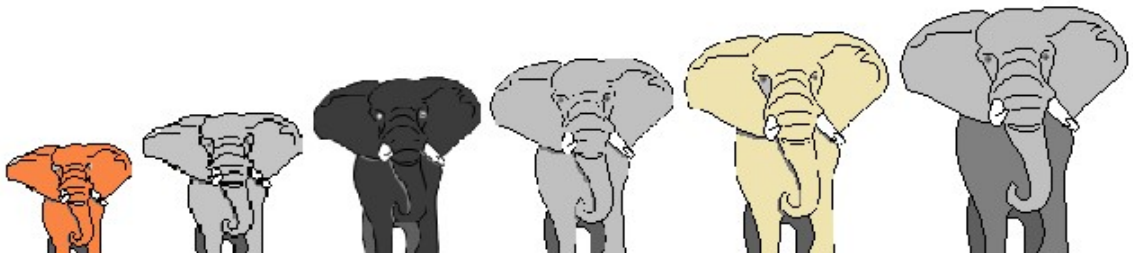
ΔΙΕΘΝΕΣ ΠΑΝΕΠΙΣΤΗΜΙΟ ΤΗΣ ΕΛΛΑΔΟΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ & ΗΛΕΚΤΡΟΝΙΚΩΝ
ΥΣΤΗΜΑΤΩΝ

Αντικειμενοστρεφής Προγραμματισμός

(https://people.iee.ihu.gr/~adamidis/OOP.html)

Τεχνικές και Αλγόριθμοι Ταξινόμησης



Παναγιώτης Αδαμίδης
adamidis@ihu.gr

Θεσσαλονίκη, Φεβρουάριος 2020

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή

Σε αυτή την ενότητα θα εξετάσουμε διάφορες τεχνικές ταξινόμησης. Μερικοί από τους αλγόριθμους είναι μικροί, ενώ μερικοί βελτιώνουν σημαντικά τον χρόνο τους μέσω τεχνασμάτων στην κωδικοποίησή τους και μας επιτρέπουν να διερευνήσουμε τα σημεία κοινών και καθιερωμένων θεμάτων/τεχνασμάτων τα οποία κάνουν τις καλές ή τις κακές τεχνικές. Οι αλγόριθμοι ταξινόμησης παρουσιάζουν μία ευκαιρία εισαγωγής σε έννοιες της θεωρίας της επιστήμης υπολογιστών.

Χωρίζουμε την μελέτη των τεχνικών ταξινόμησης σε τρείς κατηγορίες:

1. ***Τεχνικές ανταλλαγής (Exchange techniques)***, όπως οι μέθοδοι bubble sort και insertion sort, οι οποίες έχουν χρονική πολυπλοκότητα $O(n^2)$
2. ***Τεχνικές βασισμένες σε δομές δένδρου (Tree-based techniques)***, όπως η δυαδική δενδρική ταξινόμηση (binary tree sorting) οι οποίες έχουν χρονική πολυπλοκότητα $O(nlog_2n)$
3. ***Αναδρομικές Τεχνικές (Recursive techniques)***, οι οποίες έχουν επίσης πολυπλοκότητα $O(nlog_2n)$, αλλά με δυνατότητα καλύτερης (πιο συμπαγούς) κωδικοποίησης και (δυναμικά) καλύτερης απόδοσης από άλλες τεχνικές.

Σε κάθε μία κατηγορία παρουσιάζεται τουλάχιστον μία αντιπροσωπευτική τεχνική, εκτός της περίπτωσης των δενδροειδών δομών, και προσπαθούμε να κατανοήσουμε την πηγή της χρονικής πολυπλοκότητας και της πολυπλοκότητας χώρου των συγκεκριμένων αλγορίθμων.

2. Υπολογιστική Πολυπλοκότητα

Ο χρόνος εκτέλεσης του αλγορίθμου της σειριακής αναζήτησης (sequential search), όπως είδαμε, είναι χονδρικά ανάλογος με το μέγεθος του προβλήματος. Εάν από ένα πρόβλημα μεγέθους n πάμε σε ένα πρόβλημα μεγέθους $n+1$, αυτό στην χειρότερη περίπτωση θα προσθέσει ένα σταθερό αριθμό βημάτων στον χρόνο που απαιτείται. Εάν πάμε σε ένα πρόβλημα μεγέθους $2n$ αυτό γενικά θα διπλασιάσει το χρόνο που απαιτείται για την εκτέλεση του αλγόριθμου.

Η δυαδική αναζήτηση (binary search) λειτουργεί διαφορετικά. Ο διπλασιασμός του μεγέθους του προβλήματος προσθέτει μόνο ένα σταθερό αριθμό βημάτων και συνεπώς ένα σταθερό χρονικό διάστημα στο συνολικό χρόνο που απαιτείται για την εκτέλεση του αλγόριθμου. Για παράδειγμα θεωρείστε το εξής πρόβλημα. Μαντέψτε ένα αριθμό σε μία συγκεκριμένη περιοχή πχ από 1-200. Εάν η πρώτη σας επιλογή είναι 100, αμέσως ελαττώνετε το πρόβλημα στο μισό δηλ. στην περιοχή 1-99 ή 101-200, ή λύνετε το πρόβλημα. Αλγόριθμοι με αυτή την συμπεριφορά ονομάζονται ***λογαριθμικοί*** *(logarithmic)*.

Τυπικά ως υπολογιστική *χρονική πολυπλοκότητα (time complexity)* ορίζεται η διάρκεια χρόνου ενός προγράμματος (ή αλγορίθμου, ή διαδικασίας) σαν συνάρτηση του πλήθους των εισόδων (δεδομένων) του. Ομοίως ως *πολυπλοκότητα χώρου (space complexity)* ενός αλγορίθμου, ορίζεται το μέγεθος της απαιτούμενης μνήμης σαν συνάρτηση του πλήθους των εισόδων του.

Αυτοί οι ορισμοί δεν είναι τόσο απλοί όσο φαίνονται. Για παράδειγμα πρέπει να διαχωρίσουμε μεταξύ της χειρότερης περίπτωσης του χρόνου ή του χώρου που απαιτείται, τον μέσο χρόνο ή χώρο και τον καλύτερο χρόνο ή χώρο. Επίσης πρέπει πάντα να έχουμε υπόψη μας το πως συμπεριφέρεται ένας αλγόριθμος με ένα συγκεκριμένο σύνολο εισόδων (δεδομένων). Συνήθως αναφερόμαστε στην μέση απόδοση ενός αλγορίθμου, αλλά όταν η καλύτερη, ή η χειρότερη, περίπτωση είναι πολύ διαφορετικές τότε θα πρέπει να αναφέρονται και αυτές.

Συχνά η επιλογή ενός αλγόριθμου είναι αποτέλεσμα συνδυασμών και συμβιβασμών. Ο ρόλος μας ως προγραμματιστές δεν είναι απαραίτητα να επιλέξουμε έναν αλγόριθμο ο οποίος λειτουργεί καλύτερα σε όλες τις πιθανές περιπτώσεις, αφού είναι εξαιρετικά απίθανο να υπάρχει ένας τέτοιος αλγόριθμος. Ο ρόλος και η δουλειά ενός προγραμματιστή είναι να έχει ευρεία γνώση πολλών αλγορίθμων καθώς και τεχνικών ανάλυσης βασικών αλγορίθμων, έτσι ώστε να μπορεί να κάνει λογικές επιλογές και να αποφύγει κάποιες καταστροφές. Όταν κατασκευάζονται τα προγράμματά μας χρησιμοποιούμε κάποια προγραμματιστικά εργαλεία (ή/και τεχνικές) τα οποία μας επιτρέπουν να δούμε **πού** πραγματικά σπαταλείται ο χρόνος. Αυτή η πειραματική μέθοδος μας επιτρέπει να βελτιώσουμε ένα τμήμα κώδικα.

Ήδη ορίσαμε την χρονική πολυπλοκότητα ενός αλγόριθμου ως την χρονική διάρκεια ολοκλήρωσης του αλγόριθμου συναρτήσει του πλήθους n των εισόδων του. Υποθέτουμε ότι έχουμε ένα αλγόριθμο με πολυπλοκότητα χρόνου $t(n)$ η οποία δίνεται από την σχέση:

$$t(n) = C\,f(n) + k$$

όπου C και k είναι σταθερές ανεξάρτητες του n. Οι αλγόριθμοι που εξετάσαμε ή θα εξετάσουμε έχουν συναρτήσεις χρονικής πολυπλοκότητας $f(n)$ όπως n^2 ή $n\log_2 n$ ή 2^n ή n^3+n^2-n κ.ο.κ. Οι δύο σταθερές C και k εξαρτώνται από τον συγκεκριμένο υπολογιστή, την γλώσσα, τον μεταγλωττιστή και τον χρόνο αρχικοποίησης του αλγόριθμου. Για μικρές τιμές του n, ο υπολογιστικός χρόνος είναι πιθανό να κυριαρχείται από εξωτερικούς παράγοντες οι οποίοι συμπεριλαμβάνονται στην σταθερά k. Έτσι το πιο ενδιαφέρον στοιχείο είναι το πως μεταβάλλεται ο υπολογιστικός χρόνος καθώς αυξάνεται το n.

Τυπικά αυτό ορίζεται με το σύμβολο "Ο" το οποίο περιγράφει την συμπεριφορά μιας συνάρτησης όπως η $t(n)$, καθώς το n αυξάνεται. Εάν η συνάρτηση $t(n)$ αυξάνεται με το τετράγωνο του n (n^2) καθώς το n αυξάνεται τότε λέμε ότι η *πολυπλοκότητα της* $t(n)$ είναι $O(n^2)$.

Ο συμβολισμός αυτός μας λέει ότι η χρονική πολυπλοκότητα συμπεριφέρεται με ένα συγκεκριμένο τρόπο, για ένα αριθμό εισόδων n. Μας δίνει ένα μηχανισμό περιγραφής της λειτουργίας του αλγόριθμου ασυμπτωτικά, καθώς και ένα μέτρο σύγκρισης διαφορετικών αλγορίθμων. Οι


αλγόριθμοι που αντιμετωπίζουμε έχουν συνήθως πολυπλοκότητα O*(n)* (πολυωνυμική-polynomial), O(n^2) (εκθετική-exponential) ή O($n\log_2 n$) (λογαριθμική-logarithmic).

Στο Σχ. 2.1 δίνεται ένας απλός αλγόριθμος ταξινόμησης (ταξινόμηση με ανταλλαγή – exchange sort) για να εξετάσουμε την πολυπλοκότητα χρόνου και χώρου πιο τυπικά. Για το σκοπό αυτό κάνουμε κάποιες παραδοχές προσπαθώντας να αποφύγουμε να αναφερθούμε σε συγκεκριμένο Η/Υ. Μην ξεχνάτε ότι η μεταγλώττιση δεν επηρεάζει την ασυμπτωτική συμπεριφορά του αλγόριθμου και περιέχεται στις σταθερές.

```
public class Ex_Sort {

    public static void main (String[] args) {
        int[] numbers = {3, 9, 6, 1, 2};

        Exchange_Sort.sort (numbers);

        for (int i = 0; i < numbers.length; i++)
            System.out.println (numbers[i]);

    } // method main
} // class Ex_Sort

class Exchange_Sort {

   public static void sort (int[] numbers) {
      int i, j, temp;
      for (i=0; i < numbers.length-1; i++)              {1}
          for (j=i+1; j < numbers.length; j++)          {2}
              if (numbers[i] > numbers[j]) {            {3}
                  temp=numbers[i];                      {4}
                  numbers[i]= numbers[j];               {5}
                  numbers[j]= temp;                     {6}
               }
    } // method sort
} //class Exchange_Sort
```

Σχήμα 2.1. Αλγόριθμος ταξινόμησης με ανταλλαγή

Ο αλγόριθμος ταξινόμησης με ανταλλαγή λειτουργεί συγκρίνοντας το πρώτο στοιχείο με όλα τα επόμενα. Όταν βρίσκει κάποιο το οποίο είναι μικρότερο (για ταξινόμηση σε αύξουσα σειρά) ανταλάσσει τα δύο στοιχεία. Έτσι το μικρότερο (μέχρι αυτή την στιγμή) στοιχείο είναι στην πρώτη θέση. Οι συγκρίσεις με το πρώτο στοιχείο και οι πιθανές απαραίτητες ανταλλαγές συνεχίζονται μέχρι το τέλος των στοιχείων (του πίνακα numbers στον παραπάνω κώδικα). Αυτό αποτελεί ένα πέρασμα του πίνακα. Επαναλαμβάνουμε την διαδικασία ξεκινώντας από το δεύτερο στοιχείο (αφού το πρώτο είναι το μικρότερο όλων και άρα στην σωστή θέση). Με το τέλος του δεύτερου περάσματος το δεύτερο μικρότερο στοιχείο είναι στην δεύτερη θέση. Τελικά μετά από *n*-1 περάσματα τα στοιχεία (ο πίνακας numbers) είναι πλήρως ταξινομημένα.


Ο αλγόριθμος αυτός μπορεί να βελτιωθεί και αργότερα θα τον δούμε λεπτομερώς. Εδώ θα εξετάσουμε την πολυπλοκότητα του αλγόριθμου.

Οι συγκρίσεις των τιμών των στοιχείων γίνονται στην εντολή 3. Εάν η σειρά των στοιχείων δεν είναι σωστή τότε εκτελούνται οι εντολές 4-6 (ανταλλαγή θέσεων των δύο στοιχείων), αλλιώς τελειώνουν και οι δύο βρόχοι for. Η εντολή 1 εκτελείται *n* φορές. Το πόσες φορές θα εκτελεστεί η δύο εξαρτάται από την τιμή του i. Όταν το i είναι 1 εκτελείται *n* φορές, ενώ στο τέλος όταν είναι *n*-1 εκτελείται 1 φορά (Πίνακας 2.1).

Πίνακας 2.1. Αριθμός φορών που εκτελείται η εντολή 2 ενώ το i αυξάνεται

i	Εντολή 2
1	n
2	$n-1$
3	$n-2$
:	:
n - 1	1

Έτσι ο συνολικός αριθμός εκτέλεσης της 2 είναι $\sum_{i=1}^{n} i$ δηλαδή $n(n+1)/2$.

Η εντολή 3 (σύγκριση) εκτελείται μία φορά λιγότερο από την 2 για κάθε τιμή του i δηλ. $n(n-1)/2$.

Για να υπολογίσουμε τον συνολικό υπολογιστικό χρόνο του αλγόριθμου προσθέτουμε τον χρόνο εκτέλεσης κάθε εντολής. Επειδή οι εντολές είναι διαφορετικές, για να είμαστε ακριβείς θα πρέπει να γνωρίζουμε τον μεταγλωττιστή και τον Η/Υ που θα εκτελεστούν. Για να μπορούμε να γενικεύσουμε τα συμπεράσματά μας κάνουμε την υπόθεση ότι κάθε απλή εντολή εκτελείται στον ίδιο χρόνο. Έτσι ο συνολικός χρόνος του αλγόριθμου είναι ο αριθμός των εντολών οι οποίες θα εκτελεστούν. Ο πίνακας 2.2 δείχνει ότι ο αριθμός που θα εκτελεστούν οι εντολές 4-6 ποικίλει. Ο ακριβής αριθμός εξαρτάται από την ήδη υπάρχουσα σειρά των στοιχείων του πίνακα. Εάν ο πίνακας είναι ήδη ταξινομημένος τότε η συνθήκη της εντολής `if {3}` είναι πάντα ψευδής και οι εντολές 4-6 δεν θα εκτελεστούν καμμία φορά. Εάν ο πίνακας είναι ήδη αντίστροφα ταξινομημένος, τότε η συνθήκη της εντολής if {3} είναι πάντα αληθής και οι εντολές 4-6 θα εκτελεστούν όσες και η εντολή 3: $n(n+1)/2$.

Πίνακας 2.2. Αριθμός φορών που εκτελείται κάθε εντολή

Εντολή	Πλήθος
1	n-1
2	$n(n+1)/2$
3	$n(n-1)/2$
4-6	Ποικίλει: Max: $n(n-1)/2$ Min: 0

Ο συνολικός αριθμός των εντολών που θα εκτελεστούν και συνεπώς ο συνολικός χρόνος που απαιτείται για να ταξινομηθεί ο πίνακας Α, είναι το άθροισμα των φορών που εκτελείται η κάθε εντολή, δηλ.

$$(n-1) + (n(n+1)/2) + (n(n-1)/2) + X \approx n^2 + n + X$$

όπου X είναι μεταξύ 0 και $n(n\text{-}1)/2$. Αυτή η έκφραση για τον υπολογιστικό χρόνο κυριαρχείται από τον όρο n^2. Δηλαδή καθώς το n αυξάνεται, ο υπολογιστικός χρόνος αυξάνεται με το τετράγωνο του n (n^2). Έτσι η πολυπλοκότητα του αλγόριθμου είναι $O(n^2)$ και αυτό είναι ανεξάρτητο από την σειρά των στοιχείων στον πίνακα, από τον μεταγλωτιστή, ή από τον Η/Υ που θα χρησιμοποιηθεί.

Θα πρέπει να σημειώσουμε ότι αυτή η έκφραση της πολυπλοκότητας αν και μας παρέχει ένα τρόπο αποφυγής των "κακών" αλγορίθμων, δεν είναι αρκετά ακριβής για να συγκρίνουμε δύο διαφορετικά προγράμματα που υλοποιούν τον ίδιο αλγόριθμο. Σε πολλές περιπτώσεις μπορούμε να βελτιώσουμε τον χρόνο εκτέλεσης ενός προγράμματος αλλά η έκφραση της πολυπλοκότητας του αλγόριθμου δεν θα αλλάξει.

3. Ταξινομήσεις με Ανταλλαγή (Exchange Sorting)

3.1. Bubble Sort

Ίσως η πιο γνωστή τεχνική ταξινόμησης. Το όνομα οφείλεται στο ότι σε διαδοχικές επαναλήψεις του πίνακα οι μικρότερες τιμές αναδύονται σαν φυσαλίδες σε κατάλληλες θέσεις. Ο αλγόριθμος ταξινόμησης bubble sort (Σχ. 3.1) λειτουργεί συγκρίνοντας κάθε στοιχείο με το επόμενό του, ξεκινώντας από την μία άκρη του πίνακα μέχρι την άλλη. Γενικά για να ταξινομηθεί ο πίνακας χρειάζονται N-1 «περάσματα» του πίνακα, όπου N το πλήθος των στοιχείων.

Παρακάτω δίνεται ένα παράδειγμα λειτουργία της "bubble sort" (σχήμα 3.1) σε ταξινόμηση κατά αύξουσα σειρά. Έστω ότι έχουμε να ταξινομήσουμε τον παρακάτω πίνακα:

15	4	7	3	21	13	2	19	8

Ξεκινάμε με την σύγκριση των δύο πρώτων αριθμών, δηλ. των 15 και 4. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	**15**	7	3	21	13	2	19	8

Συνεχίζουμε με την των τιμών σύγκριση της $2^{ης}$ και $3^{ης}$ θέσης, δηλ. των 15 και 7. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	**7**	**15**	3	21	13	2	19	8

Συνεχίζουμε με την σύγκριση των τιμών της $3^{ης}$ και $4^{ης}$ θέσης, δηλ. των 15 και 3. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	7	**3**	**15**	21	13	2	19	8

Συνεχίζουμε με την σύγκριση των τιμών της 4ης και 5ης θέσης, δηλ. των 15 και 21. Επειδή η σειρά τους είναι σωστή, συνεχίζουμε με την σύγκριση των τιμών της 5ης και 6ης θέσης, δηλ. των 21 και 13. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	7	3	15	13	21	2	19	8

Συνεχίζουμε με την σύγκριση των τιμών της 6ης και 7ης θέσης, δηλ. των 21 και 2. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	7	3	15	13	2	21	19	8

Συνεχίζουμε με την σύγκριση των τιμών της 7ης και 8ης θέσης, δηλ. των 21 και 19. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	7	3	15	13	2	21	19	8

Συνεχίζουμε με την σύγκριση των τιμών της 8ης και 9ης θέσης, δηλ. των 21 και 19. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

4	7	3	15	13	2	19	21	8

Τελειώνουμε το 1ο «πέρασμα» του πίνακα με την σύγκριση των τιμών των δύο τελευταίων θέσεων του πίνακα (8η και 9η), δηλ. των 21 και 8. Επειδή η σειρά τους είναι λανθασμένη τους αντιμεταθέτουμε και ο πίνακας γίνεται:

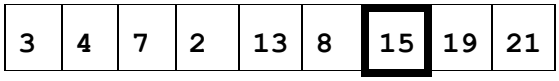
1ο πέρασμα ⟶	4	7	3	15	13	2	19	8	21

Έτσι είναι διατεταγμένα τα στοιχεία του πίνακα μετά το πρώτο πέρασμα. Παρατηρούμε ότι μετά το 1ο «πέρασμα» η μεγαλύτερη τιμή του πίνακα (δηλ. η τιμή 21) έχει αναδυθεί στην τελευταία θέση του πίνακα. Αυτή η ιδιότητα, δηλ. η ανάδυση των μεγαλύτερων τιμών προς τα πάνω (τις τελευταίες θέσεις του πίνακα) έχει δώσει και το όνομά της στον αλγόριθμο.

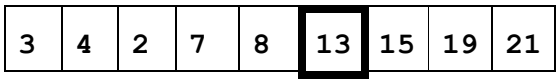
Στο επόμενο 2ο πέρασμα μετά τις συγκρίσεις και τις αντιμεταθέσεις η αμέσως μεγαλύτερη τιμή μετά το 21, δηλ. το 19 θα έχει αναδυθεί στην δεύτερη από το τέλος θέση, όπως φαίνεται παρακάτω:

	4	7	3	15	13	2	19	8	21
	4	3	7	15	13	2	19	8	21
	4	3	7	13	15	2	19	8	21
	4	3	7	13	2	15	19	8	21
2ο πέρασμα ⟶	4	3	7	13	2	15	8	19	21

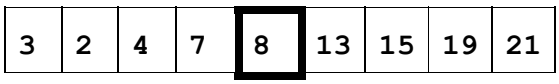
Σε κάθε «πέρασμα» αποτέλεσμα των αντιμεταθέσεων των αριθμών, είναι η αμέσως επόμενη μεγαλύτερη τιμή να αναδύεται στην αντίστοιχη θέση. Στο 3ο «πέρασμα», η 3η μεγαλύτερη τιμή, το 15, θα αναδυθεί στην 3η από το τέλος θέση.

3ο πέρασμα → 

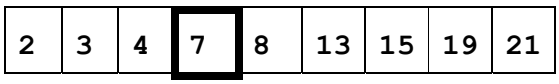
Στο 4ο «πέρασμα», η 4η μεγαλύτερη τιμή, το 13, θα αναδυθεί στην 4η από το τέλος θέση.

4ο πέρασμα → 

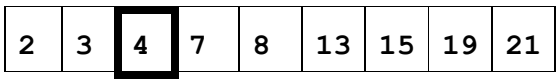
Στο 5ο «πέρασμα», η 5η μεγαλύτερη τιμή, το 8, θα αναδυθεί στην 5η από το τέλος θέση.

5ο πέρασμα → 

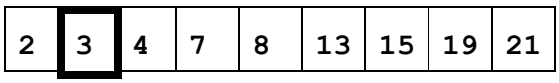
Στο 6ο «πέρασμα», η 6η μεγαλύτερη τιμή, το 7, θα αναδυθεί στην 6η από το τέλος θέση.

6ο πέρασμα → 

Στο 7ο «πέρασμα», η 7η μεγαλύτερη τιμή, το 4, θα αναδυθεί στην 7η από το τέλος θέση.

7ο πέρασμα → 

Στο 8ο «πέρασμα», η 8η μεγαλύτερη τιμή, το 3, θα αναδυθεί στην 8η από το τέλος θέση.

8ο πέρασμα → 

Δεν χρειάζεται να γίνει άλλο «πέρασμα» αφού από τις δύο τελευταίες τιμές που είχαν μείνει, η μεγαλύτερη τοποθετήθηκε στη σωστή σειρά και η μικρότερη έμεινε στην 1η θέση του πίνακα. Ο πίνακας είναι πλέον ταξινομημένος.

Εδώ μπορούμε να κάνουμε μία δεύτερη παρατήρηση (η πρώτη παρατήρηση ήταν η ανάδυση των μεγαλύτερων αριθμών). Παρατηρούμε ότι ο πίνακας ήταν ήδη ταξινομημένος μετά το 6ο πέρασμα και δεν χρειάζεται να γίνουν άλλα περάσματα.

```
public class TestBubbleSort {

    public static void main (String[] args) {
        int[] numbers = {3, 9, 6, 1, 2};

        BubbleSort.sort (numbers);

        for (int i = 0; i < numbers.length; i++)
            System.out.println (numbers[i]);

    } // method main
} // class Ex_Sort

class BubbleSort {

   public static void sort (int[] numbers) {
      int i, j, temp;
      boolean flag;
      for (i=1; i < numbers.length; i++) {
          flag=true;
          for (j=0; j < numbers.length-i; j++)
              if (numbers[j] > numbers[j+1]) {
                  temp=numbers[j];
                  numbers[j]= numbers[j+1];
                  numbers[j+1]= temp;
                  flag=false;
              }
          if (flag) return;
      }
    } // method sort
} //class Exchange_Sort
```

Σχήμα 3.1. Αλγόριθμος ταξινόμησης Bubblesort

Για να ανταποκριθούμε στις δύο αυτές παρατηρήσεις εισάγουμε τα εξής:

1. Αφού σε κάθε «πέρασμα» ο εκάστοτε μεγαλύτερος αριθμός αναδύεται στις τελευταίες θέσεις του πίνακα, τότε σε κάθε «πέρασμα» μπορούμε να ελέγχουμε μία λιγότερη θέση του πίνακα. Αυτό υλοποιείται με την αφαίρεση της τιμής του i (το οποίο αντιστοιχεί στο εκάστοτε «πέρασμα» του πίνακα) από το μήκος του πίνακα στην εσωτερική ανακύκλωση (μεταβλητή ελέγχου j) του κώδικα του Σχ. 3.1:

   ```
   for (j=0; j < numbers.length-i; j++)
   ```

2. Χρησιμοποιούμε μία μεταβλητή (πχ. flag) η οποία θα σηματοδοτεί εάν ο πίνακας είναι ταξινομημένος ή όχι. Πριν από κάθε πέρασμα του πίνακα αρχικοποιούμε αυτή την μεταβλητή σε κάποια συγκεκριμένη τιμή (πχ. flag=true). Εάν κατά την σύγκριση δύο τιμών χρειαστεί να ανταλλαχθούν τότε αλλάζουμε και την τιμή της μεταβλητής (πχ. flag=false). Με το τέλος του "περάσματος" ελέγχουμε την τιμή της μεταβλητής. Εάν η μεταβλητή διατηρεί την αρχική τιμή


της, τότε αυτό σημαίνει ότι δεν έχει γίνει κάποια ανταλλαγή τιμών, επομένως ο πίνακας είναι ταξινομημένος και δεν χρειάζεται να γίνει άλλο "πέρασμα". Εάν η μεταβλητή έχει αλλάξει τιμή, σημαίνει ότι κάποιες τιμές έχουν αλλάξει θέση, επομένως ο πίνακας δεν ήταν ταξινομημένος και χρειάζεται να γίνει και άλλο "πέρασμα".

Ο αλγόριθμος έχει πολυπλοκότητα $O(n^2)$ και αυτό φαίνεται στον πίνακα 3.1 όπου παρουσιάζονται οι χρόνοι (σε δευτερόλεπτα) ταξινόμησης ενός πίνακα n τυχαίων στοιχείων.

Πίνακας 3.1. Ενδεικτικοί χρόνοι "bubble sort"

Πλήθος στοιχείων (*n*)	Χρόνος (sec)
100	0,9
200	3,5
400	13,7
800	55,2

3.2. Ταξινόμηση με Εισαγωγή (Insertion Sort)

Η ταξινόμηση "bubble sort" γενικά δεν είναι κατάλληλη γιατί είναι πολύ αργή. Χρειαζόμαστε μία τεχνική η οποία χρησιμοποιεί μόνο λίγο επιπλέον χώρο αλλά είναι πιο γρήγορη από την "bubble sort". Μία τέτοια αποτελεσματική μέθοδος είναι η ταξινόμηση με εισαγωγή.

Η ταξινόμηση με εισαγωγή λειτουργεί ως εξής: Παίρνουμε κάθε στοιχείο και το τοποθετούμε στη σωστή θέση στον ταξινομημένο πίνακα αριστερά του τρέχοντος στοιχείου. Εάν ξεκινήσουμε από το πρώτο στοιχείο, τότε αφού δεν υπάρχουν άλλα στοιχεία αριστερά, το στοιχείο αυτό βρίσκεται στη σωστή θέση (μέχρι τώρα). Επομένως δεν χρειάζεται να ξεκινήσουμε από το πρώτο στοιχείο.

Έστω ότι έχουμε να ταξινομήσουμε τον παρακάτω πίνακα

15	4	7	3	21	13	2	19	8

Το δεύτερο στοιχείο (4) έχει αριστερά του ένα πίνακα με ένα μόνο στοιχείο (15), ο οποίος αφού έχει ένα μόνο στοιχείο είναι ταξινομημένος. Τοποθετούμε το 4 στη σωστή θέση (πριν το 15) και το 15 μετακινείται μία θέση δεξιά. Έτσι ο πίνακας γίνεται:

4	15	7	3	21	13	2	19	8

Συνεχίζουμε με το επόμενο-τρίτο στοιχείο (7) το οποίο για να τοποθετηθεί στη σωστή θέση θα πρέπει να πάει μετά το 4 και το 15 να μετακινηθεί δεξιά μέχρι τη θέση του τρέχοντος στοιχείου. Έτσι ο πίνακας ταξινομείται σιγά-σιγά, ένα στοιχείο κάθε φορά και γίνεται:

4	7	15	3	21	13	2	19	8

Συνεχίζουμε με το τέταρτο στοιχείο (3) το οποίο πρέπει να τοποθετηθεί πρώτο και όλα τα υπόλοιπα να μετακινηθούν μία θέση δεξιά μέχρι τη θέση του τρέχοντος στοιχείου και ο πίνακας γίνεται:

3	4	7	15	21	13	2	19	8

Συνεχίζουμε με το επόμενο-πέμπτο στοιχείο (21) το οποίο πρέπει να μείνει στη θέση που βρίσκεται αφού είναι μεγαλύτερο από το 15. Έτσι ο πίνακας γίνεται:

3	4	7	15	21	13	2	19	8

Η διαδικασία συνεχίζεται με τους επόμενους αριθμούς. Κάθε στοιχείο τοποθετείται στη σωστή θέση και τα επόμενα μετακινούνται κατά μία θέση, μέχρι και το τελευταίο στοιχείο του πίνακα. Με την τοποθέτηση του κάθε στοιχείου στη θέση του ο πίνακας θα γίνεται μετά από την τοποθέτηση κάθε στοιχείου. Το 2, το 19 και τέλος το 8.

3	4	7	13	15	21	2	19	8
2	3	4	7	13	15	21	19	8
2	3	4	7	13	15	19	21	8
2	3	4	7	8	13	15	19	21

Στο Σχ. 3.2 δίνεται ο κώδικας υλοποίησης του αλγόριθμου της ταξινόμησης με εισαγωγή. Το πρόγραμμα χρησιμοποιεί δύο βρόχους επανάληψης για να ταξινομήσει ένα πίνακα ακεραίων. Ο εξωτερικός βρόχος ελέγχει τον δείκτη του πίνακα όπου θα εισαχθεί η επόμενη τιμή. Ο εσωτερικός βρόχος ελέγχει την τρέχουσα τιμή η οποία θα εισαχθεί, με τιμές οι οποίες είναι στα αριστερά του πίνακα και οι οποίες αποτελούν ένα ταξινομημένο υποσύνολο του συνολικού πίνακα. Εάν η τρέχουσα τιμή είναι μικρότερη από την τιμή στη θέση position τότε η τιμή αυτή μετακινείται δεξιά. Η μετακίνηση συνεχίζει και με τις υπόλοιπες τιμές. Κάθε επανάληψη του εξωτερικού βρόχου προσθέτει μία ακόμη τιμή στο ταξινομημένο υποσύνολο του πίνακα.

```
public class InsSort {

    public static void main (String[] args) {
        int[] numbers = {3, 9, 6, 1, 2};

        InsertionSort.sort (numbers);

        for (int i = 0; i < numbers.length; i++)
            System.out.println (numbers[i]);
    } // method main
```

```
} // class Ins_Sort

class InsertionSort {

    public static void sort (int[] numbers) {

        for (int i=1; i < numbers.length; i++) {
            int current = numbers[i];
            int position=i;

            // shift larger values to the right
            while (position>0 && numbers[position-1] > current) {
                numbers[position] = numbers[position-1];
                position--;
            }
            numbers[position] = current;
        }
    } // method sort
} // class Insertion_Sort
```

Σχήμα 3.2. Αλγόριθμος ταξινόμησης με εισαγωγή (έκδοση 1)

Το Σχ. 3.3 παρουσιάζει μία διαφορετική υλοποίηση του αλγόριθμου ταξινόμησης με εισαγωγή. Η υλοποίηση λειτουργεί ως εξής. Αρχίζουμε από την αρχή του πίνακα και συγκρίνουμε την πρώτη τιμή με όλες τις επόμενες διαδοχικά. Όταν φτάνουμε σε κάποιο στοιχείο μεγαλύτερο από αυτό που συγκρίνουμε τότε το εισάγουμε σε αυτή την θέση, μετακινώντας κατά μία θέση όλα τα υπόλοιπα στοιχεία.

```
public class InsSort_2 {

    public static void main (String[] args) {
        int[] numbers = {3, 9, 6, 1, 2};

        InsertionSort_2.sort (numbers);

        for (int i = 0; i < numbers.length; i++)
            System.out.println (numbers[i]);
    } // method main
} // class InsSort_2

class InsertionSort_2 {
    public static void sort (int[] numbers) {
        boolean found;
        int j, temp;
        for (int i=1; i < numbers.length; i++) {
            j=0;
            found=false;
            while (j<i && ! found)
                if (numbers[i] < numbers[j])
                    found=true;
```

```
                else
                     j=j+1;

            if (found) {
                 temp=numbers[i];
                 for (int k=i-1; k>=j; k--)
                      numbers[k+1]=numbers[k];
                 numbers[j]=temp;
            }
        }
    } // method sort
} // class InsertionSort_2
```

Σχήμα 3.3. Αλγόριθμος ταξινόμησης με εισαγωγή (έκδοση 2)

3.3. Ταξινόμηση με Επιλογή (Selection Sort)

Η ταξινόμηση με επιλογή λειτουργεί τοποθετώντας κάθε τιμή (μία κάθε φορά) στην σωστή, τελική και ταξινομημένη θέση. Με άλλα λόγια, για κάθε θέση του πίνακα ο αλγόριθμος επιλέγει την τιμή η οποία θα πρέπει να πάει σε εκείνη την θέση.

Ο αλγόριθμος γενικά λειτουργεί ως εξής: Βρίσκουμε την μικρότερη τιμή του πίνακα και την ανταλλάσσουμε με την τιμή στην πρώτη θέση του πίνακα. Μετά βρίσκουμε την μικρότερη τιμή από τις υπόλοιπες (εκτός της πρώτης) και την ανταλλάσουμε με την δεύτερη θέση του πίνακα. Η διαδικασία συνεχίζεται για κάθε θέση του πίνακα.

```
public class SelSort {

    public static void main (String[] args) {
        int[] numbers = {3, 9, 6, 1, 2};

        SelectionSort.sort (numbers);

        for (int i = 0; i < numbers.length; i++)
            System.out.println (numbers[i]);
    } // method main
} // class SelSort

class SelectionSort {

    public static void sort (int[] numbers) {
        int min, temp;
        for (int i = 0; i < numbers.length-1; i++){

            // find position of minimum value
            min=i;
            for (int k=i+1; k<numbers.length; k++)
```

```
                if (numbers[k] < numbers[min])
                    min=k;

            //swap the values in positions "min" and "i"
            temp = numbers[min];
            numbers[min] = numbers[i];
            numbers[i] = temp;
        }
    } // method sort
} //class SelectionSort
```

Σχήμα 3.4. Αλγόριθμος ταξινόμησης με επιλογή

4. Αναδρομικές Τεχνικές Ταξινόμησης (Recursive Sorting)

Με τους συνήθης σειριακούς επεξεργαστές η μέγιστη δυνατή ταχύτητα ταξινόμησης είναι $O(n\log_2 n)$. Δύο τέτοιες τεχνικές που έχουν ιδιαίτερο ενδιαφέρον είναι η *γρήγορη αναζήτηση* (*quick sort*) και η *ταξινόμηση με συγχώνευση* (*merge sort*). Η γρήγορη ταξινόμηση, όπως λέει και το όνομά της είναι η πιο γρήγορη γενικού σκοπού μέθοδος ταξινόμησης. Η μέθοδος συγχώνευσης είναι επίσης πολύ γρήγορη και έχει αρκετά συμπαγή κώδικα. Και οι δύο τεχνικές κωδικοποιούνται εύκολα αναδρομικά, γιατί έτσι είναι πιο εύκολο να κατανοηθούν και να υλοποιηθούν.

4.1. Γρήγορη Ταξινόμηση (Quick Sort)

Βασική αρχή είναι ότι προσπαθούμε να χωρίσουμε τα στοιχεία του πίνακα, σε δύο υπο-πίνακες έτσι ώστε ο ένας να περιέχει όλα τα στοιχεία τα οποία είναι μικρότερα από κάποια συγκεκριμένη τιμή, ενώ ο άλλος όλα τα μεγαλύτερα από αυτή την τιμή. Ως τέτοια τιμή, μπορεί να επιλεγεί οποιαδήποτε τιμή του πίνακα. Στην γρήγορη ταξινόμηση η τιμή αυτή ονομάζεται "κεντρική (*pivot*)".

Θεωρώντας ότι τα στοιχεία ταξινομούνται κατά αύξουσα σειρά, η ιδέα είναι όπως φαίνεται στο σχήμα 4.1:

- η κεντρική τιμή να τοποθετηθεί στη σωστή θέση της στον αρχικό πίνακα
- όλα τα στοιχεία τα οποία είναι μικρότερα της κεντρικής τιμής και βρίσκονται στα δεξιά της, να μετακινηθούν στα αριστερά της κεντρικής τιμής
- όλα τα στοιχεία τα οποία είναι μεγαλύτερα της κεντρικής τιμής και βρίσκονται στα αριστερά της, να μετακινηθούν στα δεξιά της κεντρικής τιμής

Όπως ήδη τονίστηκε, βασικό χαρακτηριστικό της αλγοριθμικής διαδικασίας είναι ότι κάθε στοιχείο του αριστερού υπο-πίνακα είναι μικρότερο από κάθε στοιχείο του δεξιού υπο-πίνακα.

Σχ. 4.1. Ένας πίνακας ακεραίων χωρίζεται σε δύο υπο-πίνακες με όλα τα στοιχεία αριστερά της κεντρικής τιμής να είναι μικρότερα των στοιχείων που βρίσκονται στα δεξιά της.

Η ίδια διαδικασία μπορεί να επαναληφθεί για κάθε έναν από τους δύο υπο-πίνακες χωρίζοντας κάθε έναν σε δύο επιπλέον υπο-πίνακες. Συνεχίζοντας αναδρομικά στους υπο-πίνακες, τελικά ταξινομεί ολόκληρο τον αρχικό πίνακα. Χρησιμοποιώντας ψευδοκώδικα, αυτό μπορεί να εκφραστεί ως:

```
μέθοδος quickSort (ARRAYTYPE A[], int L,int U);
begin
   if l>f then
       begin
           "Διαχωρισμός του πίνακα A σε δύο υπο-πίνακες έτσι ώστε
           A[i]<=A[Mid] για κάθε i του αριστερού υπο-πίνακα και
           A[i]>A[Mid] για κάθε i του δεξιού."
           Quick_sort(A,L,Mid-1);
           Quick_sort(A,Mid+1,U)
        end
end;
```

Ο διαχωρισμός γίνεται σε δύο βήματα. Στο πρώτο γίνεται η επιλογή της κατάλληλης κεντρικής τιμής και στο δεύτερο ο διαχωρισμός του πίνακα σε δύο μέρη.

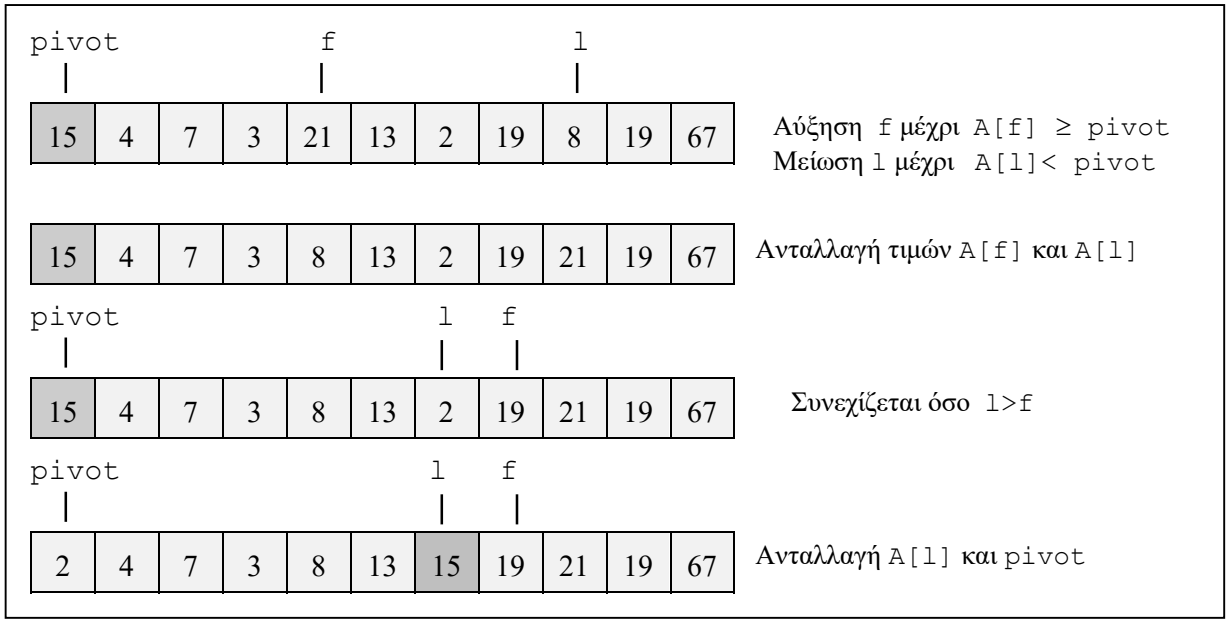
Οποιαδήποτε τιμή του πίνακα μπορεί να χρησιμοποιηθεί ως "κεντρική". Μία απλή λύση είναι να επιλεγεί ως κεντρική η πρώτη θέση του πίνακα, αλλά εάν ο πίνακας είναι ήδη ταξινομημένος, τότε η τιμή αυτή θα διαιρέσει τον πίνακα σε δύο μέρη, ένα με μέγεθος 0 και ένα με μέγεθος n-1, κάτι που δεν είναι αποτελεσματικό. Η καλύτερη περίπτωση θα ήταν να χωρισθεί ο πίνακας σε δύο ίσα μέρη. Για να γίνει όμως αυτό χρειάζεται να βρούμε την μέση τιμή των στοιχείων του πίνακα, κάτι που εισάγει σημαντικό υπολογιστικό κόστος. Έτσι οδηγούμαστε στην χρήση της τιμής της μεσαίας θέσης του πίνακα η οποία στην περίπτωση του ήδη ταξινομημένου πίνακα θα τον χωρίσει σε δύο μέρη με ίσο πλήθος στοιχείων.

Ο διαχωρισμός επιτυγχάνεται με την χρήση δύο δεικτών: το `f` γενικά δείχνει σε τιμές του πίνακα μικρότερες της κεντρικής και το `l` στις μεγαλύτερες τιμές από την κεντρική.

Ξεκινάμε θέτοντας το f στην αριστερή άκρη (αρχή) του πίνακα και το l στην δεξιά άκρη (τέλος) του πίνακα.

Ακολουθεί η σύγκριση του στοιχείου που δείχνει ο δείκτης f, δηλ. το A[f] με την κεντρική τιμή. Εάν η κεντρική τιμή είναι μεγαλύτερη (pivot > A[f]), τότε το f αυξάνεται ώστε να δείχνει στην επόμενη θέση. Το f συνεχίζει να αυξάνεται όσο η κεντρική τιμή είναι μεγαλύτερη από την τιμή που περιέχεται στη θέση του πίνακα που δείχνει το f. Όταν η κεντρική τιμή γίνει μικρότερη από το A[f] (βρεθεί δηλ. στοιχείο το οποίο είναι μεγαλύτερο της κεντρικής τιμής), τότε σταματάει η αύξηση του f.

Η διαδικασία συνεχίζεται με το δείκτη l. Συγκρίνουμε το στοιχείο που δείχνει ο δείκτης l, δηλ. το A[l] με την κεντρική τιμή. Εάν pivot < A[l], τότε το l μειώνεται ώστε να δείχνει στην προηγούμενη θέση του πίνακα. Το l συνεχίζει να μειώνεται όσο η κεντρική τιμή είναι μικρότερη του A[l]. Όταν η κεντρική τιμή γίνει μεγαλύτερη από το A[l] (βρεθεί δηλ. στοιχείο το οποίο είναι μικρότερο της κεντρικής τιμής), τότε σταματάει η μείωση του l.



Σχ. 4.2. Διαδικασία διαχωρισμού πίνακα κατά την γρήγορη ταξινόμηση

Συνεχίζουμε με τη σύγκριση των τιμών των f και l. Εάν το f βρίσκεται αριστερά του l τότε ανταλλάσσονται τα περιεχόμενα των θέσεων που δείχνουν δηλ. τα A[f] και A[l].

Η διαδικασία συνεχίζεται μέχρι το l να περάσει αριστερά του f, δηλαδή όταν l<f (σχ. 4.2.). Τότε ανταλλάσσουμε το pivot με το στοιχείο που δείχνει το f ή το l.

Η διαδικασία επαναλαμβάνεται αναδρομικά για τους δύο υπο-πίνακες, αριστερά και δεξιά της κεντρικής τιμής.

Το σχήμα 4.3. παρουσιάζει τον κώδικα της γρήγορης ταξινόμησης με την χρήση της διαδικασίας διαχωρισμού που περιγράφηκε νωρίτερα.

```
public class Q_Sort {
    public static void main (String[] args) {
        int[] numbers={15, 9, 6, 11, 2};
        QuickSort.sort(numbers);
        for (int i=0; i<numbers.length; i++)
            System.out.println (numbers[i]);
    }
}

class QuickSort {
    public static void swap (int A[], int x, int y) {
        int temp = A[x];
        A[x] = A[y];
        A[y] = temp;
    }

    public static int partition (int A[], int f, int l) {
        int retValue=0;
        int lowerLimit = f;
        int mid=(f+l)/2;
        swap(A,f,mid);

        int pivot =A[f];
        f++;

        while (f<l) {
            while (A[f] <= pivot && f< l) f++;
            while (A[l] >= pivot && f<= l) l--;
            if (f<l) swap (A,f,l);
        }
        if (pivot > A[f]) {
            swap (A,f,lowerLimit);
            retValue=f;
        }
        else {
            if (pivot >= A[l]) {
                swap (A,l,lowerLimit);
                retValue=l;
            }
        }
        return retValue;
    } //  method partition

    public static void qSort (int[] A, int f, int l) {
        final int MIN=5;
        if (l-f>0) {
/*          if (l-f+1<MIN)
              On2_Sort(A,f,l);
           else  {*/
              int pivot_index = partition (A, f, l);
```

```
                qSort (A, f, pivot_index-1);
                qSort (A, pivot_index+1, l);
//            }
        }
    }
    public static void sort (int[] A)  {
        qSort (A, 0, A.length-1);
    }
} // class QuickSort
```

Σχήμα 4.3. Αλγόριθμος γρήγορης ταξινόμησης

Επειδή η διαδικασία διαχωρισμού είναι αρκετά χρονοβόρα ακόμη και για μικρούς πίνακες, είναι καλό για μικρούς πίνακες να μην καλείται η Quick_sort αλλά κάποια άλλη μέθοδος με πολυπλοκότητα $O(n^2)$. Αυτό φαίνεται και μέσα στον παραπάνω κώδικα της `Quick_sort`, όπου για μέγεθος πίνακα μικρότερο κάποιας τιμής MIN καλείται η μέθοδος `On2_Sort(A, f, l)`, η οποία μπορεί να είναι κάποια μέθοδος ταξινόμησης με ανταλλαγή (με εισαγωγή, ή με επιλογή, ή ακόμη και bubble sort).

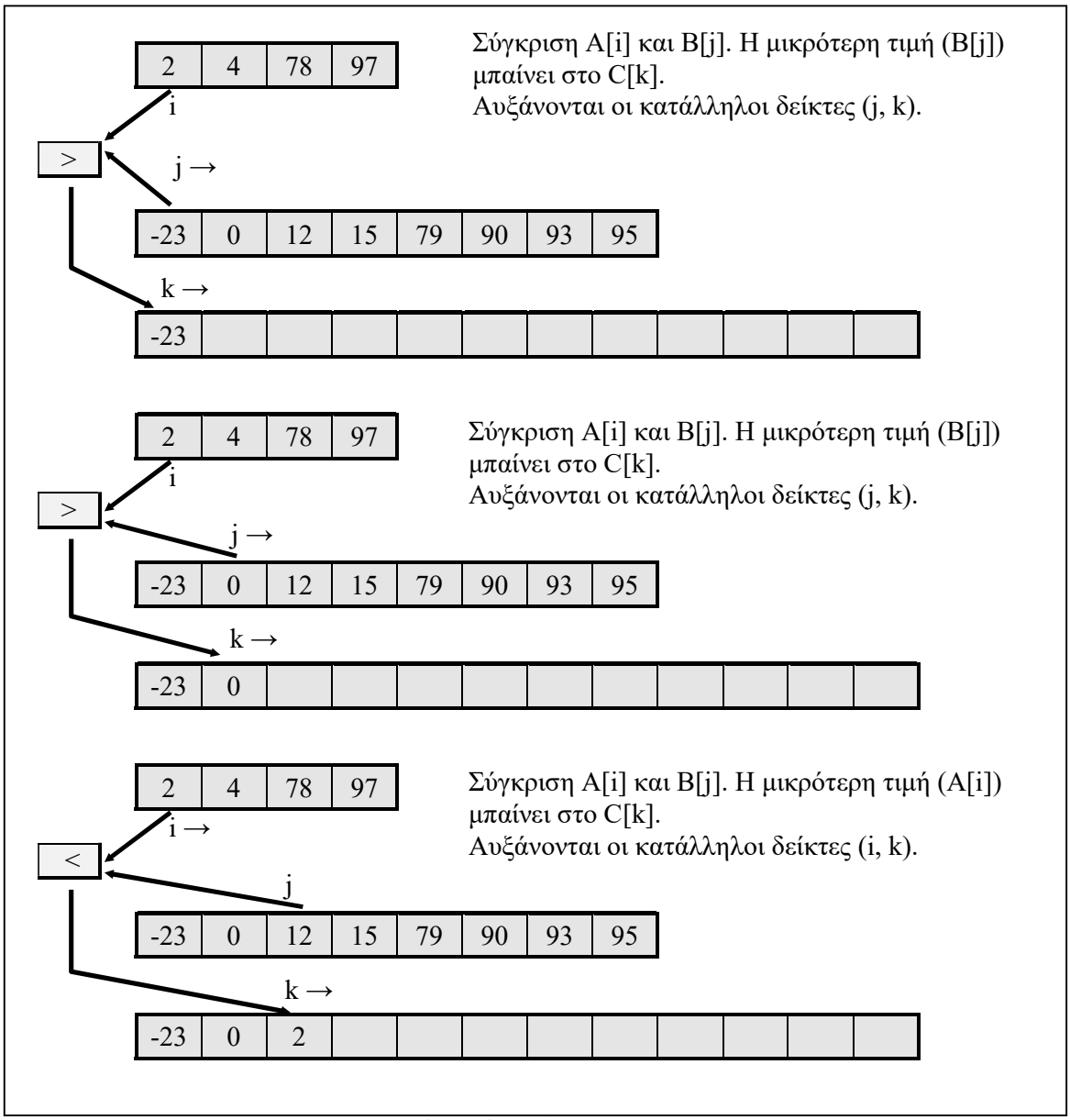
4.2. Ταξινόμηση με Συγχώνευση (Merge Sort)

Βασική Ιδέα: Χωρίζουμε στη μέση τον πίνακα και συγχωνεύουμε τους δύο ταξινομημένους υπο-πίνακες. Προφανώς στη συντριπτική πλειοψηφία των περιπτώσεων οι δύο υπο-πίνακες δεν είναι ταξινομημένοι. Γι' αυτό η διαδικασία συνεχίζεται για καθένα από τους δύο υπο-πίνακες για όσο κάθε νέος υπο-πίνακας έχει περισσότερα από 1 στοιχεία. Οι συγχωνεύσεις των υπο-πινάκων γίνονται αναδρομικά, ξεκινώντας από τους υπο-πίνακες με ένα στοιχείο, οι οποίοι είναι ταξινομημένοι, αφού έχουν μόνο ένα στοιχείο.

Δηλαδή, η ταξινόμηση με συγχώνευση ενός πίνακα υλοποιείται χρησιμοποιώντας επαναλαμβανόμενες συγχωνεύσεις με αναδρομική κλήση της διαδικασίας συγχώνευσης ως εξής: χωρίζουμε τον πίνακα σε δύο μέρη και ταξινομούμε πρώτα το αριστερό μισό μέρος του πίνακα και κατόπιν το δεξιό. Ακολουθεί η συγχώνευση των δύο πινάκων. Η διαδικασία επαναλαμβάνεται αναδρομικά για τα δύο μέρη του πίνακα, μέχρι το μήκος των υπο-πινάκων οι οποίοι προκύπτουν να είναι μικρότερο ή ίσο του ένα.

Διαδικασία συγχώνευσης δύο πινάκων: Χρησιμοποιούμε δύο δείκτες, ένα για κάθε πίνακα, και ένα τρίτο πίνακα για την αποθήκευση της συγχώνευσης των δύο αρχικών πινάκων. Συγκρίνουμε τις δύο τιμές των πινάκων εισόδου που δείχνουν οι δείκτες και αντιγράφουμε την μικρότερη τιμή στην κατάλληλη θέση του πίνακα εξόδου. Ο κώδικας δίνεται στην διαδικασία Merge. Μετρώντας από τις γραμμές του κώδικα βλέπουμε ότι η διαδικασία είναι O(m+n), όπου m είναι το πλήθος των στοιχείων του πρώτου πίνακα και n το πλήθος των στοιχείων του δεύτερου πίνακα. Έτσι ο αλγόριθμος είναι πολυπλοκότητας O(n).

Το σχήμα 4.4 περιγράφει την διαδικασία συγχώνευσης δύο ταξινομημένων πινάκων σε ένα τρίτο πίνακα (προσοχή: η διαδικασία αυτή αποτελεί μόνο ένα μέρος της ταξινόμησης με συγχώνευση). Ενδεικτικός κώδικας της διαδικασίας συγχώνευσης δύο ήδη ταξινομημένων πινάκων δίνεται στη μέθοδο `merge` του σχήματος 4.5.



Σχήμα 4.4. Συγχώνευση δύο ήδη ταξινομημένων πινάκων σε ένα τρίτο

Η διαδικασία ταξινόμησης με συγχώνευση λειτουργεί με τον ίδιο τρόπο ανεξάρτητα από την διάταξη των στοιχείων (είτε τα δεδομένα είναι ταξινομημένα είτε έχουν τυχαία διάταξη).

Περιγραφή Ταξινόμησης: Ο πίνακας διαιρείται σε δύο υποπίνακες. Εάν κάθε υποπίνακας έχει περισσότερα από 1 στοιχεία τότε διαιρείται ξανά σε δύο υποπίνακες. Η διαίρεση των υποπινάκων σε νέους συνεχίζεται μέχρι οι νέοι υποπίνακες να έχουν μόνο ένα στοιχείο έτσι ώστε να είμαστε

σίγουροι ότι είναι ταξινομημένοι, αφού έχουν μόνο ένα στοιχείο. Μετά αρχίζουν οι συγχωνεύσεις των ταξινομημένων υποπινάκων, ξεκινώντας από τους υποπίνακες που έχουν μόνο ένα στοιχείο.

Το σχήμα 4.5 παρουσιάζει ένα παράδειγμα ταξινόμησης με συγχώνευση για τον πίνακα με στοιχεία: [17, 29, 12, 32, 9, 20, 19, 5, 15, 8, 27, 14]

Αρχικά, τα δεδομένα διαιρούνται στη μέση στους δύο υποπίνακες:

[17, 29, 12, 32, 9, 20] και [19, 5, 15, 8, 27, 14]

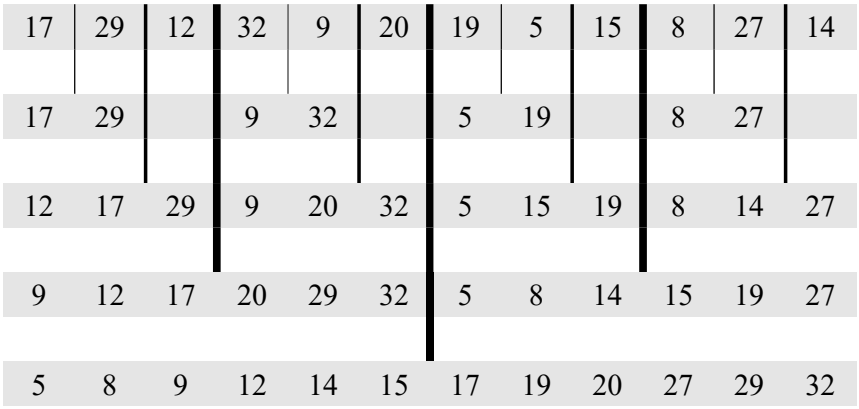
Στο επόμενο βήμα (επόμενη αναδρομική κλήση), κάθε υποπίνακας διαιρείται επίσης σε δύο νέους υποπίνακες αφού έχουν περισσότερα από ένα στοιχεία. Η διαδικασία συνεχίζεται μέχρι κάθε υποπίνακας να έχει μόνο ένα στοιχείο, οπότε αρχίζει η συγχώνευση.

[17, 29, 12, 32, 9, 20] → [17, 29, 12] και [32, 9, 20]
[17, 29, 12] → [17, 29] και [12]
[17, 29] → [17] και [29]
Συγχ: [17] και [29] → [17, 29]
Συγχ: [17, 29] και [12] → [12, 17, 29]
[32, 9, 20] → [32, 9] και [20]
[32, 9] → [32] και [9]
Συγχ: [32] και [9] → [9, 32]
Συγχ: [9, 32] και [20] → [9, 20, 32]
Συγχ: [12, 17, 29] και [9, 20, 32] → [9, 12, 17, 20, 29, 32]

[19, 5, 15, 8, 27, 14] → [19, 5, 15] και [8, 27, 14]
[19, 5, 15] → [19, 5] και [15]
[19, 5] → [19] και [5]
Συγχ: [19] και [5] → [5, 19]
Συγχ: [5, 19] και [15] → [5, 15, 19]
[8, 27, 14] → [8, 27] και [14]
[8, 27] → [8] και [28]
Συγχ: [8] και [27] → [8, 27]
Συγχ: [8, 27] και [14] → [8, 14, 27]
Συγχ: [5, 15, 19] και [8, 14, 27] → [5, 8, 14, 15, 19, 27]

Τελική συγχώνευση:

[9, 12, 17, 20, 29, 32] και [5, 8, 14, 15, 19, 27] → [5, 8, 9, 12, 14, 15, 17, 19, 20, 27, 29, 32]



Σχήμα 4.5. Παράδειγμα ταξινόμησης με συγχώνευση

Το σχήμα 4.6 παρουσιάζει τον κώδικα της διαδικασίας ταξινόμησης με συγχώνευση.

```
public class TestMergeSort {
    public static void main (String[] args) {
        int[] numbers={3, 9, 6, 1, 2};
        MergeSort.sort(numbers);
        for (int i=0; i<numbers.length; i++)
            System.out.println (numbers[i]);
    }
}

class MergeSort {

    public static void sort (int[] A) {
        mSort (A, 0, A.length-1);
    }

    public static void mSort (int[] A, int f, int l) {
        if (f==l) return;
        int mid=(f+l)/2;       // Μεσαία θέση του πίνακα
        mSort (A, f, mid);     // Αναδρομική κλήση για το 1ο μισό
        mSort (A, mid+1, l);  // Αναδρομική κλήση για το 2ο μισό
        merge(A,f,l,mid);// Συγχώνευση των δύο υπο-πινάκων
    }

    public static void merge (int A[], int f, int l, int mid) {

        int n = l-f+1;  // size of the range to be merged

        // temporary array b
        int[] b = new int[n];

        int i1=f, i2=mid+1;
        int j=0;    // next open position in b
```

```
        // merge both halves into a temporary array b
        while (i1<=mid && i2<=l) {
            if (A[i1] < A[i2]) {
                b[j]=A[i1];
                i1++;
            }
            else {
                b[j]=A[i2];
                i2++;
            }
            j++;
        }

        // Copy the rest elements of the first half of the array
        while (i1<=mid) {
            b[j]=A[i1];
            i1++;
            j++;
        }

        // Copy the rest elements of the second half of the array
        while (i2<=l) {
            b[j]=A[i2];
            i2++;
            j++;
        }

        // copy back from the temporary array
        for (j=0; j<n; j++)
            A[f+j]=b[j];

    } //  method merge

} // class MergeSort
```

Σχήμα 4.6. Αλγόριθμος ταξινόμησης με συγχώνευση

Μπορούμε να αναλύσουμε την χρονική πολυπλοκότητα της συγχώνευσης αναλύοντας την συμπεριφορά του αλγόριθμου κατά την ταξινόμηση ενός πίνακα. Είδαμε ότι η ταξινόμηση με συγχώνευση χωρίζει πάντα τον πίνακα σε δύο ίσους υπο-πίνακες. Ένας πίνακας μεγέθους **n** θα χωριστεί $n\log_2 n$ φορές. Αν και για μεγάλους πίνακες η ταξινόμηση με συγχώνευση είναι σχεδόν 4 φορές πιο αργή από την γρήγορη ταξινόμηση και επίσης χρησιμοποιεί και ένα βοηθητικό πίνακα, είναι ιδιαίτερα χρήσιμη κατά την ταξινόμηση ενός πλήθους στοιχείων τα οποία είναι πάρα πολλά για να χωρέσουν στην κύρια μνήμη του Η/Υ.

5. Σύγκριση Τεχνικών Ταξινόμησης

Οι πίνακες 5.1 και 5.2 παρουσιάζουν κάποιες τιμές χρόνων ταξινόμησης για τις πέντε μεθόδους που εξετάστηκαν. Οι τιμές δίνονται σε χιλιοστά του δευτερολέπτου (ms) και είναι το αποτέλεσμα του μέσου όρου περισσοτέρων από 1000 επαναλήψεων εκτέλεσης του κάθε αλγόριθμου σε τυχαία/διαφορετικά σύνολα δεδομένων για διαφορετικό πλήθος στοιχείων (50-10000).

Οι χρόνοι του πίνακα 5.1 μετρήθηκαν σε Η/Υ με Pentium IV, 2.4GHz και 256 M RAM.

Οι χρόνοι του πίνακα 5.2 μετρήθηκαν σε Η/Υ με Intel Core i7-4700MQ, 2.4GHz και 8 GB RAM

Πίνακας 5.1. Χρόνοι διαφόρων τεχνικών ταξινόμησης (pentium)

ΠΛΗΘΟΣ ΣΤΟΙΧΕΙΩΝ	BUBBLE	SELECTION	INSERTION	MERGE	QUICK
50	0.012	0.006	0.004	0.010	0.004
100	0.047	0.022	0.016	0.022	0.009
500	1.15	0.47	0.35	0.13	0.06
1000	4.74	1.79	1.31	0.29	0.14
2000	18.72	7.05	5.24	0.61	0.29
5000	119.04	43.97	32.94	1.71	0.80
10000	497.14	179.48	132.14	3.69	1.75

Πίνακας 5.2. Χρόνοι διαφόρων τεχνικών ταξινόμησης (i7)

ΠΛΗΘΟΣ ΣΤΟΙΧΕΙΩΝ	BUBBLE	SELECTION	INSERTION	MERGE	QUICK
1000	1.187	0.469	0.188	0.125	0.078
2000	4.687	1.687	0.719	0.203	0.172
5000	32.17	9.79	4.05	0.65	0.46
10000	151.65	38.18	16.38	1.37	1.03
20000	620.32	98.08	64.17	2.98	2.05
100000	3003.56	661.22	284.25	5.63	4.20

Ακόμη και οι τεχνικές με πολυπλοκότητα $O(n^2)$ έχουν την θέση τους στην επιλογή της κατάλληλης μεθόδου ταξινόμησης. Όταν το πλήθος των στοιχείων προς ταξινόμηση είναι μικρό (πχ. μικρότερο των 100), η μέθοδος ταξινόμησης με εισαγωγή δίνει χρόνους μόνο σχεδόν διπλάσιους της γρήγορης ταξινόμησης. Επειδή η μέθοδος με εισαγωγή είναι συμπαγής, εύκολο να κωδικοποιηθεί


και απαιτεί λίγες μετακινήσεις εγγραφών, είναι θεμιτό να χρησιμοποιείται για ταξινόμηση μικρών αρχείων.

Καθώς το πλήθος των στοιχείων αυξάνεται πρέπει να εγκαταλείψουμε τις τεχνικές με πολυπλοκότητα $O(n^2)$ και να διερευνήσουμε κάποια $O(n\log_2 n)$ τεχνική. Γενικά η γρήγορη αναζήτηση θα είναι η μέθοδος της επιλογής μας εκτός εάν υποπτευόμαστε ότι τα δεδομένα μας έχουν κάποιες ιδιότητες οι οποίες κάνουν κατάλληλη κάποια από τις άλλες τεχνικές.