Το πρόβλημα της ταξινόμησης

Είσοδος: Μία ακολουθία αριθμών $< a_1,\ a_2,\ a_3,\ \cdots\ a_n >$.

Έξοδος: Μία μετάθεση $< a'_1,\ a'_2,\ a'_3,\ \cdots\ a'_n >$ της ακολουθίας εισόδου, έτσι ώστε τ $a'_1 \leq a'_2 \leq a'_3 \leq\ \cdots \leq a'_n$.

Παράδειγμα

$$< 6, 5, 3, 4, 1, 2 > \Longrightarrow \boxed{\text{ΤΑΞΙΝΟΜΗΣΗ}} \Longrightarrow < 1, 2, 3, 4, 5, 6 >$$

Με ποιο τρόπο γίνεται η ταξινόμηση;

Αλγόριθμος 1: *Insertion Sorting (‘ταξινόμηση μέσω εισαγωγής ’)*

- Επεξεργαζόμαστε έναν αριθμό της εισόδου κάθε φορά.
- Διατηρούμε μία ταξινομημένη ακολουθία των αριθμών που έχουμε ήδη επεξεργαστεί.
- **Εισάγουμε (insert)** τον αριθμό υπό επεξεργασία στη ‘σωστή’ θέση μέσα στην διατηρούμενη ταξινομημένη ακολουθία.

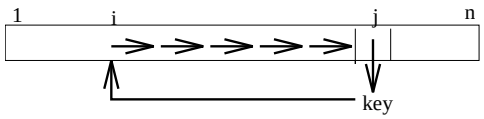
Παράδειγμα Είσοδος:$< 6, 5, 3, 4, 1, 2 >$

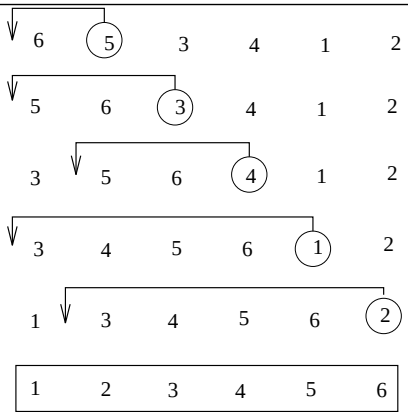
Αριθμός υπό επεξεργασία	Ταξινομημένη ακολουθία
6	<>
5	$< 6 >$
3	$< 5, 6 >$
4	$< 3, 5, 6 >$
1	$< 3, 4, 5, 6 >$
2	$< 1, 3, 4, 5, 6 >$
NIL	$< 1, 2, 3, 4, 5, 6 >$

Ερώτηση: Πόσο **‘καλή’** είναι η μέθοδος Insertion Sort;

‘Καλή’ συνήθως σημαίνει **γρήγορη!**

```
    Insertion_Sort(A)
1   for j ← 2 to length(A) do
2       key ← A[j]
3       /* Insert A[j] into the sorted sequence A[1...j − 1] */
4       i ← j − 1
5       while i > 0 and a[i] > key do
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← key
```





Σημείωση: Η μέθοδος Insertion sort εργάζεται **'τοπικά'** (in place).

	Insertion_Sort(A)	Κόστος	Επαναλήψεις
1	**for** $j \leftarrow 2$ **to** $length(A)$ **do**	c_1	n
2	$key \leftarrow A[j]$	c_2	$n-1$
3	/* ... */	0	
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ **and** $a[i] > key$ **do**	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

όπου, t_j είναι ο αριθμός των επαναλήψεων που εκτελείται το **while loop** για τη συγκεκριμένη τιμή j.

- $T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n}(t_j - 1)$

- Ο χρόνος εκτέλεσης running time είναι συνάρτηση του n, δηλαδή, του μεγέθους της εισόδου.
- Για εισόδους ίδιου μεγέθους, ο χρόνος εκτέλεσης μπορεί να είναι διαφορετικός.

$$\boxed{T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)}$$

- Εάν η είσοδος είναι ήδη ταξινομημένη, $t_j = 1$, φορ $j = 2, 3, \cdots, n$.

$$\begin{aligned} \Longrightarrow \; T(n) = & \quad c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5(n-1) + 0 \\ = & \quad c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1) \\ = & \quad (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

[γραμμική ως προς το n]

- Εάν η είσοδος είναι ταξινομημένη σε φθίνουσα σειρά, $t_j = j$, φορ $j = 2, 3, \cdots, n$.

$$\begin{aligned} \Longrightarrow \; T(n) = & \quad c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) + (c_6 + c_7)\tfrac{n(n-1)}{2} \\ & \quad \vdots \\ = & \quad \tfrac{(c_5+c_6+c_7)}{2} n^2 + (c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8)n \\ & \quad -(c_2 + c_4 + c_5 + c_8) \\ = & \quad an^2 + bn + c \quad \text{όπου } a, b, c \text{ είναι σταθερές} \end{aligned}$$

[τετραγωνική ως προς το n]

Στο μάθημα αυτό μας ενδιαφέρει η **'τάξη της αύξησης '** του χρόνου εκτέλεσης.

(order of growth)

Χείριστη πολυπλοκότητα (Worst case analysis)

- Παρέχει ένα άνω φράγμα για τον απαιτούμενο χρόνο εκτέλεσης για **οποιαδήποτε** είσοδο.
- Για αρκετούς αλγόριθμους η χείριστη πολυπλοκότητα παρουσιάζεται πολύ συχνά (πχ. αλγόριθμοι αναζήτησης).
- Μερικές φορές η **μέση πολυπλοκότητα** είναι τόσο 'άσχημη' όσο η χείριστη.

Μέση πολυπλοκότητα (Average case complexity)

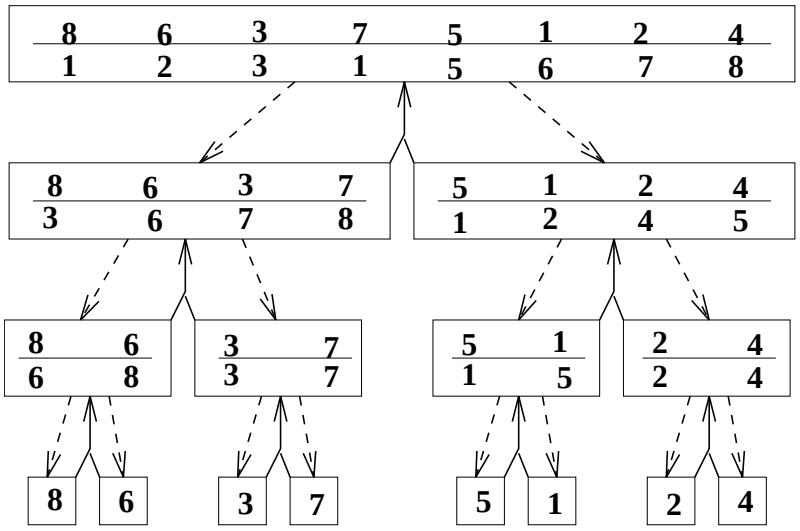
- Υποθέτει ότι όλες οι είσοδοι συγκεκριμένου μεγέθους είναι ισοπίθανες.
- Για τη μέθοδο insertion sort: $t_j = j/2 \implies$
  $T(n) = \cdots = a'n^2 + b'n + c' = \Theta(n^2)$

  Η χείριστη πολυπλοκότητα για τη μέθοδο insertion sort είναι επίσης $\Theta(n^2)$.

Αλγόριθμος 2: *Merge Sort (‘ταξινόμηση μέσω συγχώνευσης ’)*

- Χώρισε την είσοδο σε δύο ‘ίσα’ μέρη.
- **Αναδρομικά** ταξινόμησε κάθε ένα από αυτά.
- **Συγχώνευσε** τα δύο ήδη ταξινομημένα μέρη.

Παράδειγμα



Πως γίνεται η συγχώνευση;

Είσοδος: 2 ταξινομημένες λίστες A και B με a και b στοιχεία η καθεμία, αντίστοιχα.

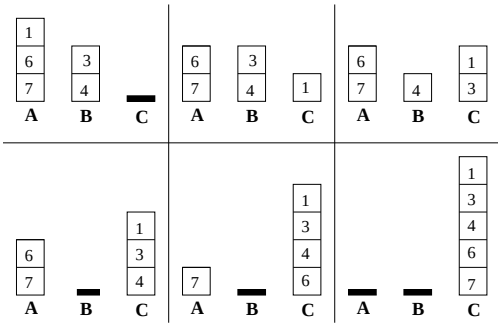
Έξοδος: Μια ταξινομημένη λίστα C η οποία περιέχει όλα τα στοιχεία των A και B. Το μέγεθος της λίστας C είναι $n = a + b$.

Merge(A, B, C)

while υπάρχουν ακόμη στοιχεία στην A ή την B **do**

- Σύγκρινε τα ‘πρώτα’ στοιχείο της A και της B.
- Τοποθέτησε το μικρότερο από τα δύο στο ‘τέλος’ της λίστας C.

Παράδειγμα



Πολυπλοκότητα: $\Theta(n)$

Merge-Sort(A, p, r)

if $p < r$ **then**

- $q \leftarrow \lfloor (p + r)/2 \rfloor$
- Merge-Sort(A, p, q)
- Merge-Sort$(A, q + 1, r)$
- Merge(A, p, q, r)

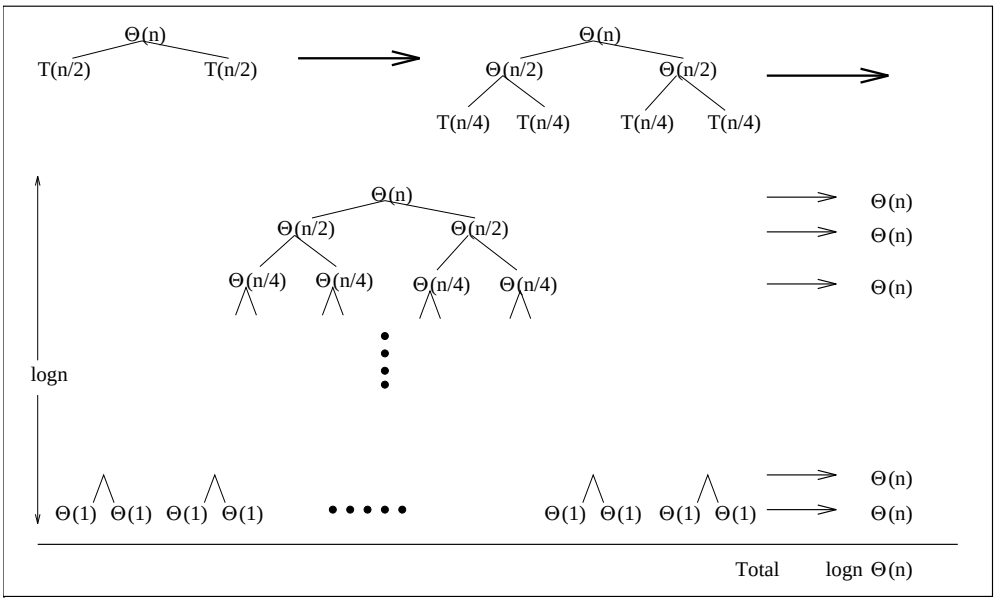
Ανάλυση

- Εάν το μέγεθος της εισόδου είναι μικρό, δηλαδή, $n \leq c$ για κάποια σταθερά c, τότε το πρόβλημα λύνεται σε σταθερό χρόνο. **Πολυπλοκότητα** $\Theta(1)$.
- Έστω $T(n)$ ο χρόνος που χρειάζεται για την ταξινόμηση λίστας μεγέθους n.
- Έστω $C(n)$ ο χρόνος που χρειάζεται για τη συγχώνευση 2 λιστών συνολικού μεγέθους n. Γνωρίζουμε ότι $C(n) = \Theta(n)$.
- Έστω ότι το πρόβλήμα μπορεί να διαιρεθεί σε 2 υπο-προβλήματα σε σταθερό χρόνο, και ότι $c = 1$. Τότε:

$$T(n) = \begin{cases} \Theta(1) & \text{εάν } n \leq 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{εάν } n > 1 \end{cases}$$

Ερώτηση: Υπάρχει κλειστός (μη-αναδρομικός) τύπος για το $T(n)$;

Υποθέτουμε ότι $n = 2^k$ (ή, $\log n = k$).



$\Longrightarrow T(n) = \Theta(n \log n)$

Παρατηρήσεις:

- Για τη μέθοδο merge-sort ισχύει:

$$\left.\begin{array}{l}\text{Χείριστη πολυπλοκότητα χρόνου}\\ \text{Μέση πολυπλοκότητα χρόνου}\\ \text{Βέλτιστη πολυπλοκότητα χρόνου}\end{array}\right\} = \Theta(n \log n)$$

- Για μικρές τιμές του n, η μέθοδος insertion-sort μπορεί να είναι γρηγορότερη από τη μέθοδο merge-sort.

Μας ενδιαφέρει η επίδοση των αλγορίθμων όταν το μέγεθος της εισόδου $n \longrightarrow \infty$.

Η ανάλυση αυτού του είδους ονομάζεται **ασυμπτωτική.**

Στο μάθημα των αλγορίθμων ...

- Θα μελετήσουμε διαφορετικούς αλγόριθμους για το ίδιο πρόβλημα σε σχέση με την ασυμπτωτική τους συμπεριφορά.

Υπό αυτή τη συνθήκη,

Ένας	$\Theta(n)$	αλγόριθμος είναι καλύτερος από ένα
	$\Theta(n \log n)$	αλγόριθμο , ο οποίος είναι καλύτερος από ένα
	$\Theta(n \log^2 n)$	′′
	$\Theta(n^2)$	′′
	$\Theta(n^3)$	′′
	$\Theta(2^n)$	′′
	$\Theta(n2^n)$	′′
	$\Theta(n!)$	′′
	$\Theta(2^{2^n})$	′′