



Διάλεξη 19: Αλγόριθμοι Ταξινόμησης II

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Οι αλγόριθμοι ταξινόμησης:

- Γ. MergeSort** – Ταξινόμηση με Συγχώνευση

- Δ. BucketSort** – Ταξινόμηση με Κάδους

Διδάσκων: Παναγιώτης Ανδρέου

Γ. Ταξινόμηση με Συγχώνευση (Merge sort)

- Η **ταξινόμηση με συγχώνευση** είναι διαδικασία **διαίρει και βασίλευε** (**Divide and Conquer**: αναδρομική διαδικασία όπου το πρόβλημα μοιράζεται σε μέρη τα οποία λύνονται ξεχωριστά, και μετά οι λύσεις συνδυάζονται.)
- Περιγραφή του Mergesort
 1. **Διαίρεση**: Αναδρομικά μοιράζουμε τον πίνακα στα δύο μέχρι να φτάσουμε σε πίνακες μεγέθους ένα (**DIVIDE**)
 2. **Συγχώνευση**: Ταξινομούμε αναδρομικά τους πίνακες αυτούς με την συγχώνευση γειτονικών πινάκων (χρησιμοποιώντας βοηθητικό πίνακα). (**CONQUER**)

MergeSort: Βασική Ιδέα

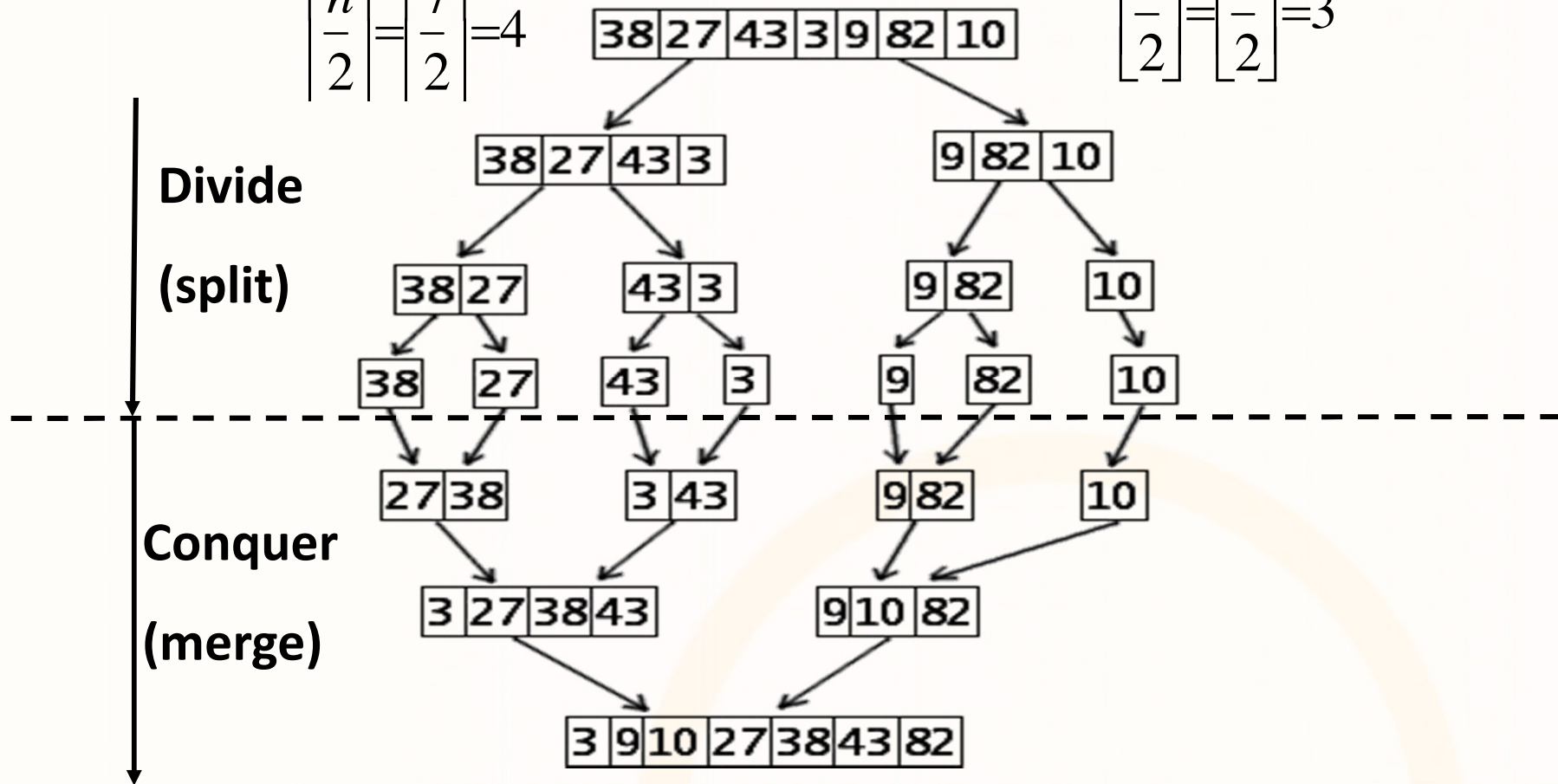
(#στοιχείων αριστερά)

$$\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{7}{2} \right\rfloor = 4$$

$n=7$

(#στοιχείων δεξιά)

$$\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{7}{2} \right\rceil = 3$$



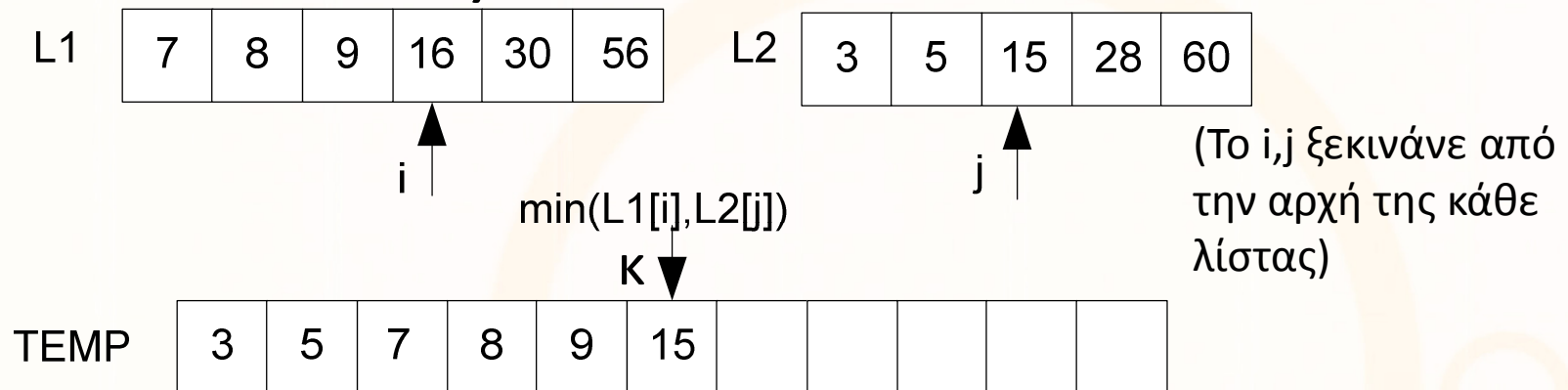
Συγχώνευση 2 Λιστών

Υποθέστε ότι θέλετε να συγχωνεύσετε 2 ταξινομημένες λίστες L1, L2 και να δημιουργήσετε μια νέα ταξινομημένη λίστα TEMP

Διαδικασία

1. Τοποθέτησε τους δείκτες i , j στην κεφαλή κάθε λίστας.
2. Διάλεξε το μικρότερο από την λίστα L1 και L2 και τοποθέτησε τον στον πίνακα TEMP στην θέση k
3. Προχώρησε τον δείκτη i (αν το μικρότερο στοιχείο ήταν από την λίστα L1) ή τον δείκτη j στην αντίθετη περίπτωση.
4. Επανάλαβε τα βήματα 2-4 μέχρι να εισαχθούν όλα τα στοιχεία στον TEMP

Μετά από 6 εκτελέσεις:



MergeSort: Πρόγραμμα στην C

```
void MergeSort(int A[], int temp[], int l, int r){  
  
    // η συνθήκη τερματισμού της ανάδρομης  
    if (l==r) return;  
    int mid = (l+r)/2;  
    // για ελαχιστοποίηση overflow(για μεγάλα l,r)  
    // int mid = l + ((r - l) / 2);  
  
    // μοιράζουμε αναδρομικά τον πίνακα  
    MergeSort(A, temp, l, mid);  
    MergeSort(A, temp, mid+1, r);  
  
    // συνεχίζεται
```

DIVIDE

MergeSort: Πρόγραμμα στην C (συν.)

```
// Τώρα οι πίνακες [1..mid] και [mid+1..r] είναι  
// ταξινομημένοι → Η διαδικασία συγχώνευσης
```

```
int k=1, i=1, j=mid+1;
```

```
// συγχώνευση στον temp μέχρι μια από  
// τις λίστες αδειάσει
```

```
while ((i<=mid) && (j<=r)) {  
    if (A[i]<A[j]){  
        temp[k] = A[i]; i++;  
    }  
    else {  
        temp[k] = A[j]; j++;  
    }  
    k++;  
}
```

MERGE

```
// συνεχίζεται
```

MergeSort: Πρόγραμμα στην C (συν.)

```
// copy όλων των υπόλοιπων στοιχείων λίστας L1
while (i<=mid) {
    temp[k] = A[i];
    k++;i++;
}

// copy όλων των υπόλοιπων στοιχείων λίστας L2
while (j<=r) {
    temp[k] = A[j];
    k++;j++;
}

// αντιγραφή όλων των στοιχείων από TEMP -> A
for (i=1; i<=r; i++) {
    A[i] = temp[i];
}
}
```

Παράδειγμα Εκτέλεσης Merge Sort

BEFORE: [8,4,8,43,3,5,2,1,10]
Index: 0 1 2 3 4 5 6 7 8

0,8: [8,4,8,43,3 | 5,2,1,10]

0,4: [8,4,8 | 43,3]

0,2: [8,4 | 8]

0,1: [8 | 4]

0,0: [8]

1,1: [4]

Merging: [A0,A0] [A1,A1] => [4,8,]

2,2: [8]

Merging: [A0,A1] [A2,A2] => [4,8,8,]

3,4: [43,3]

3,3: [43]

4,4: [3]

Merging: [A3,A3] [A4,A4] => [3,43]

Merging: [A0,A2] [A3,A4] => [3,4,8,8,43]

5,8: [5,2 | 1,10]

5,6: [5 | 2]

5,5: [5]

6,6: [2]

Merging: [A5,A5] [A6,A6] => [2,5]

7,8: [1 | 10]

7,7: [1]

8,8: [10]

Merging: [A7,A7] [A8,A8] => [1,10]

Merging: [A5,A6] [A7,A8] => [1,2,5,10]

Merging: [A0,A4] [A5,A8] => [1,2,3,4,5,8,8,10,43]

AFTER: [1,2,3,4,5,8,8,10,43]

MergeSort: Ανάλυση Χρ. Εκτέλ. - Αναδρομή

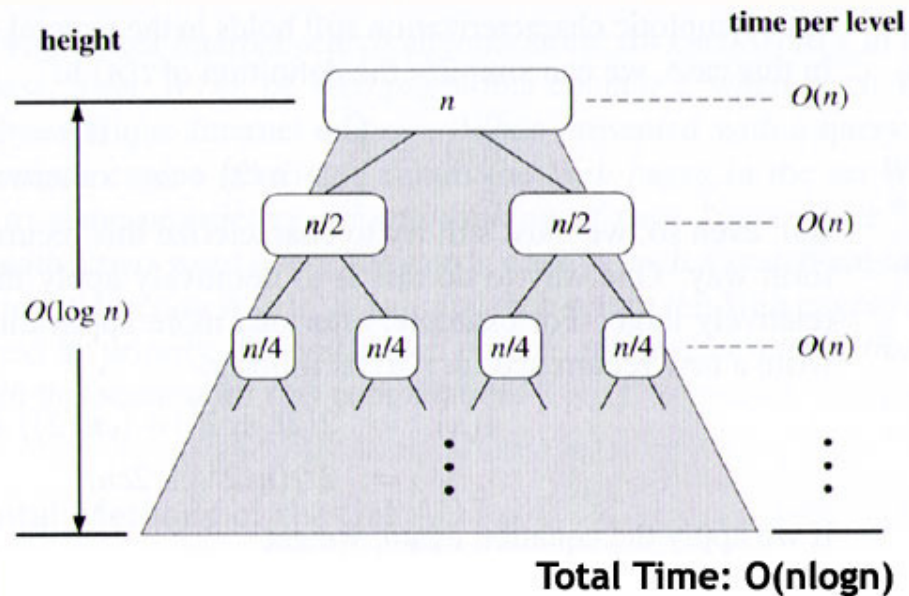
- Το πρόβλημα μοιράζει αναδρομικά σε δυο μέρη την λίστα που θέλουμε να ταξινομήσουμε.
- Όταν φτάσουμε στην λίστα που έχει μέγεθος 1 τότε σταματά η αναδρομή και το πρόγραμμα αρχίζει να συνδυάζει (merge) τις επιμέρους λίστες.
- Παρατηρούμε ότι πάνω σε μια λίστα μεγέθους N η αναδρομή εκτελείται $2\log_2 n$ φορές. Δηλαδή ο πίνακας μοιράζεται ως εξής: $n, n/2, n/4, \dots, 2, 1$

```
void MergeSort(int A[], int temp[], int l, int r){  
    if (l==r) return;  
    int mid = (l+r)/2;  
  
    MergeSort(A, temp, l, mid);  
    MergeSort(A, temp, mid+1, r);  
}
```

$\Theta(\log n)$

MergeSort: Ανάλυση Χρ. Εκτέλ. - Συγχώνευση

- Σε κάθε επίπεδο της ανάδρομης περνάμε μια φορά από το κάθε στοιχείο.
- Επομένως η συγχώνευση των στοιχείων σε κάθε επίπεδο της εκτέλεσης χρειάζεται **γραμμικό** χρόνο $\Theta(n)$.
- Σημειώστε ότι η διαδικασία απαιτεί τη χρήση βοηθητικού πίνακα.
- Μπορούμε να χρησιμοποιούμε τον ίδιο βοηθητικό πίνακα temp για όλες τις (αναδρομικές) κλήσεις του MergeSort.



MergeSort: Ανάλυση Χρ. Εκτέλ. - Συνολικός

- Η αντιγραφή και η συγχώνευση παίρνουν χρόνο $\Theta(n)$ και η αναδρομή παίρνει χρόνο $\Theta(\log n)$. Συνολικά $\Theta(n \log n)$.
- Ο χρόνος εκτέλεσης εκφράζεται και από την αναδρομική εξίσωση

$$T(0) = T(1) = 1$$

$$T(n) = 2 \cdot T(n/2) + n$$

- ... η οποία μπορεί να λυθεί με το Master Theorem ή με την μέθοδο της αντικατάστασης.
- **Πλεονέκτημα MergeSort:**
Ο συνολικός χρόνος εκτέλεσης είναι $\Theta(n \log n)$
(σε αντίθεση με το SelectionSort ($\Theta(n^2)$) και το InsertionSort ($O(n^2)$))
- **Μειονέκτημα:**
Απαιτεί τη χρήση βοηθητικού πίνακα (δηλαδή χρειάζεται διπλάσιο χώρο αποθήκευσης για την εκτέλεση του). Αυτό δεν καθιστά την μέθοδο πολύ εύχρηστη.

Δ. Ταξινόμηση με Κάδους - BucketSort

- Έστω ότι ο πίνακας A 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 **A** ξεκινώντας από το πρώτο στοιχείο. Αν διαβάσουμε το στοιχείο **a**, τότε αυξάνουμε την τιμή του **buckets[a]** 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 **A** με την ταξινομημένη ακολουθία.

BUCKETS	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 **1^η** εκτέλεση του BucketSort (Εισαγωγή του 1)

BUCKETS	0	1	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 **2^η** εκτέλεση του BucketSort (Εισαγωγή του 11)

BUCKETS	0	1	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 **3^η** εκτέλεση του BucketSort (Εισαγωγή του 1)

BUCKETS	0	2	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 **8^ή** εκτέλεση του BucketSort

BUCKETS	0	3	1	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: Πρόγραμμα στην C

```
// 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]]++;
}
```

$O(n)$

```
// 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++;
    }
}
```

$O(n+m)$

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 A σε χρόνο

$$\Theta(n+m)$$

- Σημαίνει ότι ο αλγόριθμος είναι καλύτερος από τον Mergesort $\Theta(n \log n)$;
(Κατ' ακρίβειαν μπορεί να αποδειχθεί ότι όλοι οι αλγόριθμοι ταξινόμησης, με **δυαδική σύγκριση**, έχουν σαν κάτω φράγμα $\Omega(n \log n)$)
- **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 $O(1)$.
- 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.