Διάλεξη 10: Αλγόριθμοι Ταξινόμησης ΙΙ

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Οι αλγόριθμοι ταξινόμησης:
  - **Δ. QuickSort** – Γρήγορη Ταξινόμηση
  - **Ε. BucketSort** – Ταξινόμηση με Κάδους
- Έμμεση Ταξινόμηση
- Εξωτερική Ταξινόμηση

Διδάσκων: Παναγιώτης Ανδρέου

Δ. Γρήγορη Ταξινόμηση (QuickSort)

- Η **γρήγορη ταξινόμηση (QuickSort)** είναι, όπως ο Mergesort, διαδικασία **διαίρει και βασίλευε** (divide and conquer, δηλ. αναδρομική διαδικασία όπου το πρόβλημα μοιράζεται σε μέρη τα οποία λύνονται ξεχωριστά, και μετά οι λύσεις συνδυάζονται).
- Δεν χρειάζεται βοηθητικό πίνακα (όπως στην Mergesort)
- **Πρακτικά, ο πιο γρήγορος αλγόριθμος.**
- Στη χείριστη περίπτωση ο αλγόριθμος Quick Sort είναι $O(n^2)$ αλλά στην καλύτερη περίπτωση χρειάζεται $\Omega(n\log n)$:
- Τα περισσότερα συστήματα χρησιμοποιούν το QuickSort (π.χ., Unix) και οι περισσότερες γλώσσες προγραμματισμού το προσφέρουν σαν μέρος των βασικών βιβλιοθηκών τους πχ. C, JAVA, C++, etc.

QuickSort: Αλγόριθμος

- Αν το δεδομένο εισόδου περιέχει 0 ή 1 στοιχεία δεν κάνουμε τίποτα.
- Διαφορετικά, αναδρομικά:
  1. διαλέγουμε ένα στοιχείο **p** (ακόμα δεν ορίζουμε πιο ακριβώς), το οποίο ονομάζουμε το ***άξονα*** **(pivot)** στοιχείο και το αφαιρούμε από το δεδομένο εισόδου.
  2. χωρίζουμε τον πίνακα σε δύο μέρη **S1** και **S2**, όπου το **S1** θα περιέχει όλα τα στοιχεία του πίνακα που είναι μικρότερα από το **p**, και το **S2** θα περιέχει τα υπόλοιπα στοιχεία (όλα τα στοιχεία που είναι μεγαλύτερα ή ίσα από το **p**).
  3. Καλούμε αναδρομικά τον αλγόριθμο

     στο **S1**, και παίρνουμε απάντηση το **T1**,

     και

     στο **S2**, και παίρνουμε απάντηση το **T2**.
  4. Επιστρέφουμε τον πίνακα [**T1, p, T2**].

QuickSort: Βασική Ιδέα

[72, 12, 1, 34, 3, 50, 28, 6, 5, 22, 91, 73]

χωρίζουμε με pivot το 28
(μπορούσε να είναι οποιοδήποτε άλλο στοιχείο)
Θέτουμε αριστερά του pivot τα μικρότερα και δεξιά τα μεγαλύτερα του

< 28   >= 28

[12, 1, 3, 6, 5, 22] **28** [72,34,50,91,73]

Quicksort   Quicksort

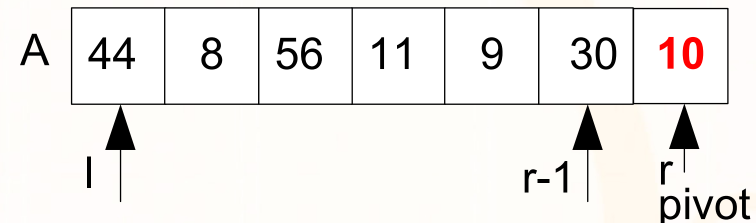
1, 3, 5, 6, 12, 22   **34, 50, 72, 73, 91**

Αποτέλεσμα: 1, 3, 5, 6, 12, 22, 28, 34, 50, 72, 73, 91

QuickSort: Ψευδοκώδικας

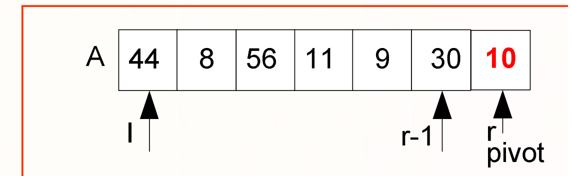
```
void Quicksort(int A[], int l, int r){
   if (l>=r) return;

   int pivotIndex = findPivotIndex(A, l, r);
   int pivot = A[pivotIndex];

   // κάνουμε swap τον pivot με το τελευταίο.
   swap(A, pivotIndex, r);

   /* Η διαδικασία partition χωρίζει τον πίνακα A[l…r-1] έτσι ώστε
   A[l..k-1] να περιέχει στοιχεία < pivot, A[k…r-1] να περιέχει
   στοιχεία >=pivot, και επιστρέφει την τιμή k. */
   int k = partition (A, l, r-1, pivot);

   // κάνουμε swap τον k με το τελευταίο.
   swap(A, k, r);

   Quicksort(A, l, k-1);
   QuickSort(A, k+1, r);
}
```

Έστω ότι διαλέξαμε τον μεσαίο, δηλ., (l+r)/2;. Θα μπορούσαμε να διαλέξουμε οποιονδήποτε άλλο



Διαδικασία Partition(A, l, r, p)

- Με δεδομένο εισόδου τον πίνακα A[l…r], και **pivot p**, θέλουμε να χωρίσουμε τον πίνακα σε δύο μέρη ως προς το p.
- Το πιο πάνω πρέπει να επιτευχθεί **χωρίς** τη χρήση δεύτερου πίνακα (το sorting θα γίνει επί τόπου).
- Βασική Ιδέα:



1. Επαναλαμβάνουμε τα εξής μέχρις ότου τα **l** και **r** να συναντηθούν.
2. Προχώρα το **r** προς τα **αριστερά** όσο τα στοιχεία που βρίσκεις είναι μεγαλύτερα (ή ίσα) του p,
3. Προχώρα το **l** προς τα **δεξιά** όσο τα στοιχεία που βρίσκεις είναι μικρότερα του **p**,
4. αντάλλαξε τα στοιχεία που δείχνονται από τα **l** και **r.**

Παράδειγμα Εκτέλεσης Partition

Δεδομένο Εισόδου:

index	0	1	2	3	4	5	6	7
	72	*6*	*37*	*48*	*30*	*42*	*83*	*75*

pivot = 48, μετακίνηση του pivot στο τέλος (swap(4, 8)):

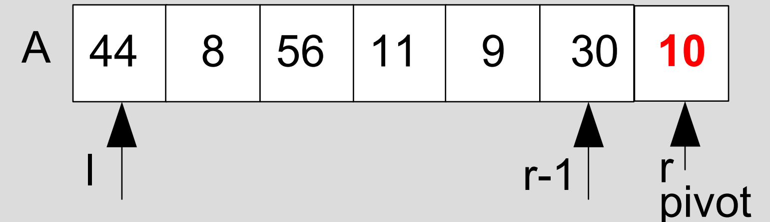
72	6	37	75	30	42	83	48 (pivot)
l						r	

εκτέλεση του Partition(A, l, r, 48):

72	6	37	75	30	42	83	48
l					r		
42	6	37	75	30	72	83	48
			l	r			
42	6	37	30	75	72	83	48
			l	r			
42	6	37	30	75	72	83	48
			r	l			

Quicksort: Υλοποίηση

```
public static void QuickSort(int A[], int l, int r) {
  int pivot, pivotIndex;

  if (l >= r) return;

  // Διαλέγουμε το pivot
  pivotIndex = findPivotIndex(l, r);
  pivot = A[pivotIndex];

  // Κάνουμε swap το pivot με το τελευταίο
  swap(A, pivotIndex, r);

  /* Η διαδικασία partition χωρίζει τον πίνακα A[l…r-1] έτσι ώστε
   * A[l..k-1] να περιέχει στοιχεία < pivot, A[k…r-1] να περιέχει στοιχεία
   * >=pivot, και επιστρέφει την τιμή k. */
   int k = partition(A, l, r - 1, pivot);

  // Κάνουμε swap το k με το τελευταίο
  if (A[r] < A[k]) swap(A, k, r);

  QuickSort(A, l, k - 1);
  QuickSort(A, k + 1, r);
}
```

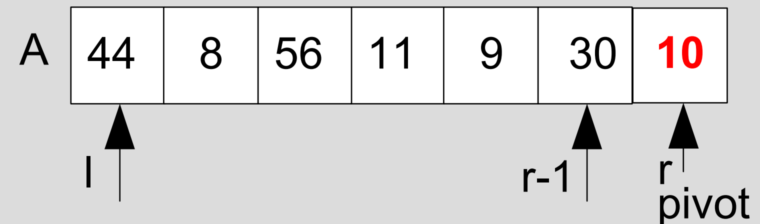


Quicksort: Υλοποίηση (συν.)

```
public static int findPivotIndex(int l, int r) {
  return (l + r) / 2;
}

public static void swap(int A[], int pivot, int r) {
  int tmp = A[pivot];
  A[pivot] = A[r];
  A[r] = tmp;
}
```

Quicksort: Υλοποίηση (συν.)

```
public static int partition(int A[], int l, int r, int pivot) {
  while (l < r) {
    // προχωρούμε από l στο r μέχρι να χρειαστεί ένα swap
    while (A[l] < pivot   &&   l < r)
      l++;     // leave "<pivot" on left

    // προχωρούμε από r στο l μέχρι να χρειαστεί ένα swap
    while (pivot <= A[r]   &&   l < r)
      r--;     // leave ">=pivot" on right

    if (l == r) break;

    // τώρα κάνε το swap
    if (A[l] >= pivot)
      swap(A, l, r); // move ">=" to the right
  }

  // επέστρεψε το σημείο στο οποίο θέλουμε να γίνει η
  // εισαγωγή του pivot
  return l;
}
```



Παράδειγμα Εκτέλεσης QuickSort

```
BEFORE:[72,6,37,48,30,42,83,75]
   Index: 0  1  2  3  4  5  6  7

** QuickSort [0,7]
[72,6,37,48,30,42,83,75,]
PivotIndex: 3(48)  =>   Swapping 48, 75
[72,6,37,75,30,42,83,48,]
Partitioning [0,6]
   Swapping 72, 42
[42,6,37,75,30,72,83,48,]
   Swapping 75, 30
[42,6,37,30,75,72,83,48,]
Inserting Pivot at Position:4
Swapping 75, 48
[42,6,37,30,48,72,83,75,]

** QuickSort [0,3]
[42,6,37,30,48,72,83,75,]
PivotIndex: 1(6)  =>    Swapping 6, 30
[42,30,37,6,48,72,83,75,]
Partitioning [0,2]
Inserting Pivot at Position:0
Swapping 42, 6

** QuickSort [0,-1] -> RETURN
```

```
** QuickSort [1,3]
[6,30,37,42,48,72,83,75,]
PivotIndex: 2(37)   =>   Swapping 37, 42
[6,30,42,37,48,72,83,75,]
Partitioning [1,2]
Inserting Pivot at Position:2
Swapping 42, 37

** QuickSort [1,1] -> RETURN
** QuickSort [3,3] -> RETURN

** QuickSort [5,7]
[6,30,37,42,48,72,83,75,]
PivotIndex: 6(83)   =>   Swapping 83, 75
[6,30,37,42,48,72,75,83,]
Partitioning [5,6] with pivot:83
Inserting Pivot at Position:6

** QuickSort [5,5] -> RETURN
** QuickSort [7,7] -> RETURN

AFTER:[6,30,37,42,48,72,75,83,]
```

Ανάλυση του Χρόνου Εκτέλεσης

- Η **εύρεση του pivot** απαιτεί χρόνο O(1) και η διαδικασία Partition(A, l, r, p) εκτελείται σε **χρόνο O(n)** σε κάθε επίπεδο της αναδρομής.
- Η **αναδρομική εκτέλεση** του QuickSort παίρνει στην χείριστη περίπτωση χρόνο **O(n)** και στην καλύτερη περίπτωση χρόνο **Ω(logn)**

- ***Χείριστη περίπτωση***:

  Κάθε φορά που επιλέγουμε τον pivot όλα τα στοιχεία τυγχάνει να ταξινομούνται είτε **μόνο** αριστερά του (δηλαδή <pivot) ή **μόνο** δεξιά του (δηλαδή >=pivot) πχ 9,9,9,9,9,9,9

  Σ**υνολικός** χρόνος εκτέλεσης **$T(n) \in O(n^2)$.**

- ***Βέλτιστη περίπτωση***:

  Κάθε φορά που επιλέγουμε τον pivot τα **μισά** στοιχεία ταξινομούνται αριστερά του (δηλαδή <pivot) και τα **υπόλοιπα μισά** δεξιά του (δηλαδή >=pivot) πχ 1,2,3,4,5,6,7

  Συνολικός χρόνος εκτέλεσης **$T(n) \in \Omega(n \log n)$.**

Ανάλυση Μέσης Περίπτωσης

- Θεωρούμε όλες τις πιθανές περιπτώσεις της συμπεριφοράς της μεθόδου Partition(A, i, j, p), όπου j-i = n-1. Υπάρχουν n τέτοιες περιπτώσεις: το αριστερό κομμάτι του partition μπορεί να έχει από 0 μέχρι n-1 στοιχεία.
- Ας υποθέσουμε πως οι n αυτές περιπτώσεις είναι ισοπίθανες, δηλαδή η κάθε μια έχει πιθανότητα 1/n.
- Τότε η μέση περίπτωση του T(n) δίνεται ως

$$T(n) = c \cdot n + \frac{1}{n} \cdot \sum_{i=0}^{n-1} [T(i) + T(n-1-i)]$$

$$= c \cdot n + \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i)$$

- Επίλυση της αναδρομικής σχέσης δίνει $T(n) \in O(n \log n)$.

Παρατηρήσεις

- Πως δουλεύει η μέθοδος Partition με δεδομένο εισόδου πίνακα με πολλά στοιχεία ίσα με το pivot;
- Υπάρχουν άλλες στρατηγικές για επιλογή του pivot;
  - pivot = mid(A[1], A[n], A[n/2])
  - Επέλεξε τυχαία κάποιο στοιχείο του πίνακα

- Στην πράξη, για δεδομένα εισόδου μικρού μεγέθους το InsertionSort δουλεύει πιο αποδοτικά. Επομένως μια καλή στρατηγική θα ήταν να συνδυάσουμε τους δύο αλγόριθμους ώστε σε μικρούς πίνακες (π.χ. $n \leq 10$) να χρησιμοποιείται το InsertionSort και σε μεγάλους το Quicksort: στη μέθοδος Quicksort ανταλλάξτε την πρώτη γραμμή με την εξής:

```
if (j-i)<= 10 InsertionSort(A[i…j], j-i);
```

- Ακόμα ένας πιθανός τρόπος βελτίωσης του χρόνου εκτέλεσης είναι η χρήση στοίβας αντί αναδρομής.

Κάτω φράγμα για αλγόριθμους ταξινόμησης

- Ξέρουμε πως το πρόβλημα ταξινόμησης μπορεί να λυθεί σε χρόνο O(n log n) (QuickSort και MergeSort).
- Υπάρχει πιο αποδοτικός αλγόριθμος ταξινόμησης;
- Θα δείξουμε πως κάθε αλγόριθμος ταξινόμησης είναι Ω(n log n).
- Ως μονάδα μέτρησης αποδοτικότητας θα χρησιμοποιήσουμε τον *αριθμό συγκρίσεων* που απαιτεί κάποιος αλγόριθμος.
- Υποθέτουμε ότι κάθε στοιχείο του πίνακα που θέλουμε να ταξινομήσουμε είναι διαφορετικό από όλα τα άλλα.

Σειριακή διάταξη στοιχείων και ταξινόμηση

- Η συμπεριφορά ενός αλγόριθμου ταξινόμησης εξαρτάται μόνο από τη σχετική σειρά μεταξύ των στοιχείων που ταξινομούμε και όχι από τα συγκεκριμένα στοιχεία.
- Δηλαδή: αν *A* και *B* είναι δύο πίνακες τέτοιοι ώστε για κάθε i και j,

  $A[i] < A[j]$ αν και μόνο αν $B[i] < B[j]$,

  τότε ο αριθμός των βημάτων (όπως και ο αριθμός των συγκρίσεων) που θα εκτελέσει κάποιος αλγόριθμος με δεδομένο εισόδου *A* θα είναι ο ίδιος με τον ανάλογο αριθμό που θα εκτελέσει με δεδομένο εισόδου *B*.
- Άρα, η σειριακή διάταξη των στοιχείων του δεδομένου εισόδου *A*, $A[1], A[2], \ldots, A[n]$, έχει κύρια σημασία.
- Υπάρχουν n! 'διαφορετικές' τοποθετήσεις n ξεχωριστών στοιχείων. Άρα υπάρχουν n! διαφορετικά δεδομένα εισόδου.

Ανάλυση Αλγόριθμων Ταξινόμησης

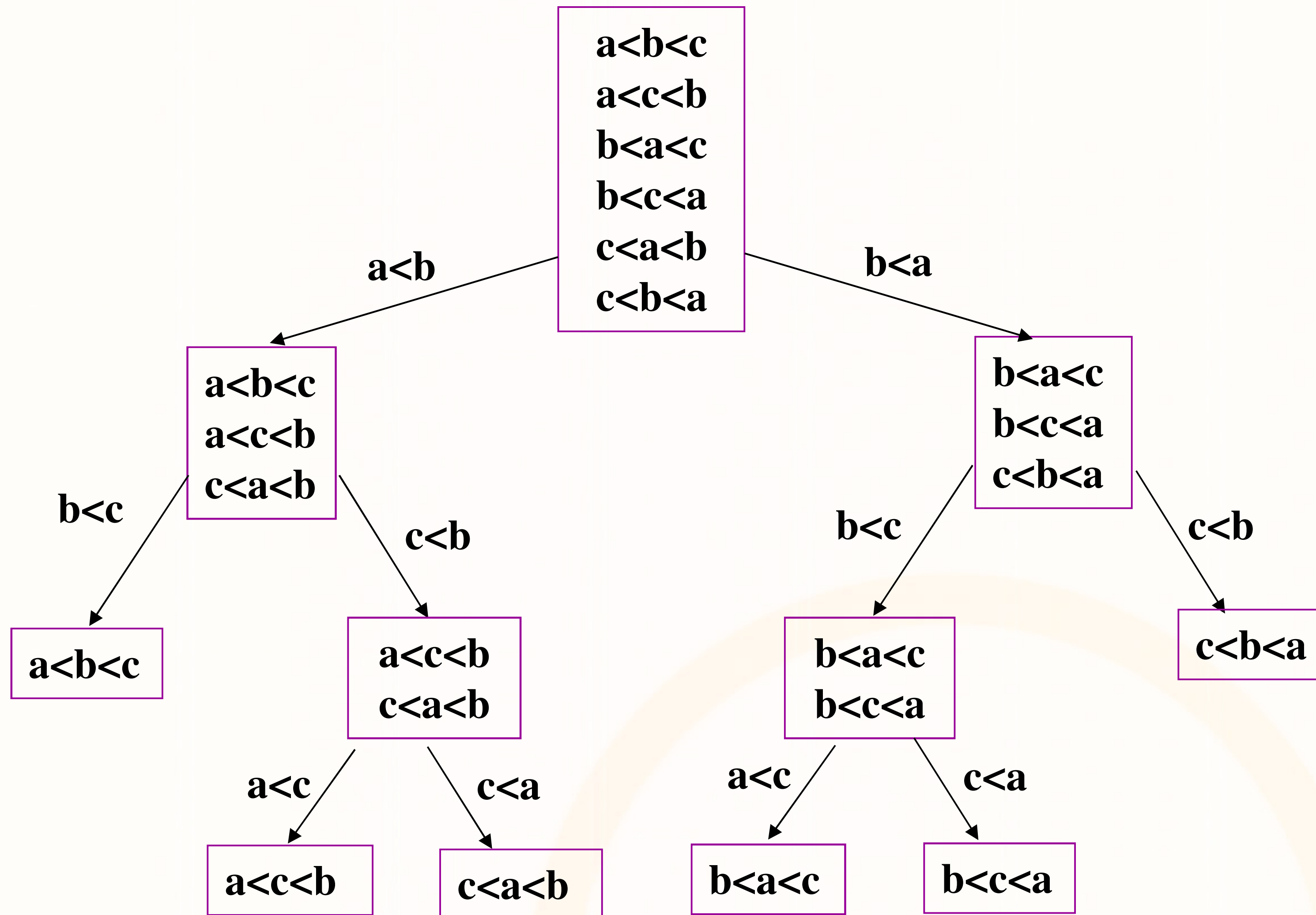
- Θα περιγράψουμε τη συμπεριφορά ενός αλγόριθμου ως ένα *δένδρο αποφάσεων* (decision tree).
- Στη ρίζα επιτρέπονται όλες οι διαφορετικές ‘σειρές’ των στοιχείων.

- Ας υποθέσουμε πως ο αλγόριθμος συγκρίνει τα δύο πρώτα στοιχεία $A[1]$ και $A[2]$. Τότε το αριστερό παιδί του δένδρου αντιστοιχεί στην περίπτωση $A[1] < A[2]$ και το δεξί παιδί της ρίζας στην περίπτωση $A[2] < A[1]$.

- Σε κάθε κόμβο, μια σειρά στοιχείων είναι νόμιμη αν ικανοποιεί όλες τις συγκρίσεις στο μονοπάτι από τη ρίζα στον κόμβο.

- Τα φύλλα αντιστοιχούν στον τερματισμό του αλγόριθμου και κάθε φύλλο περιέχει το πολύ μια νόμιμη σειρά στοιχείων.

Δένδρο αποφάσεων για 3 στοιχεία



Κάτω φράγμα

- Έστω *P* ένας αλγόριθμός ταξινόμησης, και έστω *T* το δένδρο αποφάσεων του *P* με δεδομένο εισόδου μεγέθους n.
- Ο αριθμός των φύλλων του *T* είναι n!
- Το ύψος του *T* είναι ένα κάτω φράγμα του χείριστου χρόνου εκτέλεσης του αλγόριθμου *P*.
- Ένα δυαδικό δένδρο ύψους d έχει το πολύ 2^d φύλλα.
- Άρα το *T* έχει ύψος το λιγότερο log n!
- **Συμπέρασμα: $P \in \Omega(\log n!) = \Omega(n \log n)$.**
- **Η μέση περίπτωση είναι επίσης n log n.**

Ε. Ταξινόμηση με Κάδους - BucketSort

- Έστω ότι ο πίνακας Α ***n*** στοιχείων περιέχει στοιχεία που ανήκουν στο διάστημα [***1..m***].

- Ο ***αλγόριθμος BucketSort*** βασίζεται πάνω στα ακόλουθα βήματα:
  1. Δημιουργούμε ένα πίνακα **buckets** μήκους **m** και θέτουμε **buckets[i]=0,** για όλα τα **i** (Αυτά τα είναι τα buckets - κάδοι)
  2. Διαβάζουμε τον πίνακα **Α** ξεκινώντας από το πρώτο στοιχείο. Αν διαβάσουμε το στοιχείο **α**, τότε αυξάνουμε την τιμή του **buckets[α]** κατά ένα. Επαναλαμβάνουμε το βήμα μέχρι το τελευταίο στοιχείο.
  3. Τέλος, διαβάζουμε γραμμικά τον πίνακα **buckets**, ο οποίος περιέχει αναπαράσταση του ταξινομημένου πίνακα, και θέτουμε τα στοιχεία του πίνακα **Α** με την ταξινομημένη ακολουθία.

BUCKETS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

BucketSort: Βασική Ιδέα

Δεδομένο Εισόδου: Τα στοιχεία είναι στο εύρος m=[0,14], n=8

1	11	1	7	2	14	7	1

0 ... n-1

Μετά την **1η** εκτέλεση του Bucketsort (Εισαγωγή του 1)

BUCKETS	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

m

Μετά τη **2η** εκτέλεση του Bucketsort (Εισαγωγή του 11)

BUCKETS	0	**1**	0	0	0	0	0	0	0	0	0	**1**	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

m

Μετά τη **3η** εκτέλεση του Bucketsort (Εισαγωγή του 1)

BUCKETS	0	**2**	0	0	0	0	0	0	0	0	0	**1**	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

m

Μετά την **8ή** εκτέλεση του Bucketsort

BUCKETS	0	**3**	**1**	0	0	0	0	**2**	0	0	0	**1**	0	0	**1**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

m

BucketSort: Υλοποίηση

```
// A: Πίνακας για ταξινόμηση μεγέθους n
// Buckets: Βοηθητικός πίνακας μεγέθους m
void BucketSort(int A[], int buckets[], int n, int m){
  int i, j, k=0;

  // Κατανομή στοιχείων στους σωστούς κάδους
  for (i=0; i<n; i++) {
    buckets[A[i]]++;
  }

  // Αντιγραφή στοιχείων από πίνακα BUCKETS
  // στον πίνακα A
  for (i=0; i<m; i++) {
    for (j=0; j<buckets[i]; j++) {
      A[k] = i;
    k++;
    }
  }
}
```

O(n)

O(n+m)

Συνολικά περνάμε 1 φορά από τα στοιχεία του πίνακα BUCKETS (m) και μια φορά από αυτά του A (n)

BucketSort: Χρόνος Εκτέλεσης

- **Ο αλγόριθμος BucketSort πετυχαίνει ταξινόμηση του Α σε χρόνο**

  Θ(n+m)

- **Σημαίνει ότι ο αλγόριθμος είναι καλύτερος από τον MergeSort Θ(nlogn);**
- Αυτό διαψεύδει το κάτω φράγμα Ω(n log n); (θυμειθείτε ότι έχουμε αποδείξει ότι όλοι οι αλγόριθμοι ταξινόμησης, με **δυαδική σύγκριση**, έχουν σαν κάτω φράγμα Ω(n log n))

- **ΟΧΙ,** γιατί το μοντέλο είναι διαφορετικό. Μέχρι τώρα υποθέσαμε ότι η μόνη πράξη που μπορούμε να εφαρμόσουμε στα δεδομένα είναι η δυαδική σύγκριση ή ανταλλαγή στοιχείων. Ο αλγόριθμος BucketSort όμως στο Βήμα 2 ουσιαστικά εφαρμόζει **m-αδική (m-ary) σύγκριση**, σε χρόνο Ο(1).

- Αυτό μας υπενθυμίζει πως σχεδιάζοντας ένα αλγόριθμο και λαμβάνοντας υπόψη κάποια αποδεδειγμένα κάτω φράγματα πρέπει πάντα να αναλύουμε το μοντέλο στο οποίο δουλεύουμε.

- Η ύπαρξη και αξιοποίηση περισσότερων πληροφοριών πιθανόν να επιτρέπουν τη δημιουργία αποδοτικότερων αλγορίθμων.

Έμμεση Ταξινόμηση (Indirect Sorting)

Έμμεση Ταξινόμηση (Indirect Sorting)

- Οι αλγόριθμου που έχουμε παρουσιάσει υποθέτουν πως οι πίνακες-δεδομένα εισόδου περιέχουν **ακέραιους αριθμούς** και προϋποθέτουν μετακίνηση των στοιχείων μέσα στον πίνακα.

  Τι γίνεται αν θέλουμε να ταξινομήσουμε αρχεία που περιέχουν πολύπλοκα αντικείμενα (π.χ. εγγραφές με πολλαπλές στήλες)?
- Σε τέτοιες περιπτώσεις η μετακίνηση (swapping) στοιχείων είναι δαπανηρή.
- Αυτό μπορεί να αποφευχθεί με τη ***χρήση δεικτών (C), αναφορών (JAVA)***: ως δεδομένο εισόδου χρησιμοποιούμε **πίνακα που περιέχει δείκτες** στα στοιχεία που θέλουμε να ταξινομήσουμε.
- Σύγκριση και ανταλλαγή γίνεται μεταξύ των δεικτών αντί μεταξύ των ιδίων των αντικειμένων. Έτσι έχουμε λιγότερη μετακίνηση στοιχείων.
- Οποιοσδήποτε από τους αλγόριθμους που αναφέραμε μπορεί να υλοποιηθεί με αυτό τον συλλογισμό χωρίς να αλλάξει ο αλγόριθμος.

Εξωτερική ταξινόμηση (External Sorting)

- **Τι γίνεται αν έχουμε 256MB RAM αλλά θέλουμε να ταξινομήσουμε ένα αρχείο με 800MB έγγραφες;**
- Η ταξινόμηση μπορεί να γίνεται κατά **τμήματα**:
  - Ένα μέρος του αρχείου μεταφέρεται στην κύρια μνήμη, ταξινομείται (**με ένα από τους αλγόριθμους που συζητήσαμε**) και αποθηκεύεται σε ένα προσωρινό αρχείο.
  - Το επόμενο τμήμα μεταφέρεται στην κύρια μνήμη και ταξινομείται και μετά συγχωνεύεται με το προσωρινό αρχείο.
  - Η διαδικασία επαναλαμβάνεται μέχρι εξάντλησης του αρχικού αρχείου.
- Με βάση αυτή την κύρια ιδέα υπάρχουν διάφοροι αλγόριθμοι εξωτερικής ταξινόμησης. Κύριος στόχος τους είναι η αποδοτική επεξεργασία της δευτερεύουσας μνήμης (δηλαδή του μαγνητικού δίσκου).
- **Τέτοιοι αλγόριθμοι βρίσκουν εφαρμογές σε Βάσεις Δεδομένων**