμφωνα με το πρώτο ψηφίο τους χρησιμοποιώντας κάποιον άλλον απλό αλγόριθμο ταξινόμησης, συνεχίζει με το επόμενο ψηφίο και στο τέλος η λίστα είναι ταξινομημένη.

Ο αλγόριθμος radix sort έχει θεωρητική πολυπλοκότητα O(n*k), όπου n είναι ο αριθμός των στοιχείων της λίστας και k το μέγεθος σε bit της αναπαράστασης των αριθμών.

6.7.1 ΕΝΑ ΤΥΠΙΚΟ ΠΑΡΑΔΕΙΓΜΑ

Για να εκτιμήσουμε τη Radix Sort, ας εξετάσουμε την εξής αναλογία: Ας υποθέσουμε ότι θέλουμε να ταξινομήσουμε μια τράπουλα 52 χαρτιών (τα διάφορα συμβολικά στοιχεία μπορούν να δοθούν ως κατάλληλες αξίες, όπως για παράδειγμα 1 για καρό, 2 για σπαθί, 3 για την κούπα και 4 για μπαστούνι). Φυσικό θα ήταν να ταξινομήσουμε πρώτα τις κάρτες σύμφωνα με το σύμβολο, τότε το κάθε είδος θα χωριζόταν σε μία από τις τέσσερις ταξινομήσεις και τέλος, θα συνδυάζαμε και τις τέσσερις με τη σειρά. Η προσέγγιση αυτή, ωστόσο, έχει εγγενή μειονεκτήματα. Όταν κάθε μία από τις σωρούς είναι η διαλογή, τότε οι άλλες σωροί πρέπει να διατηρούνται κατά μέρος και να παρακολουθούνται. Εάν, αντιθέτως, προσεγγίσουμε τη πρώτη διαλογή των καρτών από πλευράς αξίας, το πρόβλημα αυτό εξαλείφεται. Μετά το πρώτο βήμα, έχουμε τέσσερις χωριστές στοίβες που συνδυάζονται , και στη συνέχεια τις ταξινομούμε κατά σύμβολο. Εάν ένας σταθερός αλγόριθμος ταξινόμησης (δηλαδή ο αλγόριθμος που επιλύει ένα πρόβλημα με τη διατήρηση του αριθμού που λαμβάνει για πρώτη φορά στην είσοδο, με το πρώτο αριθμό στην έξοδο), μπορεί εύκολα να διαπιστώσει ότι λαμβάνονται με ακρίβεια τα τελικά αποτελέσματα.

6.7.2 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
void radixsort(int a[],int);

    void main()
```

```
{
int n,a[20],i;
clrscr();

printf("enter the number :");
scanf("%d",& n);
printf("ENTER THE DATA");
for(i=0;i<n;i++)
{
printf("%d",i+1);
scanf("%d",&a[i]);
}
radixsort(a,n);
getch();
}

void radixsort(int a[],int n)
{
int rear[10],front[10],first,p,q,exp,k,i,y,j;
struct {
int info;
int next; } node[NUMELTS];
```

```
for(i=0;i<n-1;i++)
{
node[i].info=a[i];
node[i].next=i+1;
}

node[n-1].info=a[n-1];
node[n-1].next=-1;
first=0;
for(k=1;k<=2;k++) //consider only 2 digit number
{
for(i=0;i<10;i++)
front[i]=-1;
rear[i]=-1;
}
while(first!=-1)
{
p=first;
first=node[first].next;
y=node[p].info;
exp=pow(10,k-1);
j=(y/exp)%10;
q=rear[j];
```

```
if(q==-1)
front[j]=p;
else
node[q].next=p;
rear[j]=p;
}

for(j=0;j<10&&front[j]==-1;j++)
first=front[j];
while(j!=9)
{
for(i=j+1;i<10&&front[i]==-1;i++)
if(i<=9)
{
p=i;
node[rear[j]].next=front[i];
}
j=i;
}
node[rear[p]].next=-1;
}
//copy into original array
```

```
for(i=0;i<n;i++)
{
a[i]=node[first].info;
first=node[first].next;
}

clrscr();
textcolor(YELLOW);
printf("DATA AFTER SORTING:");
for(i=0;i<n;i++)
printf("%d . %d",i+1,a[i]);
}
```

6.8 SHAKER SORT

[1] Ένας άλλος αλγόριθμος ταξινόμησης είναι ο αλγόριθμος του "shaker sort". Ο αλγόριθμος αυτός είναι μια παραλλαγή του αλγόριθμου φυσαλίδας. Η διαφορά του με τον αλγόριθμο της φυσαλίδας είναι ότι αντί να διασχίζει τη λίστα από την αρχή προς το τέλος, τη διασχίζει εναλλάξ από την αρχή προς το τέλος και μετά από το τέλος προς την αρχή και ούτω καθεξής. Με τη διαφοροποίηση αυτή ο αλγόριθμος αυτός πετυχαίνει ελαφρά βελτιωμένη απόδοση σε σχέση με τον κλασικό αλγόριθμο φυσαλίδας, γιατί ο αλγόριθμος φυσαλίδας διασχίζει τη λίστα προς μία κατεύθυνση και μπορεί να μετακινεί στοιχεία της λίστας ένα βήμα προς τα πίσω σε κάθε επανάληψη. Η θεωρητική πολυπλοκότητα του αλγορίθ-

μου αυτού είναι στη χειρότερη και στη μέση περίπτωση $O(n^2)$, ενώ πλησιάζει το O(n) στην καλύτερη περίπτωση, που συμβαίνει όταν η αρχή της λίστας είναι ταξινομημένη.

6.8.1 ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ

```
void shaker(char *items, int count)
    {
    register int a;
    int exchange;
    char t;

    do
    exchange = 0;
    for(a=count-1; a >0; –a) {
    if(items[a-1] >items[a]) {
    t = items[a-1];
    items[a-1] = items[a];
    items[a] = t;
    exchange = 1;
    }
    }
```

```
for(a=1; a <count; ++a) {
if(items[a-1] >items[a]) {
t = items[a-1];
items[a-1] = items[a];
items[a] = t;
exchange = 1;
}
}

}
while(exchange); /* sort until no exchanges take place */
}
int main(void)
{
char s[255];
printf("Enter a string:");
gets(s);
shaker(s, strlen(s));
printf("The sorted string is: %s.", s);
return 0;
}
```

7 ΣΥΓΚΡΙΣΗ ΑΛΓΟΡΙΘΜΩΝ ΤΑΞΙΝΟΜΗΣΗΣ

7.1 INSERTION SORT- SELECTION SORT

Η **insertion sort** μοιάζει πολύ με την **selection sort.** Όπως και στη selection sort μετά από k περάσματα μέσα από τον πίνακα, αρχικά τα k στοιχεία είναι ταξινομημένα σε σειρά. Για τη selection sort τα k στοιχεία είναι τα μικρότερα, ενώ στην insertion sort ανεξάρτητα από τι είναι τα πρώτα στοιχεία k αυτά βρίσκονται μη ταξινομημένα στο πίνακα. Το βασικό πλεονέκτημα της Insertion sort είναι ότι ανιχνεύει πολλά στοιχεία μαζί όπως απαιτείται για να καθοριστεί η σωστή θέση του στοιχείου με k+1 ,ενώ η selection sort πρέπει να σαρώσει όλα τα υπόλοιπα στοιχεία για να βρει με απόλυτη βεβαιώτητα ποιο είναι το μικρότερο στοιχείο.

Οι υπολογισμοί δείχνουν ότι η insertion sort συνήθως εκτελεί περίπου τη μισή διαδικασία λόγω των πολλών συγκρίσεις σε σχέση με τη selection sort. Υποθέτοντας ότι k +1 η κατάταξη των στοιχείων είναι τυχαία, στην insertion sort κατά μέσο όρο θα απαιτήσει τη μετακίνηση του μισού περασμένου στοιχείου k ενώ στηselection sort πάντα απαιτείται η σάρωση όλων των ατοποθέτητων στοιχείων. Εάν η συστοιχία εισόδου δεν είναι ταξινομημένη, η insertion sort εκτελεί περισσότερες συγκρίσεις από τη selection sort. Εάν όμως η συστοιχία εισόδου είναι ήδη ταξινομημένη, τότε η insertion sort εκτελεί τόσο λίγες συγ-

κρίσεις όσο n-1,καθιστώντας έτσι την insertion sort πιο αποτελεσματική όταν χορηγείται για την ταξινόμηση ή "σχεδόν-ταξινόμηση" πινακών.

Ενώ στην ταξινόμηση με εισαγωγή γίνονται συνήθως λιγότερες συγκρίσεις σε σχέση με το είδος επιλογής αυτό απαιτεί περισσότερο γράψιμο επειδή ο εσωτερικός βρόχος μπορεί να απαιτήσει την μετακίνηση μεγάλων τμημάτων του ταξινομημένου τμήματος του πίνακα. Σε γενικές γραμμές, στην ταξινόμηση με εισαγωγή θα γράψει τη διάταξη O (n^2) φορές, ενώ στην ταξινόμηση με επιλογή θα γράψει μόνο O (n) φορές. Γι αυτό το λόγο η ταξινόμηση με επιλογή μπορεί να είναι προτιμότερη σε περιπτώσεις όπου το γράψιμο στη μνήμη είναι σημαντικά ακριβότερο από την ανάγνωση, όπως παραδείγματος χάρη με μνήμη EEPROM ή μνήμη flash.

7.2 INSERTION SORT- QUICK SORT-MERGE SORT

Μερικοί αλγόριθμοι "διαίρει και βασίλευε" όπως είναι **quick sort** και η **merge sort** κάνουν τη διαίρεση του καταλόγου σε μικρότερους υποκαταλόγους που στη συνέχεια θα ταξινομηθούν. Μια χρήσιμη βελτιστοποίηση στην πράξη για αυτούς τους αλγόριθμους είναι να χρησιμοποιήσουμε τον insertion sort για τη κατάταξη μικρών υποκαταλόγων, γιατί η insertion sort πλεονεκτεί σε σχέση με τους πιο σύνθετους αλγόριθμους. Το μέγεθος της λίστας για την insertion sort η οποία έχει το πλεονέκτημα να ποικίλλει ανάλογα με το περιβάλλον και

την εφαρμογή, αλλά κατά κανόνα γίνεται μεταξύ οκτώ και είκοσι στοιχείων.

Επίσης η insertion sort είναι ένας στοιχειώδες αλγόριθμος ταξινόμησης. Έχει χρονική πολυπλοκότητα $\Theta(n^2)$, με αποτέλεσμα να είναι πιο αργός από τον αλγόριθμο heapsort και τον αλγόριθμο merge sort.Επιπλέον η insertion sort είναι κατάλληλη για την ταξινόμηση μικρών συνόλων δεδομένων ή για την εισαγωγή νέων στοιχείων σε ταξινομημένη λίστα.

7.3 ΠΙΝΑΚΕΣ ΣΥΓΚΡΙΣΗΣ

Στον παρακάτω πίνακα παρουσιάζονται η μέση πολυπλοκότητα (Average),η πολυπλοκότητα χειρότερης (Worst) και καλύτερης (Best) περίπτωσης, υπό την προϋπόθεση ότι το μήκος κάθε κλειδιού είναι σταθερό, και ότι όλες οι συγκρίσεις, ανταλλαγές, και άλλες αναγκαίες διαδικασίες μπορούν να προχωρήσουν σε σταθερό χρόνο. Η Memory δείχνει το ποσό της βοηθητικής αποθήκευσης που απαιτείται πέρα από αυτό που χρησιμοποιείται από τον ίδιο τον κατάλογο, στις ίδιες περιπτώσεις.

Name	*Best*	*Average*	*Worst*	*Memory*	*Stable*	*Method*
Bubblesort	$O(n)$	–	$O(n^2)$	$O(1)$	*Yes*	*Exchanging*
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	*No*	*Selection*
Insertionsort	$O(n)$	$O(n+d)$	$O(n^2)$	$O(1)$	*Yes*	*Insertion*
Mergesort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	*Yes*	*Merging*
heapsort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	*No*	*Selection*
Quicksort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	*No*	*Partitioning*

Στον επόμενο πίνακα θα δούμε τους χρόνους εκτέλεσης για κάθε αλγόριθμο. Στην αριστερή στήλη έχουμε τον αριθμό των στοιχείων που εκτελούνται. Στον Quick Sort1 αναφερόμαστε χωρίς αναδρομή και στον Quick Sort2 με αναδρομή.

	BubbleSort	*SelectionSort*	*InsertionSort*	*QuickSort*1	*QuickSort*2
50	2.085*sec*	1.492*sec*	0.882*sec*	0.664*sec*	0.492*sec*
100	7.851*sec*	5.664*sec*	2.906*sec*	1.429*sec*	1.320*sec*
500	3.13*min*	2.11*min*	1.01*min*	8.570*sec*	7.796*sec*
1000	12.43*min*	8.40*min*	4.04*min*	18.507*sec*	17.632*sec*
5000	5.29, 57*hour*	3.34, 38*hour*	1.44, 17*hour*	1.54*min*	1.42*min*
10000	22.10, 11*hour*	14.40, 21*hour*	7.07, 49*hour*	3.57*min*	3.52*min*

Διαπιστώνουμε ξεκάθαρα ότι ο αλγόριθμος Quick Sort με αναδρομή είναι ο ταχύτερος, ενώ ο Bubble Sort είναι ο πιο χρονοβόρος.

References

[1] Αναγνώστου Χρ. *Ανάλυση Επεξεργασία και Παρουσίαση των Αλγορίθμων Ταξινόμησης Heap Sort και WeakHeapsort.* Θεσσαλονίκη (2009).

[2] Gregory J.E. Rawlins. *Αλγόριθμοι Ανάλυση και Σύγκριση.* Οικονομικό Πανεπιστήμιο Αθηνών (2004).

[3] Ιωαννίδης Ν. & Μαρινάκης Κ. & Μπακογιάννης Σ. *Αλγοριθμική Δομημένες Τεχνικές.* Αθήνα (1990).

[4] Κοΐλιας Χρ. *Δομές Δεδομένων και Οργανώσεις Αρχείων.* Αθήνα (1993).

[5] Τομάρας Αλ. *C Θεωρία και Πράξη.* Αθήνα (1994).

[6] Thomas H. Cormen & Charles E. Leiserson & Ronald L. Rivest & Clifoord Stein. *Εισαγωγή στους αλγορίθμους.* Ηράκλειο (2006)

[7] Design & Analysis of Algorithms. http://www.personal.kent.edu/ rmuhamma/Algorithm

[8] Learn computer science programming. http://www.learncs.net/sorting-algorhithms.html

[9] SoftPanorama. http://www.softpanorama.org/Algorithms/sorting.shtml

[10] Sorting Algorithm Animations.http://www.sorting-algorithms.com

[11] The heroic Tales Of Sorting Algorithms. http://tanksoftware.com/tutes/uni/sorting.html