ΔΙΕΘΝΕΣ ΠΑΝΕΠΙΣΤΗΜΙΟ ΤΗΣ ΕΛΛΑΔΟΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Αντικειμενοστρεφής Προγραμματισμός

(https://people.iee.ihu.gr/~adamidis/OOP/OOP_03_search.pdf)

Πολυπλοκότητα

~ ~ ~ ~

Τεχνικές και Αλγόριθμοι Αναζήτησης

Παναγιώτης Αδαμίδης
adamidis@ihu.gr

Θεσσαλονίκη, Φεβρουάριος 2020

ΠΕΡΙΕΧΟΜΕΝΑ

1. Σύγκριση Αλγορίθμων

Υποθέστε ότι μας δίνεται ένα πρόβλημα και για την λύση του αναπτύσσουμε δύο διαφορετικούς αλγόριθμους. Επιπλέον υποθέστε ότι και οι δύο αλγόριθμοι είναι σωστοί. Φυσικά θέλουμε να ξέρουμε ποιος είναι καλύτερος. Για να μπορέσουμε να βρούμε τον καλύτερο θα πρέπει να καθορίσουμε τι εννοούμε λέγοντας "καλύτερος αλγόριθμος". Σύμφωνα με ποια κριτήρια θα πρέπει να επιλέξουμε; Σε μια πρώτη αντιμετώπιση θα λέγαμε ότι ένας αλγόριθμος είναι καλύτερος από έναν άλλο όταν είναι πιο αποτελεσματικός, δηλαδή εάν με την ίδια είσοδο δεδομένων απαιτεί λιγότερα βήματα. Αν και αυτό το μέτρο δεν είναι ακριβές, συχνά μας βοηθάει να επιλέξουμε μεταξύ δύο αλγορίθμων οι οποίοι επιλύουν το ίδιο πρόβλημα.

1.1. Ο χρόνος ως μέτρο σύγκρισης

Υπάρχουν δύο συχνά αντικρουόμενοι στόχοι:

- Θα θέλαμε έναν αλγόριθμο ο οποίος είναι εύκολο να κατανοηθεί, και να κωδικοποιηθεί χωρίς σφάλματα.
- Θα θέλαμε έναν αλγόριθμο ο οποίος εκμεταλλεύεται αποτελεσματικά τους πόρους του διαθέσιμου υπολογιστικού συστήματος και ειδικά έναν ο οποίος εκτελείται όσο το δυνατό πιο γρήγορα.

Όταν γράφουμε ένα πρόγραμμα το οποίο θα χρησιμοποιηθεί ελάχιστες φορές, τότε ο πρώτος στόχος είναι πιο σημαντικός. Το κόστος του χρόνου του προγραμματιστή κατά πάσα πιθανότητα θα ξεπεράσει κατά πολύ το κόστος της εκτέλεσης του προγράμματος. Όταν έχουμε να επιλύσουμε ένα πρόβλημα του οποίου η λύση θα χρησιμοποιηθεί πολλές φορές, τότε το κόστος της εκτέλεσης του προγράμματος θα ξεπεράσει κατά πολύ το κόστος ανάπτυξης και κωδικοποίησης, ειδικά όταν υπάρχουν μεγάλες ποσότητες δεδομένων εισόδου. Σε αυτή την περίπτωση είναι οικονομικά αποδεκτό να υλοποιηθεί ένας πιο πολύπλοκος αλγόριθμος, με την προϋπόθεση ότι ο πιο πολύπλοκος αλγόριθμος θα εκτελεστεί σημαντικά πιο γρήγορα από έναν πιο προφανή αλγόριθμο.

Ακόμη όμως και σε αυτές τις περιπτώσεις ίσως θα ήταν πιο λογικό και προτιμότερο να υλοποιήσουμε έναν απλό αλγόριθμο για να καθορίσουμε το πραγματικό κέρδος που θα είχαμε από την υλοποίηση ενός πιο πολύπλοκου προγράμματος. Κατά την δημιουργία ενός πολύπλοκου συστήματος είναι συχνά επιθυμητό να υλοποιήσουμε μία απλή έκδοση με οποία θα κάνουμε τις απαραίτητες μετρήσεις και προσομοιώσεις πριν προχωρήσουμε στην τελική πιο πολύπλοκη σχεδίαση. Οι προγραμματιστές δεν θα πρέπει να είναι γνώστες μόνο των τεχνικών και αλγορίθμων οι οποίοι βοηθούν στην γρήγορη εκτέλεση των προγραμμάτων, αλλά είναι βασικό να γνωρίζουν και πότε είναι χρήσιμο ή και απαραίτητο να διαθέσουν τον χρόνο για να τις υλοποιήσουν.

Εδώ ασχολούμαστε μόνο με αλγόριθμους οι οποίοι μπορούν να υλοποιηθούν από προγράμματα Η/Υ και κυρίως με την εκμάθηση διαθέσιμων τεχνικών και αλγορίθμων. Έτσι μία μέθοδος σύγκρισης δύο αλγορίθμων θα μπορούσε να είναι η υλοποίηση των αλγορίθμων με προγράμματα Η/Υ και κατόπιν η σύγκριση των χρόνων εκτέλεσής τους για διάφορα σύνολα δεδομένων εισόδου. Αν και αυτή η μέθοδος χρησιμοποιείται κάποιες φορές, συνήθως δεν είναι ικανοποιητική για διάφορους λόγους.

Ένας λόγος για τον οποίο η χρονική μέτρηση της εκτέλεσης των αλγορίθμων δεν είναι αρκετό μέτρο της αποτελεσματικότητας ενός αλγόριθμου είναι ότι αρκετοί άσχετοι παράγοντες μπορούν να επηρεάσουν τα αποτελέσματα. Αυτοί οι παράγοντες περιλαμβάνουν την γλώσσα προγραμματισμού με την οποία υλοποιείται ο αλγόριθμος, το είδος του Η/Υ, καθώς και η επιλογή του συνόλου των δεδομένων εισόδου. Οι αποφάσεις που παίρνουν οι προγραμματιστές και οι οποίες αφορούν την υλοποίηση, μπορούν επίσης να επηρεάσουν τα αποτελέσματα. Έτσι χρειαζόμαστε ένα μέτρο το οποίο θα εξαρτάται μόνο από τον αλγόριθμο και όχι από τις ιδιαιτερότητες της υλοποίησης.

Η μέθοδος που χρησιμοποιείται καταφεύγει στα μαθηματικά. Πρώτα εισάγουμε τις ιδέες της υλοποίησης και κατόπιν περιγράφουμε την ίδια την μέθοδο.

1.2. Παράδειγμα: Εύρεση Μέγιστης Τιμής

Υποθέστε το παρακάτω σενάριο. Δίνεται ένας χορός για κάποιο φιλανθρωπικό γεγονός. Η τιμή εισόδου δεν είναι προκαθορισμένη αλλά κάθε συμμετέχων είναι υποχρεωμένος να δωρίσει κάποιο ποσό ανάλογα με τις επιθυμίες του. Πως μπορούμε να βρούμε το μέγιστο ποσό το οποίο δωρίσθηκε;

Ένας αλγόριθμος για την επίλυση είναι ο εξής: Το ποσό της πρώτης δωρεάς είναι προφανώς το μεγαλύτερο μέχρι εκείνη την στιγμή. Όταν δίνεται η δεύτερη δωρεά, το ποσό της πρέπει να συγκριθεί με αυτό της πρώτης. Το μεγαλύτερο ποσό από τα δύο είναι η μεγαλύτερη δωρεά. Όταν δοθεί η τρίτη δωρεά, το ποσό της πρέπει να συγκριθεί με το ποσό της μεγαλύτερης από τις δύο πρώτες δωρεές για να καθοριστεί η μεγαλύτερη δωρεά μέχρι τώρα. Το ποσό κάθε επόμενης δωρεάς πρέπει να συγκρίνεται με το μεγαλύτερο ποσό όλων των προηγούμενων δωρεών έτσι ώστε να καθορίζεται η μεγαλύτερη δωρεά μέχρι την τρέχουσα. Όταν γίνει η σύγκριση για την τελευταία δωρεά, τότε καθορίζεται η μεγαλύτερη όλων των δωρεών.

Υποθέτουμε ότι τα ποσά δωρεών είναι διαθέσιμα σε ένα κατάλογο σταθερού μεγέθους ο οποίος έχει ήδη δημιουργηθεί. Το σχήμα 1 δίνει το κώδικα του αλγόριθμου εύρεσης της μέγιστης δωρεάς.

Η μέθοδος "`maxDonation`" υπολογίζει και επιστέφει την μέγιστη τιμή του πίνακα στον οποίο έχουν αποθηκευτεί οι δωρεές. Στην μέθοδο "`main`" ορίζεται η μεταβλητή "`maxDon`" η οποία γίνεται ίση με την μέγιστη δωρεά, τιμή που επιστρέφει η μέθοδος "`maxDonation`".

Ακόμη και για ένα τόσο μικρό και σχετικά απλό αλγόριθμο, το πόσες φορές θα εκτελεστεί η κάθε εντολή εξαρτάται από τις τιμές δεδομένων και την σειρά από την οποία δίνονται. Η

εντολή {1} θα εκτελεστεί μία μόνο φορά, ανεξάρτητα από τα δεδομένα, αλλά η εντολή επανάληψης {2} θα εκτελεστεί τόσες φορές όσα είναι τα δεδομένα μείον 1. Ακόμη και αν γνωρίζουμε το πλήθος των δεδομένων, δεν μπορούμε να γνωρίζουμε πόσες φορές θα εκτελεστεί η εντολή {2.1.1}.

Αυτή η εντολή μπορεί να μην εκτελεστεί ούτε μία φορά (αν η πρώτη τιμή τύχει να είναι και η μεγαλύτερη) ή μπορεί να εκτελεστεί για όλες τις τιμές (αν η διάταξη των τιμών είναι φθίνουσα).

```
class MaxDon {
   public static void main (String[] args) {
      float[] donArray = {4, 21, 9, 15, 7, 18, 27, 12};
      float maxDon = maxDonation(donArray);
      System.out.println("Donations:");

      for (int i=0; i<donArray.length; i++) {
         System.out.print(donArray[i]);
         if (donArray[i]==maxDon)
            System.out.println(" <== maximum donation");
         else
            System.out.println();
      }
   }

   static public float maxDonation (float[] donateArray) {
      int i;
      float max;
      max= donateArray[0];                    {1}
      for (i=1; i<donateArray.length; i++)    {2}
         if (max < donateArray[i])            {2.1}
            max = donateArray[i];             {2.1.1}
      return max;
   }
}
```

Σχήμα 1. Αλγόριθμος εύρεσης μέγιστης τιμής

Προτροπή 1:
Δώστε δύο ακολουθίες 10 τιμών για τον παραπάνω κώδικα. Στην πρώτη ακολουθία η εντολή {2.1.1} δεν θα πρέπει να εκτελείται ούτε μία φορά, ενώ στην δεύτερη θα πρέπει να εκτελείται 9 φορές.

Εάν γνωρίζουμε το πλήθος των τιμών μπορούμε να κάνουμε κάποιες υποθέσεις για το σύνολο των εντολών οι οποίες θα εκτελεστούν. Για παράδειγμα, εάν ο κατάλογος έχει 10 τιμές τότε μπορούμε να είμαστε σίγουροι ότι:

- Η εντολή {1} θα εκτελεστεί μία φορά
- Η εντολή {2} θα εκτελεστεί εννέα φορές
- Η εντολή {2.1} θα εκτελεστεί εννέα φορές
- Η εντολή {2.1.1} θα εκτελεστεί από 0 έως και εννέα φορές

Συνολικά θα εκτελεστούν τουλάχιστον 19 εντολές και το μέγιστο 28 εντολές.

Προτροπή 2:
Δώστε το πλήθος εκτέλεσης κάθε εντολής (αλλά και συνολικά) εάν ο κατάλογος είχε 1000 τιμές.

Γνωρίζοντας το πλήθος των δεδομένων μπορούμε να μιλήσουμε για τον ελάχιστο και τον μέγιστο αριθμό φορών που θα εκτελεστούν οι εντολές αλλά όχι για τον ακριβή αριθμό. Ο ακριβής αριθμός εξαρτάται από το πλήθος των δεδομένων αλλά και από την διάταξη των τιμών.

Ο αλγόριθμος εύρεσης της μέγιστης τιμής είναι ο ίδιος για οποιοδήποτε πλήθος "n" δεδομένων. Αναλύοντας τον αλγόριθμο για οποιοδήποτε πλήθος δεδομένων "n" έχουμε:

- Η εντολή {1} θα εκτελεστεί μία φορά
- Η εντολή {2} θα εκτελεστεί n-1 φορές
- Η εντολή {2.1} θα εκτελεστεί n-1 φορές
- Η εντολή {2.1.1} θα εκτελεστεί από 0 έως και n-1 φορές

Ο συνολικός αριθμός εντολών που θα εκτελεστούν θα είναι τουλάχιστον

`(1 + (n - 1) + (n - 1) + 0)` δηλαδή: `2n - 1` φορές

και το μέγιστο

`(1 + (n - 1) + (n - 1) + (n - 1))` δηλαδή: `3n - 2` φορές

Αυτό είναι και το μέτρο το οποίο ψάχνουμε. Γενικά θα αναζητούμε αντίστοιχο μέτρο σύγκρισης το οποίο θα αντιστοιχεί στα δεδομένα εισόδου.

Προτροπή 3:
Τροποποιείστε τον αλγόριθμο έτσι ώστε να βρίσκει την μικρότερη τιμή. Κατόπιν καθορίστε το πλήθος των εντολών οι οποίες θα εκτελεστούν. Ποιες είναι οι διαφορές από τον προηγούμενο αλγόριθμο εύρεσης της μεγαλύτερης τιμής.

2. Μέτρηση Πολυπλοκότητας

Όπως είδαμε, η γνώση του μεγέθους του προβλήματος δεν είναι αρκετή για να ορίσουμε τον ακριβή αριθμό βημάτων που χρειάζονται για την επίλυση του προβλήματος. Συνήθως συμβιβαζόμαστε με την απάντηση στις παρακάτω δύο ερωτήσεις:

- Ποια είναι η χειρότερη περίπτωση; Με άλλα λόγια, ποιος είναι ο μέγιστος αριθμός βημάτων που χρειάζονται για την επίλυση του προβλήματος;
- Ποιος είναι ο μέσος όρος; Με άλλα λόγια, εάν χρησιμοποιήσουμε τον αλγόριθμο για να λύσουμε πολλά προβλήματα ίδιου μεγέθους αλλά με διαφορετικά δεδομένα, ποιος είναι ο μέσος όρος του πλήθους των βημάτων τα οποία χρειάζονται σε κάθε επίλυση του προβλήματος;

Η απάντηση που ψάχνουμε είναι κάποιος τύπος ή κανόνας ο οποίος θα μας δίνει τον μέγιστο αριθμό (ή τον μέσο όρο) βημάτων για κάποιο μέγεθος n ενός προβλήματος. Ένας τέτοιος κανόνας αποδίδεται από μία συνάρτηση. Για παράδειγμα η συνάρτηση η οποία μας δίνει το πλήθος των βημάτων για την χειρότερη περίπτωση του αλγόριθμου εύρεσης της μέγιστης τιμής είναι:

$$\text{Χειρότερη\_περίπτωση\_μέγιστου}(n) = 3n - 2$$

Εάν δεν μπορούμε να βρούμε έναν τύπο ο οποίος θα δίνει τον ακριβή μέγιστο αριθμό βημάτων που χρειάζονται, τότε ίσως πρέπει να συμβιβαστούμε με μια καλή προσέγγιση.

Είναι ακόμη πιο δύσκολο να βρούμε μια συνάρτηση η οποία θα περιγράφει την συμπεριφορά του μέσου όρου των περιπτώσεων.

3. Χρόνος Εκτέλεσης

Τελικός σκοπός μας είναι κάποιο μέτρο του χρόνου ο οποίος απαιτείται για την επίλυση ενός προβλήματος. Δυστυχώς, η γνώση του πλήθους των βημάτων τα οποία θα εκτελεστούν δεν μας λέει και την διάρκεια εκτέλεσής τους. Διάφορα βήματα συνήθως απαιτούν και διαφορετικό χρόνο εκτέλεσης. Ένα μόνο βήμα σε κάποιον αλγόριθμο, μπορεί σε έναν άλλο αλγόριθμο επίλυσης του ίδιου προβλήματος να εκφραστεί με περισσότερα βήματα.

Για παράδειγμα στην μέθοδο εύρεσης του μέγιστου, αντί για την δομή επανάληψης “for” χρησιμοποιούμε την δομή “while” και προκύπτει ο κώδικας του σχήματος 2. Η μέθοδος “maxDonation2” επίσης υπολογίζει και επιστρέφει το μέγιστο ενός πίνακα. Μία ανάλυση του αλγόριθμου αυτού (για n τιμές επίσης) δίνει τα παρακάτω αποτελέσματα:

- Η εντολή {1} θα εκτελεστεί μία φορά
- Η εντολή {2} θα εκτελεστεί μία φορές
- Η εντολή {3} θα εκτελεστεί n φορές
- Η εντολή {3.1} θα εκτελεστεί n-1 φορές
- Η εντολή {3.1.1} θα εκτελεστεί από 0 έως και n-1 φορές
- Η εντολή {3.2} θα εκτελεστεί n-1 φορές

```
public static float maxDonation2 (float[] donateArray) {
   float max = donateArray[0];          {1}
   int i = 1;                           {2}
   while (i < donateArray.length) {     {3}
      if (max < donateArray[i])         {3.1}
         max = donateArray[i];          {3.1.1}
         i = i+1; {3.2}
   }
   return max;
}
```

Σχήμα 2. Δεύτερος αλγόριθμος εύρεσης μέγιστης τιμής

Για παράδειγμα ο κώδικας της μεθόδου "maxDonation2" ο οποίος επίσης βρίσκει την μέγιστη τιμή. Μία ανάλυση του αλγόριθμου αυτού (για n τιμές επίσης) δίνει τα παρακάτω αποτελέσματα:

- Η εντολή {1} θα εκτελεστεί μία φορά
- Η εντολή {2} θα εκτελεστεί μία φορές
- Η εντολή {3} θα εκτελεστεί n φορές
- Η εντολή {3.1} θα εκτελεστεί n-1 φορές
- Η εντολή {3.1.1} θα εκτελεστεί από 0 έως και n-1 φορές
- Η εντολή {3.2} θα εκτελεστεί n-1 φορές

Σε αυτή την έκδοση του αλγόριθμου θα εκτελεστούν τουλάχιστον 3n εντολές και το πολύ 4n-1 εντολές. Ο μεγαλύτερος αριθμός εντολών δεν σημαίνει ότι αυτός ο αλγόριθμος είναι λιγότερο αποτελεσματικός αφού κάθε εντολή μπορεί να εκτελείται γρηγορότερα.

Τι γίνεται όμως με τον χρόνο εκτέλεσης του αλγόριθμου; Εξετάζοντας τα βήματα του αλγόριθμου βλέπουμε ότι ο χρόνος εκτέλεσης του κάθε βήματος είναι ανεξάρτητος του n. Αφού ο χρόνος εκτέλεσης κάθε βήματος είναι σταθερός, ο συνολικός χρόνος εκτέλεσης του αλγόριθμου είναι χονδρικά ανάλογος του n. Έτσι όποιον αλγόριθμο από τους δύο και να χρησιμοποιήσουμε, η εύρεση του μέγιστου 5000 αριθμών θα χρειαστεί περίπου πενταπλάσιο χρόνο από την εύρεση του μέγιστου 1000 αριθμών.

Από αυτή την ανάλυση βλέπουμε πως ποικίλει ο χρόνος εκτέλεσης ενός αλγόριθμου όταν ποικίλει και το μέγεθος του προβλήματος. Έτσι από την συμπεριφορά ενός αλγόριθμου σε μικρά προβλήματα μπορούμε χονδρικά να προβλέψουμε την συμπεριφορά του σε μεγάλα προβλήματα.

4. Αλγόριθμοι Αναζήτησης (Searching Algorithms)

4.1. Σειριακή Αναζήτηση (Sequential Search)

Για να αναζητήσουμε ένα στοιχείο σε μία λίστα στοιχείων (πχ. σε έαν πίνακα), συγκρίνουμε το στοιχείο το οποίο αναζητούμε με διαδοχικά στοιχεία της λίστας, ξεκινώντας από το πρώτο στοιχείο της λίστας. Σταματάμε είτε όταν βρεθεί το προς αναζήτηση στοιχείο, είτε όταν φτάσουμε στο τέλος της λίστας. Εάν η λίστα είναι ταξινομημένη μπορούμε να σταματήσουμε όταν φτάσουμε σε κάποιο στοιχείο το οποίο έχει τιμή μεγαλύτερη από το προς αναζήτηση στοιχείο.

Ο κώδικας του σχήματος 3, της μεθόδου "`SeqSearch`" δείχνει την υλοποίηση μιας μεθόδου σειριακής αναζήτησης (sequential search) η οποία αναζητά το στοιχείο "`key`" σε ένα μη ταξινομημένο πίνακα "`numbers`".

```
public class SeqSearch {
   public static void main (String[] args) {
      int numbers[] = {7, -3, 7, 2, 8, -1, 3, 2, 5, 6, 7};
      System.out.println("The index of 6 is " +
      LinearSearch.search (numbers, 6));
   } // End of main
} // End of class SeqSearch

class LinearSearch {
   public static int search (int[] a, int key) {
      for (int i=0; i<a.length; i++)              {1}
         if (a[i] == key)                         {1.1}
            return i;                             {1.1.1}
      return -1;                                  {2}
   }
}
```

Σχήμα 3. Αλγόριθμος σειριακής αναζήτησης

Η ανάλυση πολυπλοκότητας του αλγόριθμου σειριακής αναζήτησης για `n` δεδομένα έχει τα εξής αποτελέσματα:

- Η εντολή {1} εκτελείται τουλάχιστον μία και το πολύ n+1 φορές
- Η εντολή {1.1} εκτελείται κάθε φορά που εκτελείται η {1} εκτός της τελευταίας.

- Η εντολή {1.1.1} εκτελείται καμία ή μία το πολύ φορά
- Η εντολή {2} εκτελείται καμία ή μία το πολύ φορά

Έστω T το πλήθος των φορών που εκτελείται η {1.1}. Τότε η {1} εκτελείται T+1 φορές και οι εντολές {1.1.1} και {2} εκτελούνται καμία ή μία το πολύ φορά, αλλά μία από τις δύο θα εκτελεστεί οπωσδήποτε. Το σύνολο των εντολών που εκτελούνται είναι (T + 1) + T +1. Επειδή η τιμή του T είναι μεταξύ 0 και n το σύνολο των εντολών που θα εκτελεστούν είναι τουλάχιστον 2 (όταν T=0) και το πολύ 2+2n (όταν T=n).

Για να βρούμε κάποιο κανόνα για το μέσο όρο των εντολών που θα εκτελεστούν, κάνουμε κάποιες λογικές παραδοχές. Επειδή δεν μπορούμε να ξέρουμε τίποτε για το κατά πόσο το προς αναζήτηση στοιχείο συμπεριλαμβάνεται στον πίνακα, δεν εξετάζουμε καθόλου αυτό το θέμα και υπολογίζουμε τον μέσο όρο μόνο για την περίπτωση που το στοιχείο υπάρχει στον πίνακα. Επίσης θα υποθέσουμε ότι το στοιχείο έχει ίση πιθανότητα να βρεθεί σε οποιαδήποτε θέση του πίνακα. Έτσι ο αριθμός που η {1.1} θα εκτελεστεί μπορεί να είναι 1, ή 2, ή 3, ..., ή n, όλες με ίση πιθανότητα. Έτσι ο μέσος όρος των αριθμών εκτέλεσης της {1.1} είναι:

$$\frac{1+2+3+...+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

Υπολογίζοντας και τον αριθμό εκτέλεσης των υπολοίπων εντολών, προκύπτει ο παρακάτω μέσος όρος για το σύνολο του αλγόριθμου σειριακής αναζήτησης:

$$\left(\frac{n+1}{2}+1\right)+\frac{n+1}{2}+1 = n+3$$

Έτσι ο χρόνος αναζήτησης μιας τιμής σε μη ταξινομημένο πίνακα είναι χονδρικά ανάλογος του μεγέθους του πίνακα

Προτροπή 4:

Τροποποιείστε τον αλγόριθμο σειριακής αναζήτησης έτσι ώστε να αποκτήσει δυνατότητα αναζήτησης σε ένα μόνο τμήμα του πίνακα. Η αρχή και το τέλος αναζήτησης δίνονται ως παράμετροι της μεθόδου "search". Κατόπιν καθορίστε το πλήθος των εντολών οι οποίες θα εκτελεστούν. Ποιες είναι οι διαφορές από τον προηγούμενο αλγόριθμο σειριακής αναζήτησης;

4.2. Δυαδική Αναζήτηση (Binary Search)

Ο αλγόριθμος της δυαδικής αναζήτησης (binary search) εφαρμόζεται **μόνο** σε ταξινομημένα στοιχεία. Αν τα στοιχεία δεν είναι ταξινομημένα τότε **δεν** μπορεί να εφαρμοστεί.

Εδώ, δεν συγκρίνουμε διαδοχικά κάθε στοιχείο του πίνακα με το προς αναζήτηση στοιχείο. Ο αλγόριθμος λειτουργεί ως εξής:

Βρίσκουμε το μεσαίο στοιχείο του ταξινομημένου πίνακα. Εάν το προς αναζήτηση στοιχείο είναι ίσο με το μεσαίο στοιχείο τότε σταματάμε την αναζήτηση αφού το στοιχείο βρέθηκε.

Εάν δεν βρέθηκε, τότε ελέγχουμε αν το στοιχείο που αναζητούμε είναι μικρότερο ή μεγαλύτερο από το μεσαίο στοιχείο του πίνακα. Αν είναι μικρότερο, περιορίζουμε την αναζήτηση στο πρώτο μισό του πίνακα (με την προϋπόθεση ότι τα στοιχεία είναι διατεταγμένα κατά αύξουσα σειρά), ενώ αν είναι μεγαλύτερο περιορίζουμε την αναζήτηση στο δεύτερο μισό του πίνακα.

Η διαδικασία αυτή λοιπόν επαναλαμβάνεται για το κατάλληλο πρώτο ή δεύτερο μισό πίνακα, μετά για το 1/4 του πίνακα κ.ο.κ. μέχρι είτε να βρεθεί το στοιχείο, ή να μην είναι δυνατό να χωρισθεί ο πίνακας περαιτέρω σε δύο νέα μέρη.

Το σχήμα 4 δείχνει την κλάση "`BinarySearch`" η οποία περιλαμβάνει την υλοποίηση μιας μεθόδου δυαδικής αναζήτησης ("`binSearch`") η οποία αναζητά το στοιχείο "`key`" σε ένα ταξινομημένο πίνακα "`numbers`".

Ξεκινάμε την ανάλυση πολυπλοκότητας του αλγόριθμου δυαδικής αναζήτησης για n δεδομένα με την απλή παρατήρηση ότι οι εντολές {1} και {3} εκτελούνται μία φορά. Το κλειδί της ανάλυσης είναι πόσες φορές εκτελείται η εντολή {2} η οποία είναι και η μόνη εντολή επανάληψης. Κάθε φορά που εκτελείται η {2} έχουμε:

- Η εντολή {2.1} εκτελείται ακριβώς μία φορά
- Η εντολή {2.2} εκτελείται ακριβώς μία φορά
- Η εντολή {2.2.1} εκτελείται μία ή καμία φορά
- Η εντολή {2.3} εκτελείται μία ή καμία φορά
- Η εντολή {2.3.1} εκτελείται μία ή καμία φορά
- Η εντολή {2.3.2} εκτελείται μία ή καμία φορά

```
public class TestBinSearch {
   public static void main (String[] args) {
      int[] numbers = {1, 3, 5, 7, 8, 9, 11, 15, 22, 33, 44};
      int key=11;
      int pos = BinarySearch.search (numbers, key);
      if (pos != -1)
         System.out.println ("The index of "+key+" is " + pos);
      else
         System.out.println (key + " is not part of the array");
   }
}

public class BinarySearch {
   public static int search (int[] numbers, int key) {
      int left=0, right=numbers.length-1;
      return binSearch(numbers, key, left, right);
   }
```

```
   private static int binSearch (int[] numbers, int key, int
                                          left, int right) {
      int mid, pos=-1;                          {1}
      while (pos==-1 && left<=right) {          {2}
         mid = (left+right)/2;                  {2.1}
         if (key < numbers[mid])                {2.2}
            right = mid-1;                      {2.2.1}
         else
            if (key > numbers[mid])             {2.3}
               left = mid+1;                    {2.3.1}
            else
               pos = mid;                       {2.3.2}
      }
      return pos;                               {3}
   }
}
```

Σχήμα 4. Αλγόριθμος δυαδικής αναζήτησης

Η εκτέλεση κάποιων από αυτές τις εντολές απαγορεύει την εκτέλεση κάποιων άλλων. Έτσι η χειρότερη περίπτωση είναι η εκτέλεση τεσσάρων εντολών σε κάθε εκτέλεση της εντολής επανάληψης {2} και αυτές είναι: {2.1},{2.2},{2.3}, και {2.3.1} ή {2.3.2}. Έτσι αν μπορούμε να μετρήσουμε στην χειρότερη περίπτωση τις περισσότερες φορές που εκτελείται η {2}, τότε το πολ/ζουμε επί 4 και προσθέτουμε 2.

Η χειρότερη περίπτωση του αλγόριθμου δυαδικής αναζήτησης είναι όταν δεν βρεθεί το προς αναζήτηση στοιχείο.

Έστω ότι υπάρχουν n στοιχεία. Μετά την πρώτη επανάληψη τα στοιχεία που απομένουν είναι το πολύ $n/2$. Σε κάθε διαδοχική επανάληψη τουλάχιστον τα μισά στοιχεία αφαιρούνται.

Έτσι φτάνουμε στην ερώτηση: Δίνεται ένας αριθμός *n*. Σε μια επαναληπτική διαδικασία διαιρούμε τον αριθμό n με το 2 και παίρνουμε το αποτέλεσμα με το οποίο συνεχίζουμε την διαίρεση. Πόσες φορές πρέπει να διαιρέσουμε με το 2 για να έχουμε τελικό αποτέλεσμα 0; Ο αριθμός αυτός είναι $L(n)$ το οποίο είναι ο λογάριθμος του *n* με βάση το 2. Παρακάτω δίνονται ενδεικτικά μερικές τιμές του L(*n*) (διαιρέσεις που πρέπει να γίνουν) για κάποιο πλήθος στοιχείων n:

n	L(n)
10	4
100	7
1.000	10
10.000	14
1.000.000	20

Ολοκληρώνοντας την ανάλυση του αλγορίθμου δυαδικής αναζήτησης και αφού ο αριθμός επανάληψης της {2} είναι L(n), τότε ο συνολικός αριθμός εντολών είναι 4L(n)+2. Έτσι αναζητώντας ένα στοιχείο σε μία λίστα με ένα εκατομμύριο στοιχεία, δεν χρειάζεται να

περάσουμε πάνω από 20 φορές την εντολή επανάληψης και θα είχαμε να εκτελέσουμε το πολύ 82 εντολές. Σε σύγκριση ο αλγόριθμος σειριακής αναζήτησης θα χρησιμοποιούσε 1.000.003 εντολές.

4.3. Αναδρομική Δυαδική Αναζήτηση (Recursive Binary Search)

Όπως αναφέρθηκε παραπάνω η δυαδική αναζήτηση αφού ελέγξει το μεσαίο στοιχείο, επαναλαμβάνει την διαδικασία για το πρώτο ή το δεύτερο μισό του πίνακα. Αυτή η επανάληψη κάνει την δυαδική αναζήτηση ιδανική υποψήφια για αναδρομική υλοποίηση.

```
public class RecBinSearch {
   public static void main (String[] args) {
      int[] numbers = {2, 3, 5, 7, 8, 9, 11, 15, 19, 22, 33};
      int key=7;
      int pos = RecursiveBinSearch.search (numbers, key);
      if (pos != -1)
         System.out.println ("The index of "+key+" is "+pos);
      else
         System.out.println (key+" is not part of the array");
   }
}

class RecursiveBinSearch {
   public static int search (int[] numbers, int key) {
      int left=0, right=numbers.length-1;
      return recSearch(numbers, key, left, right);
   }

   private static int recSearch (int[] numbers, int key, int
                                                    left,    int
                                                    right) {
      int mid=(left+right)/2;
      if (left>right)
         return -1;
      else
         if (key == numbers[mid])
            return mid;
         else
            if (key>numbers[mid])
               return recSearch (numbers, key, mid+1, right);
            else
               return recSearch (numbers, key, left, mid-1);
   }
}
```

Σχήμα 5. Αλγόριθμος αναδρομικής δυαδικής αναζήτησης

Ο κώδικας του σχήματος 5, δείχνει την αναδρομική υλοποίηση μιας μεθόδου δυαδικής αναζήτησης (“`recSearch`”) η οποία αναζητά το στοιχείο “key” σε ένα ταξινομημένο πίνακα “`numbers`”.

Η μέθοδος “`recSearch`” χωρίζει τον πίνακα σε δύο μέρη και μετά καλεί τον εαυτό της για τα δύο μέρη στα οποία χωρίζεται ο πίνακας. Η διαδικασία αυτή συνεχίζεται μέχρι είτε να βρεθεί το προς αναζήτηση στοιχείο (στο μέσον κάποιου υπο-πίνακα), ή να μην είναι πλέον δυνατός ο χωρισμός σε δύο υπο-πίνακες.

5. Πίνακες Κατακερματισμού (Hash Tables)

Εάν όλα τα στοιχεία τα οποία ψάχνουμε ήταν μικροί ακέραιοι πχ. στην περιοχή 1 έως n τότε η αναζήτηση θα ήταν σχετικά απλή διαδικασία. Θα αρκούσε να ορίσουμε ένα πίνακα με δείκτες τους ακέραιους δηλ. τα στοιχεία που αναζητούμε (“κλειδιά”).

Τις περισσότερες φορές όμως οι δείκτες των πινάκων δεν μπορούν να χρησιμοποιηθούν και ως αναγνωριστικά των στοιχείων που έχουν αποθηκευτεί. Αυτό μπορεί να οφείλεται στο μεγάλο πλήθος στοιχείων ή στο ότι τα κλειδιά (στοιχεία προς αναζήτηση) δεν είναι ακέραιοι αριθμοί.

Παρόλα αυτά η ιδέα της αντιστοίχησης των κλειδιών με τους δείκτες του πίνακα είναι ιδιαίτερα ελκυστική επειδή προσφέρει δυνατότητες γρήγορης εισαγωγής και αναζήτησης. Για να μπορέσουμε να υλοποιήσουμε μια τέτοια τεχνική δηλαδή να χρησιμοποιούμε το κλειδί σαν δείκτη του πίνακα, θα πρέπει με κάποιο τρόπο να αντιστοιχίσουμε τα κλειδιά σε μοναδικούς ακέραιους τους οποίους θα χρησιμοποιήσουμε ως δείκτες του πίνακα. Μία συνάρτηση η οποία κάνει αυτή την αντιστοίχηση ονομάζεται συνάρτηση κατακερματισμού (hash function), ενώ η διαδικασία μετατροπής ονομάζεται κατακερματισμός (hashing) και η δομή αποθήκευσης ονομάζεται πίνακας κατακερματισμού (hash table).

Ένας **πίνακας κατακερματισμού** (**`hash table`** ή **`hash map`**) είναι μία δομή δεδομένων η οποία συσχετίζει κλειδιά με τιμές. Για παράδειγμα, θα μπορούσε να γίνει ένας πίνακας που αντιστοιχίζει ονόματα ανθρώπων με τα τηλέφωνά τους.

Όπως θα δούμε παρακάτω κατά την διαδικασία κατακερματισμού μπορεί δύο κλειδιά να αντιστοιχηθούν στον ίδιο δείκτη. Αυτό καλείται σύγκρουση (collision). Οι συγκρούσεις είναι αναπόφευκτες εάν έχουμε σαν στόχο ένα πίνακα ο οποίος θα έχει λογικό μέγεθος. Έτσι για την αποτελεσματική λειτουργία της τεχνικής αυτής πρέπει να ορίσουμε τα εξής:

- Επιλογή κατάλληλης συνάρτησης κατακερματισμού
- Στρατηγική επίλυσης των συγκρούσεων (collision resolution strategy)

5.1. Συναρτήσεις κατακερματισμού (Hash Functions)

Μία καλή συνάρτηση κατακερματισμού $H(k)$ αντιστοιχεί τα κλειδιά σε ένα έγκυρο πίνακα δεικτών. Υπάρχουν αρκετές συναρτήσεις κατακερματισμού οι οποίες έχουν μελετηθεί και κατανοηθεί η λειτουργία και οι ιδιότητές τους. Μία επιθυμητή ιδιότητα των συναρτήσεων αυτών είναι να μπορούν να αντιστοιχούν δύο κλειδιά τα οποία είναι πολύ κοντά σε δύο καθαρά διακριτές τιμές.

Οι συναρτήσεις κατακερματισμού μπορεί να είναι αρκετά σύνθετες, αν και μερικές απλές έχουν πολύ καλά αποτελέσματα. Οι απλές συναρτήσεις έχουν και ένα επιπλέον πλεονέκτημα: είναι πιο γρήγορες. Μία τέτοια απλή συνάρτηση με πολύ καλά αποτελέσματα είναι η εύρεση του υπολοίπου ακεραίας διαίρεσης.

$$H(k) = k \ \% \ m$$

όπου m είναι το μέγεθος του πίνακα. (Έχει αποδειχθεί ότι αυτή η συνάρτηση είναι ιδιαίτερα αποτελεσματική όταν το m είναι πρώτος αριθμός).

Θεωρείστε τα εξής στοιχεία:

olive	carrot	cheese	ham
onion	tomatopaste	salt	mushroom

Μπορεί να δημιουργηθεί ένα ενδιάμεσο κλειδί για κάθε λέξη. Το κλειδί αυτό είναι το άθροισμα των θέσεων στο αλφάβητο, των γραμμάτων που αποτελούν την κάθε λέξη.

Στοιχείο	*Άθροισμα*	*Στοιχείο*	*Άθροισμα*
olive	63 (15+12+9+22+5)	carrot	75 (3+1+18+18+15+20)
cheese	45 (3+8+5+5+9+5)	ham	22 (8+1+13)
onion	67 (15+14+9+15+14)	tomatopaste	145 (20+15+13+1+20+15+16+1+19+20+5)
salt	52 (19+1+12+20)	mushroom	122 (13+21+19+8+18+15+15+13)

Το πλήθος των πιθανών συστατικών που μπορούν να χρησιμοποιηθούν μπορεί να είναι μερικές χιλιάδες, με την παραπάνω όμως διαδικασία αντιστοίχησης η μέγιστη τιμή είναι μόνο 260 όταν το σύνολο των χαρακτήρων είναι 10. Χρησιμοποιώντας το υπόλοιπο της ακεραίας διαίρεσης μπορούμε να ελαττώσουμε το μέγεθος του πίνακα κατακερματισμού ακόμη περισσότερο. Για ένα πίνακα με 13 (πρώτος αριθμός) υλικά έχουμε:

Στοιχείο	*H(στοιχείο) Άθροισμα*	*Υπόλοιπο (Άθροισμα%13)*	*Στοιχείο*	*H(στοιχείο) Άθροισμα*	*Υπόλοιπο (Άθροισμα%13)*
olive	63	11 (63%13)	carrot	75	10 (75%13)
cheese	45	6 (45%13)	ham	22	9 (22%13)
onion	67	2 (67%13)	tomatopaste	145	2 (145%13)
salt	52	0 (%13)	mushroom	122	5 (122%13)

5.2. Επίλυση Συγκρούσεων (Collision Resolution)

Κατά την εφαρμογή της συνάρτησης κατακερματισμού στο έβδομο στοιχείο (tomatopaste) βλέπουμε να δημιουργείτε ένα πρόβλημα: αντιστοιχίζεται στην ίδια θέση με το τρίτο (onion). Το πρόβλημα αυτό μπορεί να δημιουργηθεί και από την εισαγωγή ενός νέου στοιχείου. Για παράδειγμα εισάγοντας το στοιχείο "bean" (H[bean]=22 -→ 9 (22%13)) βλέπουμε ότι αντιστοιχίζεται με το προ-υπάρχον στοιχείο "ham", δηλ. πρέπει να αποθηκευτεί στην ίδια θέση που έχει ήδη αποθηκευτεί το στοιχείο "ham". Η αντιμετώπιση και επίλυση αυτών των συγκρούσεων (collision resolution) είναι το δεύτερο σημαντικό στοιχείο των τεχνικών κατακερματισμού.

Μία απλή μέθοδος αντιμετώπισης αυτών των συγκρούσεων είναι η γραμμική διερεύνηση (linear probing). Σύμφωνα με αυτή την μέθοδο, εάν κατά την εισαγωγή ενός κλειδιού υπάρχει σύγκρουση (η θέση του πίνακα είναι ήδη κατειλημμένη) τότε εξετάζεται η επόμενη θέση μέχρι να βρεθεί κάποια ελεύθερη θέση ή να εξαντληθούν διαθέσιμες θέσεις.

Το αποτέλεσμα εφαρμογής της γραμμικής διερεύνησης στα παραπάνω υλικά για ένα πίνακα 13 θέσεων φαίνεται στο σχήμα 6.

H[11] ← olive
H[6] ← cheese
H[2] ← onion
H[0] ← salt
H[10] ← carrot
H[9] ← ham
H[2] ← Σύγκρουση
H[3] ← tomatopaste
H[5] ← mushroom
H[9] ← Σύγκρουση
H[10] ← Σύγκρουση
H[11] ← Σύγκρουση
H[12] ← bean

0	salt
1	
2	onion
3	tomatopaste
4	
5	mushroom
6	cheese
7	
8	
9	ham
10	carrot
11	olive

Σχήμα 6. Πίνακας κατακερματισμού μετά την επίλυση των συγκρούσεων

Αναλυτικά: Το πρώτο στοιχείο "olive" αντιστοιχεί στην θέση με δείκτη 11 (η 12η θέση του πίνακα), όπου και τοποθετείται. Το ίδιο συμβαίνει και με τα πέντε επόμενα στοιχεία τα οποία τοποθετούνται στις θέσεις 6, 2, 0, 10 και 9 αντίστοιχα. Η πρώτη σύγκρουση έρχεται με την εισαγωγή του έβδομου στοιχείου "tomatopaste" το οποίο έχει δείκτη 2. Η θέση αυτή είναι κατειλημμένη από το στοιχείο "onion" και έτσι εξετάζεται η διαθεσιμότητα της επόμενης

θέσης. Η επόμενη θέση (3) είναι ελεύθερη και έτσι το στοιχείο τοποθετείται εκεί. Το στοιχείο "mushroom" τοποθετείται στην 5η θέση χωρίς πρόβλημα. Η εισαγωγή του στοιχείου "bean" στην θέση 9 δημιουργεί σύγκρουση με το στοιχείο "ham" και έτσι εξετάζεται η διαθεσιμότητα της επόμενης θέσης. Η επόμενη θέση είναι κατειλημμένη από το "carrot" η μεθεπόμενη από το "olive". Έτσι το στοιχείο "bean" τελικά τοποθετείται στη θέση 12.

5.2. Java και Πίνακες Κατακερματισμού

Η Java διαθέτει ενσωματωμένες συναρτήσεις κατακερματισμού οι οποίες κάνουν απλή την χρήση των τεχνικών κατακερματισμού, αφού κατανοήσετε την λειτουργία τους.

Βασική έννοια είναι ότι υπολογίζεται μία ακέραια τιμή, για κάθε τύπο δεδομένων, η οποία καθορίζει την θέση του πίνακα κατακερματισμού στην οποία θα αποθηκευτεί. Πιο συγκεκριμένα η κλάση Object περιέχει την μέθοδο hashCode() η οποία χρησιμοποιεί την τιμή του αντικειμένου για να υπολογίσει τον κωδικό κατακερματισμού του. Αυτή η μέθοδος κληρονομείται από όλα τα αντικείμενα.

Επιπρόσθετα, το πακέτο java.util περιέχει την κλάση Hashtable η οποία υλοποιεί γενικές τεχνικές κατακερματισμού αντιστοιχίζοντας κλειδιά σε τιμές. Οποιοδήποτε αντικείμενο μπορεί να χρησιμοποιηθεί ως κλειδί ή ως τιμή. Για την επιτυχή αποθήκευση και ανάκληση αντικειμένων κάποιας κλάσης με ένα πίνακα κατακερματισμού, τα αντικείμενα που χρησιμοποιούνται ως κλειδιά πρέπει να υλοποιούν τις μεθόδους `hashCode` και `equals`.

Ένα αντικείμενο `Hashtable` έχει δύο παραμέτρους που επηρεάζουν την λειτουργία του: αρχική χωρητικότητα (*initial capacity*) και παράγοντας φόρτου (*load factor*). Η αρχική χωρητικότητα ορίζει το πλήθος των στοιχείων που μπορεί να αποθηκευτεί στον πίνακα κατά την στιγμή της δημιουργίας του. Ο παράγοντας φόρτου είναι ένα μέτρο που δείχνει πόσο επιτρέπεται να γεμίσει ο πίνακας πριν αυξηθεί αυτόματα η χωρητικότητά του. Όταν οι εισαγωγές στον πίνακα κατακερματισμού ξεπεράσουν το γινόμενο της τρέχουσας χωρητικότητας επί τον παράγοντα φόρτου, η χωρητικότητα αυξάνεται καλώντας της μέθοδο `rehash`.

Η δημόσια (public) κλάση Hashtable διαθέτει τους δομητές:

- public Hashtable()
- public Hashtable(int size)
- public Hashtable(int size, float load) throws IllegalArgumentException

οι οποίοι δημιουργούν ένα πίνακα κατακερματισμού, θέτοντας την αρχική του χωρητικότητα (ή αφήνοντας το προκαθορισμένο μέγεθος που είναι 101) και ένα παράγοντα φόρτου (προκαθορισμένο 0.75).

Μερικές από τις μεθόδους που διαθέτει η κλάση Hashtable είναι οι εξής:

- public void clear()
  Αφαιρεί όλα τα στοιχεία από το πίνακα κατακερματισμού

- public boolean contains(Object arg) throws NullPointerException
  Επιστρέφει την τιμή true εάν ο πίνακας κατακερματισμού περιέχει το στοιχείο arg
- public boolean containsKey(Object index)
  Επιστρέφει την τιμή true εάν ο πίνακας κατακερματισμού περιέχει μία εισαγωγή για το κλειδί στην θέση index.
- public Object get(Object index)
- public Object put(Object index, Object arg) throws NullPointerException
- public Object remove(Object index)
  Επιστρέφει, εισάγει ή διαγράφει το στοιχείο arg το οποίο αντιστοιχεί στο κλειδί στην θέση index.
- public boolean isEmpty()
  Επιστρέφει την τιμή true εάν ο πίνακας κατακερματισμού είναι άδειος.
- protected void rehash()
  Μεταβάλλει το μέγεθος του πίνακα κατακερματισμού. Καλείται αυτόματα όταν το πλήθος των κλειδιών ξεπεράσει την χωρητικότητα και τον παράγοντα φόρτου.
- public int size()
  Επιστρέφει το πλήθος των στοιχείων που έχει ο πίνακας κατακερματισμού.
- public String toString()
  Επιστρέφει μία αναπαράσταση των ζευγών κλειδί-στοιχείο σε μορφή string.