ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

Τμήμα Μηχανικών Πληροφορικής και
Ηλεκτρονικών Συστημάτων

Αντικειμενοστρεφής Προγραμματισμός (Object Oriented Programming)

Πολυπλοκότητα - Εισαγωγή στους Αλγόριθμους Αναζήτησης

(Complexity Analysis – Searching Algorithms)

Παναγιώτης Σφέτσος, PhD
http://aetos.it.teithe.gr/~sfetsos/
sfetsos@it.teithe.gr

Αλγόριθμοι Ταξινόμησης

Στην ενότητα αυτή θα μελετηθούν οι παρακάτω αλγόριθμοι ταξινόμησης:

- **Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων**
- **Σειριακή ή γραμμική αναζήτηση** *(Sequential Search)*
- **Δυαδική αναζήτηση** *(Binary Search)*
- **Κατακερματισμός**
- **Java και Κατακερματισμός**

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Πρόβλημα ⟶ ***Αλγόριθμος1*** ⟶ ***υλοποίηση1 (instance1)***

⟶ ***Αλγόριθμος2*** ⟶ ***υλοποίηση2 (instance2)***

: : : :

Ο όρος **αλγόριθμος** *(algorithm)* χρησιμοποιείται για να περιγράψει μία *βήμα-προς-βήμα*, πεπερασμένη *(finite)*, αποτελεσματική *(affective)* και κατάλληλη για υλοποίηση *(implementation)* διαδικασία επίλυσης του προβλήματος.

- Για κάθε *(επιλύσιμο)* πρόβλημα **Π** υπάρχει ένα σύνολο αλγορίθμων **{A_1, A_2, …A_k}** που το επιλύουν. Με άλλα λόγια, για το πρόβλημα **Π** υπάρχει αλγόριθμος, A_i, 1 <= i <= k, τέτοιος ώστε κάθε στιγμιότυπο **i** του Π ή ισοδύναμα, για κάθε είσοδο **i** ο αλγόριθμος A_i δίδει σωστό αποτέλεσμα.

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Ένας αλγόριθμος **πρέπει να είναι:**

- **σωστός**, για οποιοδήποτε σύνολο δεδομένων εισόδου, δηλ. για κάθε υλοποίηση του πεδίου ορισμού του προβλήματος που λύνει.
- **αποδοτικός,** δηλ. (α) εκτελείται γρήγορα (λιγότερα βήματα εκτέλεσης) και (β) εκμεταλλεύεται αποτελεσματικά τους πόρους του συστήματος.

➢ **Συχνά οι δύο στόχοι είναι αντικρουόμενοι.**

Στόχος:

(1) η ανάλυση και ο υπολογισμός *της* ***πολυπλοκότητας χρόνου και χώρου*** *(time and space complexity)* των αλγορίθμων και

(2) ο έλεγχος αν και πότε ένας αλγόριθμος είναι ***άριστος (optimal),*** δηλαδή ο πιο αποδοτικός για το πρόβλημα για το οποίο σχεδιάστηκε.

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Μέτρηση της επίδοσης ενός αλγορίθμου:

Εμπειρική Πολυπλοκότητα: ο αλγόριθμος υλοποιείται σε μια γλώσσα προγραμματισμού και εφαρμόζεται σε ένα σύνολο δεδομένων (διάφορα μεγέθη εισόδων), για να ***υπολογισθεί ο απαιτούμενος χρόνος επεξεργασίας*** *(processing time)* και η **χωρητικότητα μνήμης** *(memory space).*

Μειονεκτήματα:

- Διαφορετικές συμπεριφορές του αλγορίθμου για άλλο σύνολο δεδομένων
- *Η εξάρτηση του χρόνου επεξεργασίας από το υλικό, τη γλώσσα προγραμματισμού και τον μεταφραστή.*

❖ ***Λανθασμένες εκτιμήσεις***

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Είναι δύσκολο να συγκρίνουμε αλγόριθμους…

….μετρώντας τον χρόνο εκτέλεσης.

Έστω δύο διαφορετικοί αλγόριθμοι που εκτελούν την ίδια εργασία, π.χ. αναζήτηση στοιχείου σε πίνακα (σειριακή και δυαδική αναζήτηση). Ποιος είναι καλύτερος; Μπορούμε να εκτελέσουμε και τους δύο αλγορίθμους και να χρονομετρήσουμε. Όμως:

- Πολλές εργασίες ***τρέχουν παράλληλα στον υπολογιστή***, άρα η ταχύτητα εξαρτάται και από τον φόρτο του συστήματος,
- Ο χρόνος εκτέλεσης εξαρτάται και από την ***συγκεκριμένη είσοδο***. Αν στην σειριακή αναζήτηση το ζητούμενο στοιχείο είναι πρώτο στην λίστα, τότε είναι γρηγορότερη από την δυαδική.

Για αυτό προέκυψε μια ***θεωρητική προσέγγιση*** ανάλυσης αλγορίθμων ανεξάρτητα από τους υπολογιστές και την συγκεκριμένη είσοδο.

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Θεωρητική προσέγγιση της Πολυπλοκότητας:
καθορίζεται μαθηματικά ο χρόνος εκτέλεσης του αλγορίθμου, ***ως συνάρτηση του μεγέθους της εισόδου του***. Τα πλεονεκτήματα της θεωρητικής προσέγγισης του υπολογισμού της πολυπλοκότητας ενός αλγόριθμου είναι ότι:

- δεν εξαρτάται από τη μηχανή,
- δεν εξαρτάται από τη γλώσσα προγραμματισμού, και
- δεν εξαρτάται από τις ικανότητες του προγραμματιστή.

Χρησιμοποιείται μία μεταβλητή **n**, που εκφράζει το μέγεθος του προβλήματος, και η συνάρτηση του τύπου **f(n)** που μας βοηθά να εκτιμήσουμε τον *χρόνο επεξεργασίας* και τον *απαιτούμενο χώρο μνήμης* - χρονική πολυπλοκότητα *(time complexity)* και πολυπλοκότητα χώρου *(space complexity)*.

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

- **Δεν μας ενδιαφέρουν οι επακριβείς τιμές**, αλλά μόνο η ***γενική συμπεριφορά των αλγορίθμων***, δηλαδή η τάξη του αλγορίθμου

- **Μονάδα έκφρασης** της θεωρητικής πολυπλοκότητας χρόνου ενός αλγόριθμου είναι η **βασική πράξη.** Για την εκτίμηση της πολυπλοκότητας θα χρειαστούμε μόνο ***το πλήθος των βασικών πράξεων που εκτελούνται*** από έναν αλγόριθμο και ***όχι τον ακριβή χρόνο*** που απαιτούν κάθε μία από τις πράξεις αυτές.

Βασικές πράξεις:

- Ανάθεση μιας τιμής σε κάποια μεταβλητή
- Σύγκριση δύο τιμών
- Αύξηση κάποια τιμής μεταβλητής
- Βασικές αριθμητικές πράξεις (π.χ. πρόσθεση, πολλαπλασιασμός, κλπ.)
- Εύρεση της τιμής ενός συγκεκριμένου στοιχείου σ' ένα πίνακα

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Παράδειγμα υπολογισμού (εύρεση του max-στοιχείου σε πίνακα):

```
(1)  int max = Array[0];
(2)  for (int i=0; i<n; ++i){
(3)    if (Array[i] >= max){
(4)      max=Array[i]; } }
```

(1) δύο εντολές (εύρεση του Array[0] και ανάθεση τιμής στη max).

(2) δύο εντολές την 1η φορά (ανάθεση i=0, σύγκριση i<n) και άλλες δύο για κάθε επανάληψη (αύξηση του i και σύγκριση i<n).
Άρα **χωρίς το σώμα της for** έχουμε: 4+2n εντολές *(n-επαναλήψεις)*.
Άρα μπορούμε να ορίσουμε μια συνάρτηση f(n), δοθέντος του n:

f(n) = 4 + 2n

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

Ανάλυση χειρότερης περίπτωσης *(worst-case analysis)*

(3) if (Array[i] >= max){… (2 εντολές – εύρεση στοιχείου και σύγκριση). Αν ο πίνακας έχει π.χ. 4 στοιχεία τότε η **χειρότερη περίπτωση** είναι η διάταξη {1, 2, 3, 4}. Δηλ., σε αυτή την περίπτωση θα έχουμε: **f(n)=4 + 2n + 4n =6n + 4**.

Ασυμπτωτική συμπεριφορά και χρονική πολυπλοκότητα

Στη f(n)=6n + 4, θα διαγράψουμε όλους τους όρους και σταθερές που δεν μεγαλώνουν ή μεγαλώνουν αργά και θα κρατήσουμε τους **όρους που μεγαλώνουν γρήγορα** όσο το **(n)** μεγαλώνει (διαφορετικές γλώσσες, διαφορετικοί χρόνοι εκτέλεσης, αρχικοποίησης, κλπ.). Άρα: f(n)=6n, διαγράφουμε και τον πολ/τή 6 (διαφορετικοί compilers) και έχουμε τελικά:

f(n) = n

Άρα: η **ασυμπτωτική συμπεριφορά** της f(n) = 6n + 4 περιγράφεται από τη συνάρτηση f(n) = n *(όσο το n μεγαλώνει)*. Όταν έχουμε βρει την f ασυμπτωτικά, τότε θα λέμε ότι ο αλγόριθμος μας έχει **χρονική πολυπλοκότητα** Θ(f(n)) ή απλούστερα ο αλγόριθμός μας με Θ(n) έχει **πολυπλοκότητα n**.

Εισαγωγή στην Πολυπλοκότητα των Αλγορίθμων

- Στους δύσκολους αλγόριθμους δεν μπορούμε να κάνουμε τέτοιους υπολογισμούς εντολή-προς-εντολή, για αυτό ψάχνουμε να βρούμε ένα **άνω όριο** *(χειρότερη περίπτωση)* τέτοιο ώστε ο δικός μας αλγόριθμος να μην το ξεπερνά (συνήθως προσθέτουμε εντολές, ακόμη και άχρηστες).
- Έτσι για την έκφραση της πολυπλοκότητας, εισάγεται ο λεγόμενος **συμβολισμός Ο** *(O – notation ή big-O),* από την αγγλική λέξη *order.*
  Για παράδειγμα, αν ο αλγόριθμος μας είναι Θ(n), τότε λέμε ότι η πολυπλοκότητα του είναι **O(n)**. Δηλαδή, ο αλγόριθμος μας, ασυμπτωτικά, δεν θα είναι χειρότερος από **n** (αλλά καλύτερος ή ίδιος με αυτό).
- Είναι *ευκολότερο να βρούμε την Ο – πολυπλοκότητα* ενός αλγορίθμου από την Θ – πολυπλοκότητα, για αυτό και χρησιμοποιείται περισσότερο.
- ***Με τον συμβολισμό - Ο εκφράζουμε ένα άνω όριο*** *(χειρότερης περίπτωσης – πολλαπλάσιο της f(n)),* ***ειδικά για μεγάλες εισόδους τιμών, που ο αλγόριθμος μας δεν ξεπερνά.***

Υπολογισμός της χρονικής Πολυπλοκότητας *(1/7)*

Θα επιχειρήσουμε να ***κατατάξουμε τους αλγορίθμους*** ανάλογα με τη χρονική τους πολυπλοκότητα, π.χ. της τάξης O(1), O(n), ή $O(n^2)$, κλπ.

Παράδειγμα η σειριακή αναζήτηση:

Η αναζήτηση στοιχείου σε πίνακα εξαρτάται από το *πλήθος των στοιχείων του πίνακα*. Έτσι αν ο πίνακας έχει 20 στοιχεία, η εκτέλεση απαιτεί 20 βήματα, αν έχει 2000, η εκτέλεση απαιτεί 2000 βήματα. Άρα η σειριακή ή γραμμική αναζήτηση εκτελείται σε χρόνο **O(n)** ή ***'γραμμικό χρόνο'*** , όπου **n** ο αριθμός των στοιχείων του πίνακα. Διαφορετικά: η σειριακή αναζήτηση είναι πολυπλοκότητας **O(n)**.

- **O(n) - *σειριακή ή γραμμική πολυπλοκότητα*:**

```
public void printAll(int[] Pinakas) {
   for (int i: Pinakas) {
       System.out.println(i); }}
```

Υπολογισμός της χρονικής Πολυπλοκότητας *(2/7)*

Παράδειγμα αναζήτησης μεγίστης τιμής *(max)* σε πίνακα

Μια άλλη ματιά στην πολυπλοκότητα του αλγορίθμου: Τα στοιχεία του πίνακα είναι **n** και ανάλογα έχουμε συγκρίσεις και τοποθετήσεις τιμών. Αν το n=2, τότε έχουμε 1 σύγκριση, αν n=3, τότε έχουμε 2 συγκρίσεις. Δηλ. θέλουμε **n-1** συγκρίσεις για να βρούμε τον max. Όμως το σημαντικό είναι να υπολογίσουμε την πολυπλοκότητα ενός αλγορίθμου σε μεγάλο πλήθος στοιχείων, δηλ. για μεγάλα n, δηλ. το -1 στην έκφραση n-1 δεν παίζει σημαντικό ρόλο, όταν το n είναι πολύ μεγάλο. Για αυτό λέμε ότι και σε αυτή την περίπτωση η πολυπλοκότητα του αλγορίθμου είναι της τάξης **O(n)**.

O(1) – *σταθερή πολυπλοκότητα:*

```
public void printOnce(int[] array) {
    System.out.println(array[0]); }
```

- Η μέθοδος εκτελείται σε χρόνο O(1) ή ***'σταθερό χρόνο'*** σε σχέση με την είσοδο. Το array μπορεί να έχει 1, 200,… στοιχεία, η εκτέλεση όμως απαιτεί ένα βήμα.

Υπολογισμός της χρονικής Πολυπλοκότητας *(3/7)*

$O(n^2)$ – *τετραγωνική πολυπλοκότητα*:

```
import java.util.*;
class Test_O_notation {
 public static void main(String args[]){
   int[] ar={1,2,3,4,5,6,7,8};
   printAll(ar); }
public static void printAll(int[] array) {
    for (int i: array) {
        for (int j: array) {
            int[] y = new int[]{i,j};
   System.out.println(Arrays.toString(y));}}}}
```

Διπλό loop συνήθως δίνει πολυπλοκότητα **$O(n^2)$**

- Εδώ έχουμε εστιασμένα loops. Αν ο πίνακας έχει n στοιχεία, το εξωτερικό loop εκτελείται n-φορές, και το *εσωτερικό n-φορές για κάθε επανάληψη του εξωτερικού*, έτσι θα έχουμε **n^2** εκτυπώσεις. Έτσι η μέθοδος εκτελείται σε **$O(n^2)$** χρόνο (*'τετραγωνικό χρόνο – τετραγωνική πολυπλοκότητα'*). Παραδείγματα οι ταξινομήσεις φυσαλίδας (bubble sort), με εισαγωγή (insertion sort) και με επιλογή (selection sort).

Υπολογισμός της χρονικής Πολυπλοκότητας *(4/7)*

O(log n) - *λογαριθμική πολυπλοκότητα* :

```
while ( low <= high ) {
    mid = ( low + high ) / 2;
    if ( target < list[mid] )
      high = mid - 1;
    else if ( target > list[mid] )
      low = mid + 1;
    else break;}
```

Ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται από το πλήθος των διαιρέσεων του n δια του 2. Και αυτό γιατί ο αλγόριθμος ***διαιρεί τον 'χώρο' στο μισό σε κάθε επανάληψη***. Η βάση του λογαρίθμου είναι το 2, που δεν επιδρά στον αλγόριθμο για αυτό μπορεί να παραλειφθεί. Έτσι λέμε ότι η λειτουργία εκτελείται σε **O(log n)** χρόνο (***'λογαριθμική πολυπλοκότητα'***). Την λογαριθμική πολυπλοκότητα θα δούμε αναλυτικά στην *δυαδική αναζήτηση*.

Υπολογισμός της χρονικής Πολυπλοκότητας *(5/7)*

O(n.log n) – ***συνδυασμός γραμμικής και λογαριθμικής πολυπλοκότητας:***

```
void quicksort (int list[], int left, int right)
    {
      int pivot = partition ( list, left, right );
      quicksort ( list, left, pivot - 1 );
      quicksort ( list, pivot + 1, right );    }
```

- Ο χρόνος εκτέλεσης του αλγορίθμου υπολογίζεται από το **n*log(n).** Δηλ. από τα n-loops *(απλά ή αναδρομικά), και το n,* έτσι η πολυπλοκότητα υπολογίζεται ως ***συνδυασμός γραμμικής και λογαριθμικής πολυπλοκότητας.*** Οι αλγόριθμοι της γρήγορης ταξινόμησης (quick sort) και ταξινόμησης με συγχώνευση (merge sort) και είναι του τύπου O(n*log(n)).

Υπολογισμός της χρονικής Πολυπλοκότητας *(6/7)*

Άλλοι συνδυασμοί χρονικών πολυπλοκοτήτων:

- **$O(n^3)$:** Κυβική πολυπλοκότητα, για προβλήματα μικρού μεγέθους.
- **$O(2^n)$:** Εκθετική πολυπλοκότητα. Σπάνια χρησιμοποιείται στην πράξη.
- **Βέλτιστος αλγόριθμος *(optimal):*** αν αποδειχτεί ότι είναι τόσο αποτελεσματικός, ώστε να μην μπορεί να κατασκευαστεί καλύτερός του.
- **Πολυωνυμικοί αλγόριθμοι *(polynomial)*:** λέγονται οι αλγόριθμοι με χρόνο εκτέλεσης $T(n)=O(n^k)$, όπου k μια θετική σταθερά, π.χ. οι αλγόριθμοι $O(n)$, $O(n^2)$, είναι πολυωνυμικοί.
- **Εκθετικοί ή μη πολυωνυμικοί:** λέγονται οι αλγόριθμοι με χρόνο εκτέλεσης $T(n)=O(c^n)$, όπου $c>1$, π.χ. οι αλγόριθμοι $O(2^n)$, $O(n.2^n)$.

Υπολογισμός της χρονικής Πολυπλοκότητας *(7/7)*

Απόδοση αλγορίθμων αναζήτησης

Για το ίδιο μέγεθος πίνακα, ο χρόνος εκτέλεσης του αλγορίθμου εξαρτάται από το στοιχείο που αναζητάμε (είσοδος). Όταν αναλύουμε αλγορίθμους αναζήτησης κοιτάμε μόνο τις ***συγκρίσεις***, γιατί δεν έχουμε αλλαγές *(μετακίνηση τιμών)*. Κοιτάμε:

- Την **καλύτερη περίπτωση** *(best-case input)***:** ο μικρότερος αριθμός συγκρίσεων για να βρούμε το στοιχείο.

- Την **χειρότερη περίπτωση** *(worst-case input)***:** ο μεγαλύτερος αριθμός συγκρίσεων για να βρούμε το στοιχείο. Συνήθως επιλέγεται γιατί υλοποιείται εύκολα.

- Την **μέση περίπτωση** *(average-case analysis)***:** η μέση τιμή των συγκρίσεων. Υλοποιείται δύσκολα, γιατί έχουμε ποικίλες εισόδους.

Υπολογισμός της Πολυπλοκότητας χώρου *(1/2)*

Μερικές φορές θέλουμε να μειώσουμε τη **χρήση μνήμης** *(χώρου)* αντί της μείωσης του χρόνου ή και σε συνδυασμό τα δύο. Το 'κόστος' σε 'μνήμη' *(πολυπλοκότητα χώρου)* είναι παρόμοιο με το κόστος σε χρόνο *(πολυπλοκότητα χρόνου)*. Απλά αναλύουμε το **συνολικό μέγεθος, *σε σχέση με το μέγεθος της εισόδου***, κάθε μεταβλητής που ορίζουμε.

Παραδείγματα:

```
public void print(int n) {
    for (int x = 0; x < n; x++) {
        System.out.println("hello"); }}
```

Η μέθοδος απαιτεί **O(1)** χώρο, δεν δεσμεύεται κάποια νέα μεταβλητή.

Υπολογισμός της Πολυπλοκότητας χώρου *(2/2)*

```
public String[] arrayX(int n) {
   String[] Array1 = new String[n];
   for (int i = 0; i < n; i++) {
       Array1[i] = "hello";
   }
   return Array1; }
```

- Η μέθοδος απαιτεί **O(n)** χώρο (το μέγεθος του Array1 κλιμακώνεται σύμφωνα με το μέγεθος της εισόδου).

- ***Στους διαφορετικούς τρόπους αναζήτησης αλλά και ταξινόμησης που ακολουθούν θα υπολογίζουμε την πολυπλοκότητα και θα συγκρίνουμε τις αποδώσεις των αλγορίθμων.***

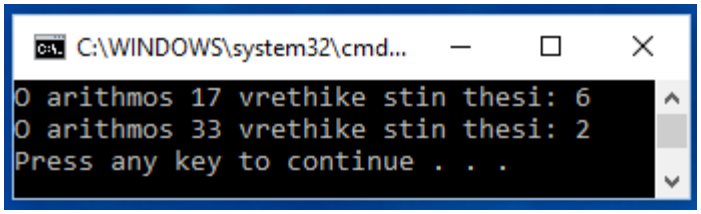
Σειριακή ή γραμμική αναζήτηση *(Sequential Search) (1/3)*

- Συγκρίνουμε το στοιχείο που αναζητούμε με τα διαδοχικά στοιχεία της λίστας (π.χ. πίνακα), ξεκινώντας από το πρώτο στοιχείο.
- Η αναζήτηση σταματά όταν βρεθεί το στοιχείο που αναζητάμε ή όταν φτάσουμε στο τέλος της λίστας.
- Σε ταξινομημένες λίστες σταματάμε στο στοιχείο με μεγαλύτερη τιμή από το προς αναζήτηση.
- Στην παρακάτω υλοποίηση αναζητάμε το στοιχείο key σε ένα ***μη ταξινομημένο πίνακα.***

```
public class SequentialSearch {
   public static int linerSearch(int[] arr, int key){
       int size = arr.length;
       for(int i=0;i<size;i++){
           if(arr[i] == key){
               return i; }
       }
       return -1;}
```

Σειριακή ή γραμμική αναζήτηση *(Sequential Search) (2/3)*

```
public static void main(String a[]){
  int[] arr1= {2,35,11,45,80,12,17,44};
  int searchKey = 17;
  System.out.println("O arithmos "+searchKey+" vrethike stin
          thesi: "+linerSearch(arr1, searchKey));
  int[] arr2= {12,51,33,55,134,13,74,56};
  searchKey = 33;
  System.out.println("O arithmos "+searchKey+" vrethike stin
           thesi: "+linerSearch(arr2, searchKey)); } }
```



Σειριακή ή γραμμική αναζήτηση *(Sequential Search) (3/3)*

Απόδοση σειριακής / γραμμικής αναζήτησης:

- ***καλύτερη περίπτωση:*** να βρούμε το στοιχείο στην $1^{η}$ θέση του πίνακα, άρα η μικρότερη τιμή είναι το **1**. Σε αυτή την περίπτωση η χρονική πολυπλοκότητα είναι O(1).
- ***χειρότερη περίπτωση:*** να το βρούμε στην τελευταία θέση του πίνακα ή να μην το βρούμε. Τότε η τιμή ισούται με το πλήθος των στοιχείων του πίνακα **(n).** Σε αυτή την περίπτωση η χρονική πολυπλοκότητα είναι **O(n)**.
- ***μέση περίπτωση:*** να το βρούμε στη μέση του πίνακα. Τότε ο αρ. των συγκρίσεων θα είναι **(n/2)**. Άρα και σε αυτή την περίπτωση η χρονική πολυπλοκότητα θα είναι **O(n)**.
- Άρα για την ***χειρότερη*** και ***μέση*** *περίπτωση* το πλήθος των συγκρίσεων εξαρτάται από το n (πλήθος στοιχείων). Σε αυτές τις περιπτώσεις ο αριθμός των συγκρίσεων είναι της τάξης **n, άρα έχουμε O(n)**. Στην ***καλύτερη*** περίπτωση έχουμε **O(1)**.

Δυαδική αναζήτηση *(Binary Search) (1/7)*

Εφαρμόζεται μόνο σε **ταξινομημένα** στοιχεία. Η ιδέα αυτής της αναζήτησης είναι ότι κάθε φορά που κάνουμε μια σύγκριση ***μικραίνουμε την λίστα των τιμών στο μισό*** μέχρις ότου βρούμε το στοιχείο ή δεν το βρούμε στην λίστα των τιμών.

Ο αλγόριθμος αναζήτησης:

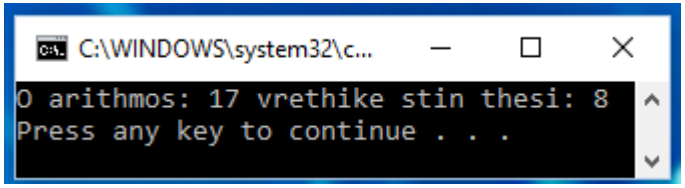
- Βρίσκουμε το ***μεσαίο στοιχείο του ταξινομημένου πίνακα*** και το συγκρίνουμε με το αναζητούμενο στοιχείο. Αν το βρήκαμε τότε σταματάμε την αναζήτηση.

- Αν δεν βρέθηκε, τότε ελέγχουμε αν το αναζητούμενο στοιχείο είναι μικρότερο ή μεγαλύτερο από το μεσαίο στοιχείο. Αν είναι μικρότερο, τότε περιορίζουμε την αναζήτηση στο ***πρώτο μισό*** του πίνακα (αύξουσα τάξη τιμών), ενώ αν είναι μεγαλύτερο, περιορίζουμε την αναζήτηση στο ***δεύτερο μισό***.

- Η διαδικασία αυτή συνεχίζεται για το κατάλληλο πρώτο ή δεύτερο μισό του πίνακα, ***το ¼ του πίνακα,*** κλπ., μέχρι να βρεθεί το στοιχείο ή να μην είναι δυνατόν να χωριστεί περαιτέρω ο πίνακας σε δύο μέρη.

Δυαδική αναζήτηση *(Binary Search) (2/7)*

```
class MyBinarySearch1 {
 public static void main (String[] args) {
  int orderednumbers[] = {-31, -22, 1, 3, 4, 5, 9, 10, 17, 23 };
  int key=17;
  System.out.println("O arithmos: "+ key + " vrethike stin
       thesi: "+ MyBinarySearch1.binarysearch(orderednumbers, key) );}

public static int binarysearch(int[] A, int x) {
 int left = 0, right = A.length-1;
 int mid, found = -1;
 while (found == -1 && left <= right) { mid = (left + right) / 2;
  if (x < A[mid]) {  // to x sto 1o miso
     right = mid-1;}
  else if (x > A[mid]) { // to x sto 2o miso
        left = mid + 1;}
  else found = mid; }
  return found; } }
```



Δυαδική αναζήτηση *(Binary Search) (3/7)*

Απόδοση της Δυαδικής αναζήτησης *(1/2)*

Ο πίνακάς ***σπάει στη μέση*** σε κάθε κλήση, έως ότου φτάσουμε στο 1 στοιχείο. Δηλ.,

$1^η$ επανάληψη: n / 2
$2^η$ επανάληψη: n / 4
$3^η$ επανάληψη: n / 8
...
i-οστή επανάληψη: $n / 2^i$
...
τελευταία επανάληψη: 1

*Σε ποια επανάληψη **(i)** θα βρούμε το στοιχείο που ζητάμε;* Πρέπει να λύσουμε την εξίσωση:
$1 = n / 2^i$
Πολ/ζουμε τα δύο σκέλη επί **2^i** και έχουμε:
$2^i = n$ *(α), (*το n είναι μια δύναμη του 2).
Στους *δυαδικούς λογάριθμους (βάση το 2)* αν λύσουμε ως προς i την *(α)* θα έχουμε:
i = log(n)

- Δηλ., ο ***αριθμός των επαναλήψεων*** που χρειάζεται για να κάνουμε δυαδική αναζήτηση είναι **log(n)**, όπου n είναι ο αριθμός των στοιχείων του πίνακα. Π.χ. για πίνακα με n=32, θα χρειαστούμε **5**-διαιρέσεις του πίνακα για να βρούμε το 1-στοιχείο *(που είναι ο λογάριθμος του 32*). Δηλ., **32 -> 16 -> 8 -> 4 -> 2 -> 1**
- ***Άρα η πολυπλοκότητα της δυαδικής αναζήτησης είναι O(log(n)).***

Δυαδική αναζήτηση *(Binary Search) (4/7)*

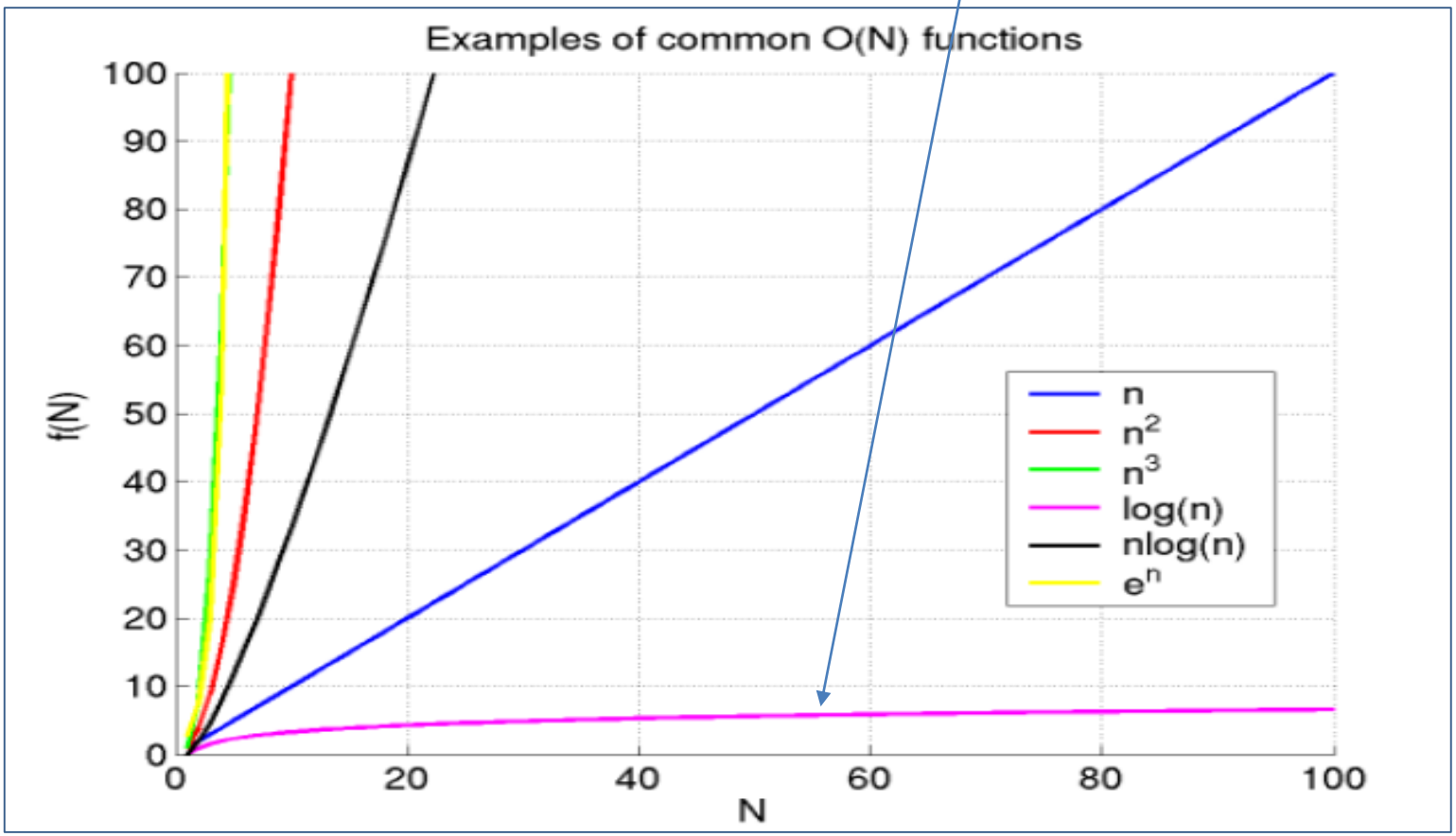
Απόδοση της Δυαδικής αναζήτησης *(2/2)*

- ***καλύτερη περίπτωση:*** να βρούμε το στοιχείο με την πρώτη, δηλ. στην μέση του πίνακα, άρα η μικρότερη τιμή είναι το 1, άρα σε αυτή την περίπτωση έχουμε **O(1)**.
- ***χειρότερη περίπτωση:*** το αναζητούμενο στοιχείο δεν βρίσκεται στην λίστα ή είναι το 1ο ή τελευταίο στοιχείο της λίστας, ή είναι μια θέση πιο μπροστά ή πίσω από το μεσαίο στοιχείο. Τότε ο αριθμός των συγκρίσεων είναι της τάξης **O(log n)**.
- ***μέση περίπτωση:*** όταν το αναζητούμενο στοιχείο βρίσκεται σε οποιοδήποτε άλλη θέση στη λίστα. Ο αριθμός των συγκρίσεων είναι περίπου ο ίδιος με την χειρότερη περίπτωση, δηλ. είναι της τάξης **O(log n)**.
- Σε όλες αυτές τις αναζητήσεις ο αριθμός των συγκρίσεων είναι $\log_2^n$, δηλ. η χειρότερη περίπτωση της δυαδικής αναζήτησης είναι της τάξης **O(log n)**.
- Γενικά, όταν ο αλγόριθμος *διαιρεί στη μέση την λίστα τιμών* ο αριθμός των συγκρίσεων είναι της τάξης **O(log n)**.

Δυαδική αναζήτηση *(Binary Search) (5/7)*

Σύγκριση σειριακού και δυαδικού αλγορίθμου αναζήτησης

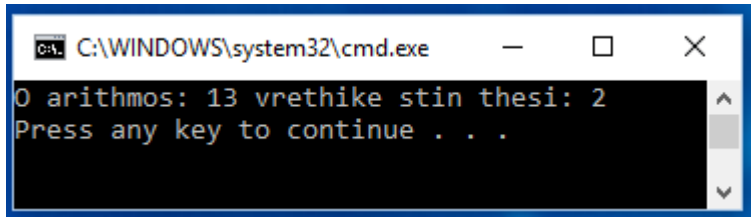
Οι αλγόριθμοι O(log n) εκτελούνται ταχύτερα από τους O(n). Όταν το n είναι μικρό οι διαφορές είναι μικρές, ενώ όσο το n μεγαλώνει, τόσο οι διαφορές στην απόδοση μεγαλώνουν.



Αναδρομική Δυαδική Αναζήτηση *(Recursive Binary Search) (6/7)*

Η επανάληψη της διαδικασίας των διαδοχικών διαιρέσεων του πίνακα, με κάθε αναζήτηση, καθιστά την δυαδική αναζήτηση κατάλληλη για αναδρομική υλοποίηση. Όπως και στη μη αναδρομική υλοποίηση έτσι και εδώ θα χρησιμοποιήσουμε τα **left** και **right**.

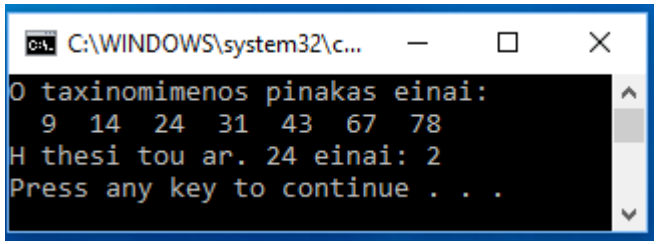
```
class RecursiveBinarySearch {
  public static void main (String[] args)   {
    int arr[] = {11, 12, 13, 14, 15, 16, 17, 18 ,19, 20};
    int key=13;
    int found = recBinarySearch(arr, key, 0, arr.length - 1);
    if (found > -1){
      System.out.println ("O arithmos: " + key + " vrethike stin thesi: " +
            found);}
    else{System.out.println("O arithmos den vrethike"); } }

public static int recBinarySearch(int[] arr, int key, int left, int right)
{   int mid;
    if (right < left){return -1;}
    mid = (left + right) / 2;

    if (arr[mid] < key)
        return recBinarySearch(arr, key, mid + 1, right);
    else if (arr[mid] > key)
        return recBinarySearch(arr, key, left, mid - 1);
    else return mid; } }
```



Εμπεριεχόμενη μέθοδος Δυαδικής Αναζήτησης στη Java *(7/7)*

Η μέθοδος: **java.util.Arrays.binarySearch(int[] a, int key),** αναζητά στον πίνακα των **ακεραίων a[]** την τιμή **value,** χρησιμοποιώντας τον αλγόριθμο binary search. Ο *πίνακας* ***πρέπει να είναι ταξινομημένος*** πριν την κλήση της μεθόδου, διαφορετικά τα αποτελέσματα θα είναι απροσδιόριστα.

```
import java.util.Arrays;
public class ArraysBinarySearch {
   public static void main(String[] args) {
   int Array[] = {14,31,67,43,24,9,78};

   // taxinomisi pinaka
   Arrays.sort(Array);
   System.out.println("O taxinomimenos pinakas einai: ");
   for (int i : Array) {System.out.print("  "+i);}
   System.out.println();

   // O ar. anazitisis p.x. 24
   int key = 24;
   int thesi = Arrays.binarySearch(Array,key);
   System.out.println("H thesi tou ar. 24 einai: " + thesi);}}
```



Κατακερματισμός - Πίνακες Κατακερματισμού – Συναρτήσεις Κατακερματισμού *(1/13)*

Αν οι εργασίες αναζήτησης αφορούσαν ένα πίνακα μικρών ακεραίων, τότε θα μπορούσαμε να ορίσουμε ένα *πίνακα με δείκτες* τους ακέραιους, δηλ. τα στοιχεία που θέλουμε να προσπελάσουμε – ***κλειδιά***. Π.χ. ένα πίνακα 1000 κελιών με δείκτες τους αρ. 1-1000, για τους 1000 πελάτες μιας επιχείρησης).

- Στην πράξη όμως έχουμε μεγάλο πλήθος στοιχείων ή τα κλειδιά δεν είναι ακέραιοι αριθμοί και επιπλέον πρέπει να αυξήσουμε την ταχύτητα αναζήτησης.

- Σε αυτές τις περιπτώσεις υλοποιούμε μια τεχνική αντιστοίχισης - μετατροπής των κλειδιών αναζήτησης σε μοναδικούς ακέραιους τον **κατακερματισμό** *(hashing),* με την βοήθεια μιας **συνάρτησης κατακερματισμού** *(hash function **(hf)**),* που αποθηκεύουμε σε ένα **πίνακα κατακερματισμού** *(hash table).*

Ανάλυση πολυπλοκότητας:

- Αν οι τιμές του πίνακα δεν είναι ταξινομημένες θα χρειαστεί να ελέγξουμε μια προς μια όλες τις τιμές του πίνακα. Αν είναι ταξινομημένες μειώνουμε την πολυπλοκότητα στο **O(log n)** με την δυαδική αναζήτηση *(χειρότερη προσπάθεια)*. Όμως θα είχαμε άμεση προσπέλαση, δηλ. O(1)-πολυπλοκότητα, αν γνωρίζαμε τον δείκτη προσπέλασης του στοιχείου (διεύθυνση στον πίνακα). Ο δείκτης δημιουργείται με την βοήθεια της hf και για κάθε δεδομένο κλειδί.

Κατακερματισμός - Πίνακες Κατακερματισμού – Συναρτήσεις Κατακερματισμού *(2/13)*

Η διαδικασία τοποθέτησης αντικειμένων με την χρήση της hf είναι η παρακάτω:

- Δημιουργία πίνακα μεγέθους n.
- Τοποθέτηση των αντικειμένων στον πίνακα κατακερματισμού με δείκτες – κλειδιά (hashcodes) που υπολογίζονται με την βοήθεια της hf: **δείκτης = h(object)**
- Δηλ. ο πίνακας κατακερματισμού είναι η δομή που συσχετίζει κλειδιά με τιμές.
- Κατά την διαδικασία κατακερματισμού μπορεί να προκύψει **σύγκρουση** *(collision),* δηλ. δύο κλειδιά να αντιστοιχηθούν στον ίδιο δείκτη. Θα πρέπει:

  (1) να επιλέγουμε με προσοχή την κατάλληλη συνάρτηση κατακερματισμού,

  (2) να εφαρμόσουμε στρατηγική επίλυσης των συγκρούσεων *(collision resolution strategy)*

Συναρτήσεις Κατακερματισμού (Hash Functions)

- Μια συνάρτηση κατακερματισμού h απεικονίζει κλειδιά ενός δεδομένου τύπου σε ακεραίους ενός σταθερού διαστήματος [0, n − 1]. Μια αποτελεσματική και απλή συνάρτηση καταμερισμού είναι της μορφής**: h(x) = x % n,** *(όπου το n = μέγεθος του πίνακα).* Θεωρείται αποτελεσματική όταν το n είναι πρώτος αριθμός. Ο ακέραιος **h(x)** ονομάζεται **τιμή κατακερματισμού** *(hash value)* του κλειδιού.

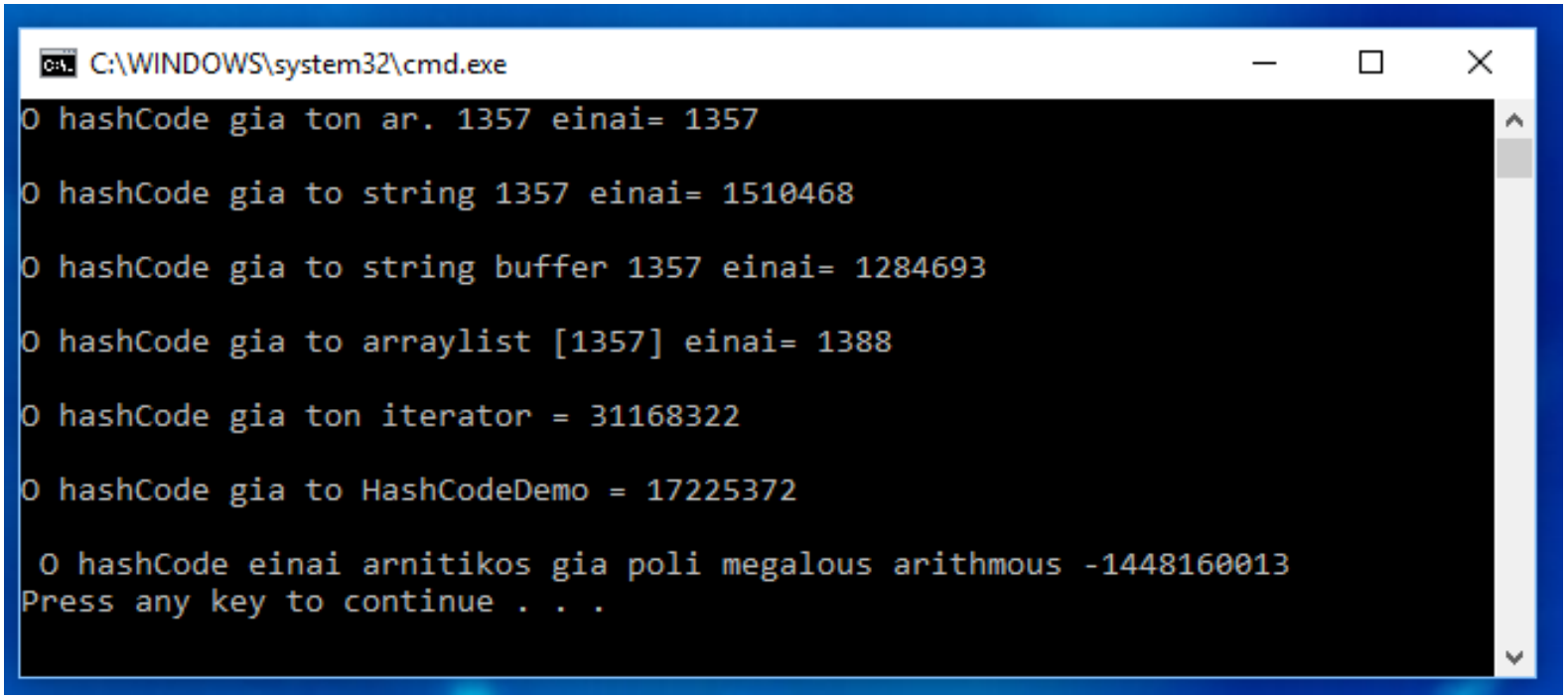
Κατακερματισμός - Πίνακες Κατακερματισμού – Συναρτήσεις Κατακερματισμού *(3/13)*

- Παράδειγμα απλού κατακερματισμού: έστω 10 κλειδιά στο διάστημα 0 – 100. Χρειαζόμαστε ένα πίνακα 10 θέσεων με δείκτες 0 – 9. Μία λύση: διαιρούμε τα κλειδιά με το 10, που είναι το μέγεθος του πίνακα και παίρνουμε το υπόλοιπο της διαίρεσης. Π.χ. η εγγραφή με κλειδί 13 θα αποθηκευτεί στη θέση 13%10 = 3. Κατά παρόμοιο τρόπο μπορεί να βρεθεί η θέση του πίνακα για οποιοδήποτε κλειδί: **Θέση πίνακα = αρχικό κλειδί % μέγεθος πίνακα**
- Μια τέτοια συνάρτηση καταμερισμού μετασχηματίζει ένα μεγάλο διάστημα τιμών σε ένα μικρότερο διάστημα.
- Τι θα συμβεί όμως για το κλειδί 23, ή 33, κλπ., δηλ. όταν περισσότερα κλειδιά αντιστοιχούν στην ίδια θέση; Τότε έχουμε σύγκρουση και οι τιμές που διεκδικούν την ίδια θέση **συνώνυμα** *(synonyms).*
- Ένας τρόπος επιλογής της hf είναι η χρήση της μεθόδου ***hashCode()*** της java που υλοποιείται στην κλάση object και κληρονομείται από κάθε άλλη κλάση. Η *hashCode()* παρέχει ένα αριθμό που αντιστοιχεί σε ένα αντικείμενο.

Κατακερματισμός - Πίνακες Κατακερματισμού – Συναρτήσεις Κατακερματισμού *(4/13)*

```
import java.util.*;
public class HashCodeDemo {
public static void main(String[] args) {
  Integer obj1 = new Integer(1357);
  System.out.println("O hashCode gia ton ar. "+obj1+ " einai= "+obj1.hashCode());

  String obj2 = new String("1357");
  System.out.println("\nO hashCode gia to string " +obj2+ " einai= "+obj2.hashCode());

  StringBuffer obj3 = new StringBuffer("1357");
  System.out.println("\nO hashCode gia to stringbuffer "+obj3+ " einai= "
          + obj3.hashCode());

  ArrayList<Integer> obj4 = new ArrayList<Integer>();
  obj4.add(new Integer(1357));
  System.out.println("\nO hashCode gia to arraylist " +obj4+ " einai= "
          + obj4.hashCode());

  Iterator obj5 = obj4.iterator();
  System.out.println("\nO hashCode gia ton iterator = " +obj5.hashCode());

  HashCodeDemo obj6 = new HashCodeDemo();
  System.out.println("\nO hashCode gia to HashCodeDemo = " + obj6.hashCode());

  String obj7 = new String("3000000000000000");
  System.out.println("\n O hashCode einai arnitikos gia poli megalous arithmous "
           + obj7.hashCode()); } }
```

Κατακερματισμός - Πίνακες Κατακερματισμού – Συναρτήσεις Κατακερματισμού *(5/13)*



- Η *hascode()* υλοποιείται διαφορετικά σε διαφορετικές κλάσεις, π.χ. στα Strings, υλοποιείται με την σχέση:
  s.charAt(0) * 31^{n-1} + s.charAt(1) * 31^{n-2} + ... + s.charAt(n-1), όπου *s* το String και *n* το μέγεθος του πίνακα. Π.χ. για το String:
  "ABC" = 'A' * 31^2 + 'B' * 31 + 'C' = 65 * 31^2 + 66 * 31 + 67 = 64578

Επίλυση Συγκρούσεων *(Collision Resolution)*

Όταν τοποθετούμε αντικείμενα στον πίνακα κατακερματισμού είναι πιθανόν δύο αντικείμενα να έχουν το ίδιο κλειδί – *hashCode()*, οπότε έχουμε σύγκρουση π.χ.

"Aa" = 'A' * 31 + 'a‘ = 2112

"BB" = 'B' * 31 + 'B' = 2112

Πως επιλύουμε την σύγκροουση;

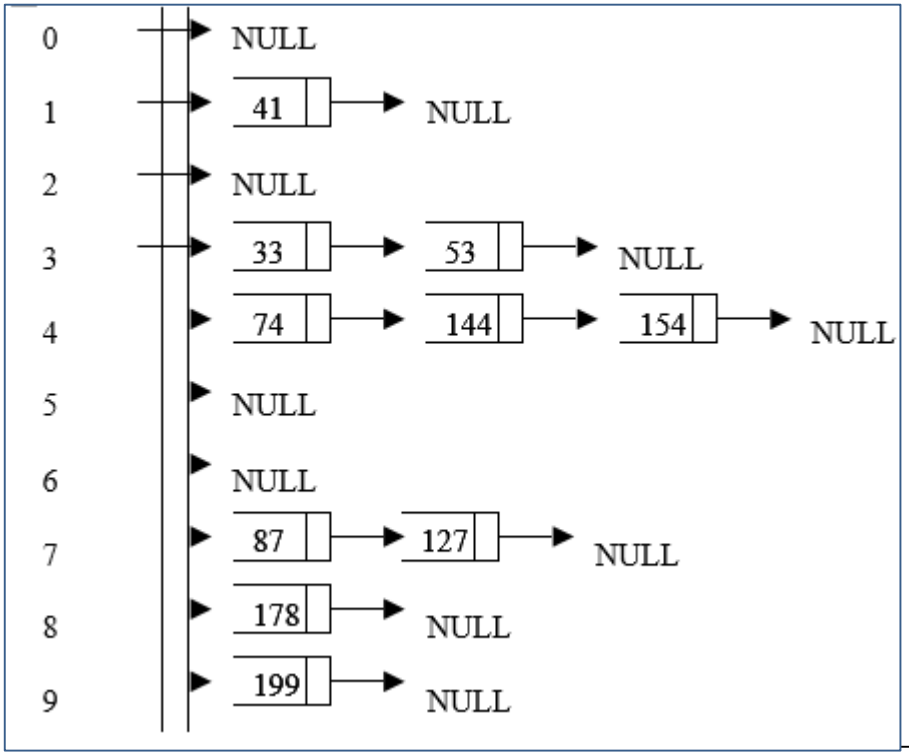
- Μία πρώτη προσέγγιση είναι η δημιουργία ενός πίνακα που αποτελείται από *συνδεδεμένες λίστες*. Έτσι, όταν συμβεί σύγκρουση, το νέο στοιχείο εισάγεται στη λίστα της συγκεκριμένης θέσης του πίνακα. Αυτή η τεχνική ονομάζεται **ξεχωριστή σύνδεση** *(separate chaining)*.

- Μία δεύτερη λύση στο πρόβλημα είναι να εξετάσουμε τον πίνακα, να βρούμε μία άδεια θέση και να εισάγουμε εκεί το νέο στοιχείο. Αυτή η τεχνική ονομάζεται **ανοιχτή διευθυνσιοδότηση** *(open addressing),* **ή γραμμική διερεύνηση** *(linear probing).*

Ξεχωριστή σύνδεση *(separate chaining)*

- Στην τεχνική αυτή ο πίνακας περιέχει ένα πλήθος από *συνδεδεμένες λίστες*.

Παράδειγμα:
Έστω τα κλειδιά 33, 41, 53, 74, 87, 127, 144, 154, 178, 199 που θέλουμε να αποθηκεύσουμε σε ένα πίνακα 10 θέσεων (0 – 9).



Χρησιμοποιώντας την μέθοδο της διαίρεσης των κλειδιών δια του 10 και παίρνοντας το υπόλοιπο ως τη θέση στον πίνακα, προκύπτει ότι τα κλειδιά 74, 144, 154, είναι συνώνυμα. Το ίδιο ισχύει και για τα 33, 53, και 87, 127.
Άρα θα χρησιμοποιήσουμε λίστες για τις θέσεις 3, 4 και 7.

Ανοιχτή διευθυνσιοδότηση *(open addressing)*

Η πιο απλή μέθοδος ανοιχτής διευθυνσιοδότησης είναι η **γραμμική εξέταση** *(linear probing)*. Η μέθοδος ψάχνει σειριακά στον πίνακα, ξεκινώντας από την επόμενη θέση από όπου συνέβη η σύγκρουση, μέχρις ότου να βρει μια άδεια θέση, όπου και αποθηκεύει το νέο στοιχείο.

Παράδειγμα:

Έστω ότι έχουμε 10 κλειδιά με τιμές 13, 28, 41, 70, 127, 131, 144, 176, 182, 199. Όταν γίνει η αποθήκευση των κλειδιών σε ένα πίνακα 10 θέσεων, θα καταλάβουν τις εξής θέσεις:

0	1	2	3	4	5	6	7	8	9
70	41	**131**	13	144	**182**	176	127	28	199

Παρατηρούμε ότι το 131 έπρεπε να αποθηκευτεί στην θέση 1, όπου όμως προηγήθηκε η αποθήκευση του 41. Επομένως αποθηκεύεται στην αμέσως επόμενη κενή θέση που είναι η $2^{η}$. Όμως το 2 είναι η φυσική θέση του κλειδιού 182, που είναι κατειλημμένη, για αυτό το τοποθετούμε στην αμέσως επόμενη κενή θέση που είναι η $5^{η}$, κ.ό.κ.

Java και Κατακερματισμός

Είδαμε ήδη την hashCode() που υπολογίζει μια ακέραια τιμή για κάθε τύπο δεδομένων η οποία καθορίζει την θέση του πίνακα κατακερματισμού στην οποία θα αποθηκευτεί. Το πακέτο java.util περιέχει την κλάση Hashtable που υλοποιεί τεχνικές κατά-κερματισμού αντιστοιχίζοντας κλειδιά σε τιμές. Ένα αντικείμενο Hashtable έχει 2-παραμέτρους:

- την **αρχική χωρητικότητα** *(initial capacity),* που ορίζει το πλήθος των στοιχείων που μπορεί να αποθηκευτεί στον πίνακα κατά την στιγμή της δημιουργίας του, και

- τον **παράγοντα φόρτου** *(load factor)*, που ορίζει πόσο επιτρέπεται να γεμίσει ο πίνακας πριν την αυτόματη αύξηση της χωρητικότητας του. Η χωρητικότητα αυξάνει, όταν οι εισαγωγές στον πίνακα κατακερματισμού ξεπεράσουν το γινόμενο της τρέχουσας χωρητικότητας επί τον παράγοντα φόρτου, καλώντας την μέθοδο *rehash().*

Java και Κατακερματισμός

Οι δομητές της Hastable (java 1.1):

- **public Hashtable()**
- **public Hashtable(int size)**
- **public Hashtable(int size, float load) throws IllegalArgumentException**
- Δημιουργούν ένα πίνακα κατακερματισμού με αρχική χωρητικότητα-size ή αφήνοντας το αρχικό (default) μέγεθος που είναι 101 και τον παράγοντα φόρτου με τιμή 0.75.

- Στην Java 2 χρησιμοποιούμε την **HashMap** αντί της Hastable. Η κλάση HashMap υλοποιεί έναν πίνακα αντιστοίχισης. Ένας τέτοιος πίνακας αντιστοιχίζει κλειδιά σε τιμές. Δεν επιτρέπονται διπλά κλειδιά, ενώ κάθε κλειδί αντιστοιχίζεται σε μία το πολύ τιμή. Μια τέτοια δομή επιτρέπει την αναζήτηση μιας συγκεκριμένης τιμής με βάση ένα μοναδικό κλειδί. Παράδειγμα ενός HashMap αποτελεί ένα λεξικό, όπου το λήμμα αποτελεί το κλειδί ενώ η ερμηνεία του, για παράδειγμα, η μετάφρασή του σε μία άλλη γλώσσα, αποτελεί την τιμή (value) στην οποία αντιστοιχίζεται το λήμμα.

Java και Κατακερματισμός

Παράδειγμα:

```
HashMap<String, String> dictionary = new HashMap<String, String>();
dictionary.put("boy", "αγόρι");
String trans = dictionary.get("boy");
```

Η συμβολοσειρά trans του παραδείγματος παίρνει την τιμή "αγόρι" , με την οποία έχει αντιστοιχηθεί στο λεξικό η συμβολοσειρά "boy".

Δομητές:

HashMap()

Constructs an empty `HashMap` with the default initial capacity (16) and the default load factor (0.75).

HashMap(int initialCapacity)

Constructs an empty `HashMap` with the specified initial capacity and the default load factor (0.75).

HashMap(int initialCapacity, float loadFactor)

Constructs an empty `HashMap` with the specified initial capacity and load factor.

Java και Κατακερματισμός

HashMap(Map<? extends K,? extends V> m)
Constructs a new HashMap with the same mappings as the specified Map.

Μέθοδοι:

void	**clear**() Removes all of the mappings from this map.
Object	**clone**() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

Java και Κατακερματισμός

boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map.
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	**remove**(Object key) Removes the mapping for the specified key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.