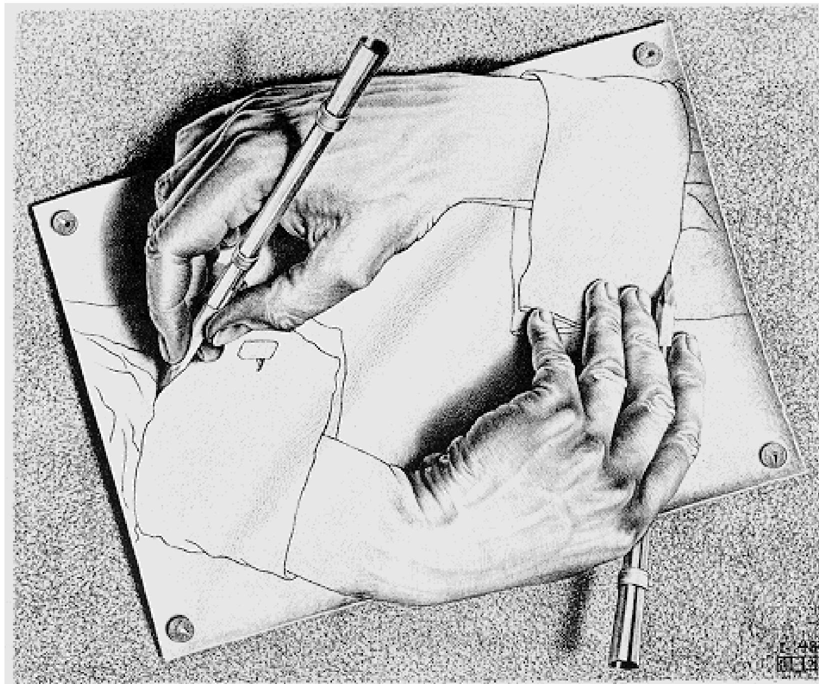
Αναδρομή

Μιχάλης Δρακόπουλος



Αναδρομικές συναρτήσεις

Η αναδρομή είναι μια αλγοριθμική μέθοδος για την υπολογιστική επίλυση προβλημάτων. Στηρίζεται στην αρχή του 'διαίρει και βασίλευε'.

Η αναδρομή βασίζεται:

1. Στον διαμερισμό του αρχικού προβλήματος σε ένα ή περισσότερα ανεξάρτητα υποπροβλήματα που έχουν την ίδια δομή με το αρχικό πρόβλημα και είναι κάπως ευκολότερο να επιλυθούν.
2. Στη συνέχεια, χρησιμοποιώντας την ίδια τεχνική διάσπασης, καθένα από αυτά τα υποπροβλήματα διαιρείται σε νέα υποπροβλήματα τα οποία είναι ακόμα λιγότερο πολύπλοκα.
3. Τελικά, τα υποπροβλήματα γίνονται τόσο απλά που μπορούν να λυθούν χωρίς περαιτέρω υποδιαίρεση, και η πλήρης λύση προσδιορίζεται από τη σύνθεση των λύσεων των υποπροβλημάτων.

Μια αναδρομική συνάρτηση καλεί *τον εαυτό της* (άμεσα ή έμμεσα).

Σημειώστε ότι η αναδρομή δεν είναι σε καμία περίπτωση η μοναδική μέθοδος επίλυσης κάποιου προβλήματος. Οποιοδήποτε πρόβλημα επιλύεται αναδρομικά, μπορεί επίσης να επιλυθεί *επαναληπτικά*

(με έναν βρόχο επανάληψης). Στην πραγματικότητα, οι αναδρομικοί αλγόριθμοι είναι συνήθως λιγότερο αποτελεσματικοί από τους επαναληπτικούς αλγόριθμους. Αυτό συμβαίνει επειδή η διαδικασία κλήσης μιας συνάρτησης εξαρτάται από την εκτέλεση μιας σειράς υπολογιστικών ενεργειών. Οι ενέργειες αυτές περιλαμβάνουν τη δέσμευση μνήμης για τις παραμέτρους και τις τοπικές μεταβλητές και την αποθήκευση της διεύθυνσης της θέσης του προγράμματος, στην οποία θα επιστρέψει ο έλεγχος μετά τον τερματισμό της συνάρτησης. Αυτές οι ενέργειες, οι οποίες συχνά αναφέρονται ως *overhead* (δηλαδή επιπρόσθετη επιβάρυνση), λαμβάνουν χώρα με κάθε κλήση συνάρτησης. Το overhead αυτό δεν υπάρχει στην επαναληπτική επίλυση.

Μερικά προβλήματα επαναλαμβανόμενης δομής, όμως, επιλύονται ευκολότερα με αναδρομή από ότι με έναν βρόχο. Ενώ ο βρόχος πλεονεκτεί σε ταχύτητα εκτέλεσης, η αναδρομή διευκολύνει τη γρηγορότερη ανάπτυξη ενός αναδρομικού αλγορίθμου από την πλευρά του προγραμματιστή. Για παράδειγμα, πολλά μαθηματικά προβλήματα είναι διατυπωμένα με αναδρομικές σχέσεις.

Το overhead της αναδρομής μειώνεται σημαντικά αν διατυπώσουμε τον αναδρομικό αλγόριθμο με *αναδρομή ουράς (tail recursion)*. Η τεχνική αυτή δεν περιλαμβάνεται στην ύλη του μαθήματος και δεν θα καλυφθεί στις σημειώσεις.

Υπολογισμός $n! = 1 \times 2 \times 3 \times \cdots \times n$

Η επαναληπτική συνάρτηση είναι:

```python
def fact_i(n):
    # Με επανάληψη
    p = 1
    for i in range(1, n+1):
        p *= i
    return p
```

Ο υπολογισμός μπορεί να διατυπωθεί και αναδρομικά:

$$n! = n \times (n-1)!, \quad 0! = 1$$

και η αντίστοιχη αναδρομική συνάρτηση:

```python
def fact_r(n):
    # Με αναδρομή
    if n == 0:
        return 1
    return n * fact_r(n-1)
```

Σε κάθε αναδρομικό αλγόριθμο διακρίνουμε:

τη βασική περίπτωση το βήμα που τερματίζει την αναδρομή (εδώ ο υπολογισμός $0!$)

το αναδρομικό βήμα που υλοποιεί την αναδρομή: `n * fact_r(n-1)`

Η ακολουθία των παραμέτρων (`n`, `n-1` κλπ) πρέπει να συγκλίνει στη βασική περίπτωση (το `n` θα γίνει τελικά 0).

Για παράδειγμα, ο υπολογισμός του $5!$ γίνεται καλώντας διαδοχικά τις `fact_r(5)`, `fact_r(4)` ... που δεν μπορούν όμως να ολοκληρωθούν προτού φτάσουμε στην κλήση της `fact_r(0)` που υλοποιεί τη βασική περίπτωση.

```
fact_r(5)
    fact_r(4)
        fact_r(3)
            fact_r(2)
                fact_r(1)
                    fact_r(0)
                    return 1
                return 1*1 = 1
            return 2*1 = 2
        return 3*2 = 6
    return 4*6 = 24
return 5*24 = 120
```

Αφού υπολογιστεί η βασική περίπτωση μπορούν να υπολογιστούν διαδοχικά οι τιμές των `fact_r(1)`, `fact_r(2)` ... επιστρέφοντας στα προηγούμενα επίπεδα μέχρι να καταλήξουμε στον τελικό υπολογισμό της `fact_r(5)`.

Μέγιστος Κοινός Διαιρέτης - Αλγόριθμος του Ευκλείδη

Ο αλγόριθμος του Ευκλείδη για τον υπολογισμό του Μέγιστου Κοινού Διαιρέτη δυο ακεραίων x, y, έστω $M = \gcd(x, y)$, στηρίζεται στην παρατήρηση ότι αν ο M διαιρεί τους x, y θα διαιρεί και το ακέραιο υπόλοιπο της διαίρεσης $r = x \bmod y$.

Ο επαναληπτικός αλγόριθμος του Ευκλείδη υλοποιείται από τα εξής βήματα:

Βήμα 1 Υπολόγισε $r = x \bmod y$.

Βήμα 2 Αν $r = 0$, τότε $\gcd(x, y) = y$

Βήμα 3 Αν $r \neq 0$, εκτέλεσε το **Βήμα 1** με $x = y$ και $y = r$.

```python
def gcd_i(x, y):
    # Επαναληπτικός αλγόριθμος του Ευκλείδη
    while y != 0:
        x, y = y, x%y
    return x
```

Ο αλγόριθμος του Ευκλείδη μπορεί να εκφραστεί και *αναδρομικά*:

$$\gcd(x, y) = \begin{cases} x & \text{αν } y = 0 \text{ (βασική περίπτωση)} \\ \gcd(y, x \bmod y) & \text{διαφορετικά (αναδρομικό βήμα)} \end{cases}$$

που ”μεταφράζεται” άμεσα στον αντίστοιχο αναδρομικό κώδικα:

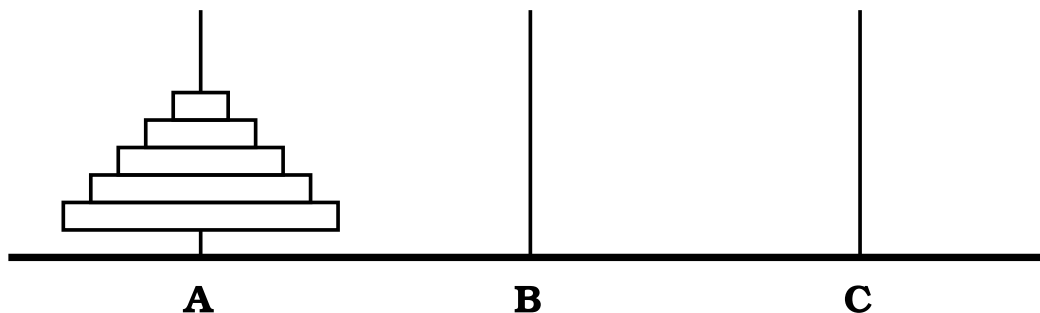
```python
def gcd_r(x, y):
    # Αναδρομικός αλγόριθμος Ευκλείδη
    if y == 0:
        return x
    return gcd_r(y, x%y)
```

Οι πύργοι του Hanoi

Είναι ένας μαθηματικός γρίφος μετακίνησης n δίσκων διαφορετικών διαμέτρων που βρίσκονται αρχικά στοιβαγμένοι σε έναν στύλο A, σε έναν *διαφορετικό* στύλο προορισμού B χρησιμοποιώντας βοηθητικά έναν τρίτο στύλο C, ακολουθώντας τους παρακάτω κανόνες:

- Μόνο ένας δίσκος μπορεί να μετακινείται κάθε φορά.
- Μπορεί να μετακινηθούν μόνο δίσκοι που βρίσκονται στην κορυφή μιας στοίβας.
- Ένας δίσκος μπορεί να τοποθετηθεί είτε σε άδειο στύλο είτε πάνω σε μεγαλύτερο δίσκο.

Παράδειγμα για $n = 5$:



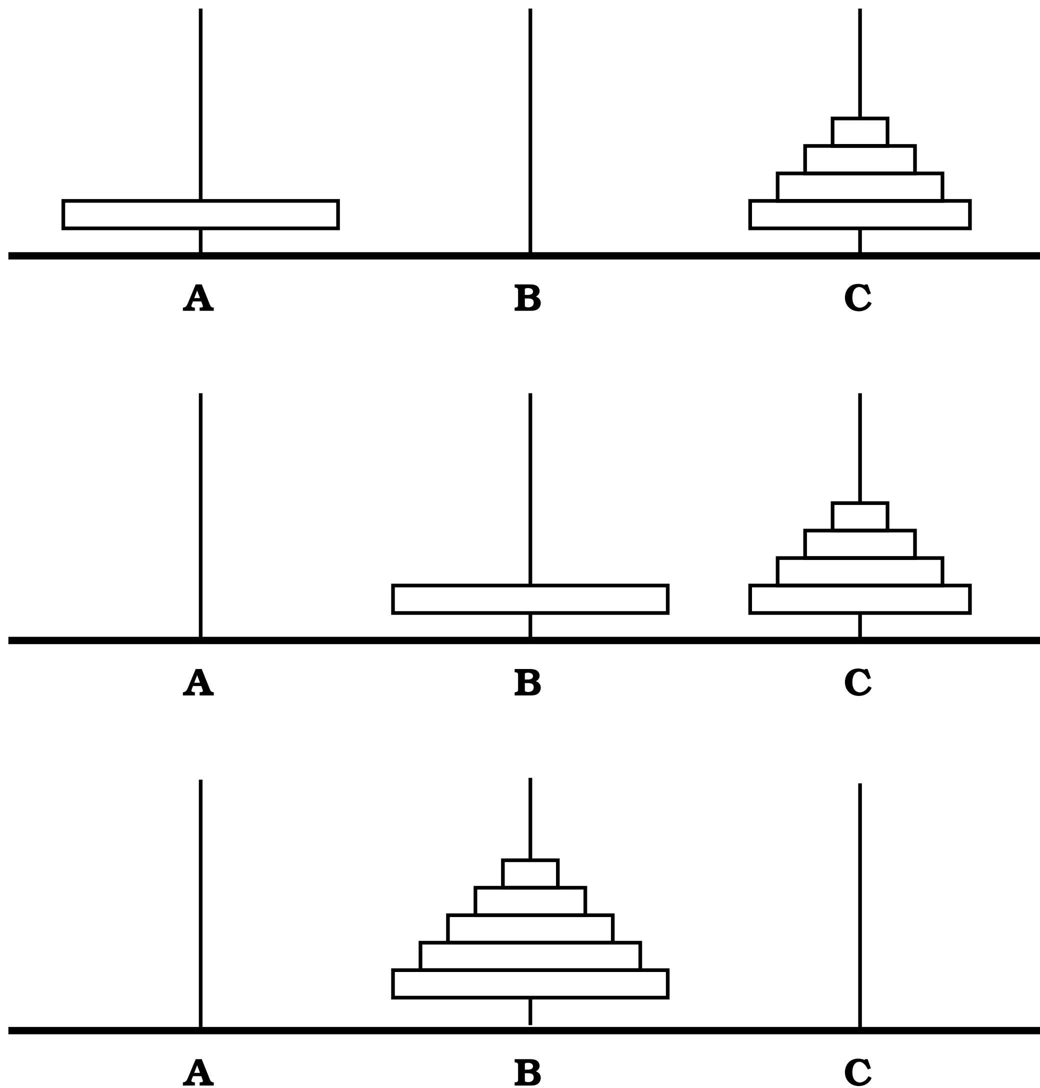
Θέλουμε να γράψουμε ένα πρόγραμμα που να επιλύει το πρόβλημα με τις λιγότερες δυνατές μετακινήσεις. Η έξοδος του προγράμματος θα μπορούσε να είναι π.χ. ”Πάρε ένα δίσκο από τον στύλο A και τοποθέτησέ τον στον στύλο B” ή για συντομία ”`A -> B`”, και να ακολουθήσουμε μια σειρά από τέτοιες οδηγίες για να γίνει η μετακίνηση.

Το πρόβλημα μπορεί να λυθεί *αναδρομικά*!

Έστω ότι μπορούμε να μετακινήσουμε με *μια κίνηση* $n-1$ δίσκους. Τότε μπορούμε:

1. να μεταφέρουμε $n-1$ δίσκους από τον αρχικό στύλο A στον βοηθητικό στύλο C
2. να μεταφέρουμε τον μοναδικό δίσκο στον στύλο A στην τελική του θέση στον στύλο B.
3. να μεταφέρουμε $n-1$ δίσκους από τον βοηθητικό στύλο C στον τελικό στύλο B.

Παρατηρούμε ότι στα βήματα 1 και 3 στην ουσία έχουμε να επιλύσουμε το ίδιο πρόβλημα, αλλά για $n-1$ δίσκους.



Αν χρησιμοποιήσουμε τις μεταβλητές `start`, `finish` και `temp` για τον αρχικό, τελικό και βοηθητικό στύλο αντίστοιχα, έχουμε τον εξής αλγόριθμο:

- Αν `n == 1` μετακίνησε τον δίσκο από `start` σε `finish` (βασική περίπτωση)
- Αν `n > 1` διαίρεσε το πρόβλημα σε 3 υποπροβλήματα:
    1. Με τον ίδιο αλγόριθμο μετακίνησε τους πάνω `n-1` δίσκους από `start` σε `temp`. Στο βήμα αυτό ο στύλος `finish` χρησιμοποιείται ως βοηθητικός.
    2. Μετακίνησε τον τελευταίο δίσκο από `start` σε `finish`.
    3. Μετακίνησε τους πάνω `n-1` δίσκους από `temp` σε `finish`. Στο βήμα αυτό ο στύλος `start` χρησιμοποιείται ως βοηθητικός.

Η αντίστοιχη αναδρομική συνάρτηση:

```
def move_tower(n, start, finish, temp):
    if n == 1:
        print(f'{start} -> {finish}')
    else:
        move_tower(n-1, start, temp, finish)
        print(f'{start} -> {finish}')
        move_tower(n-1, temp, finish, start)
```

Και θα την καλούσαμε π.χ. για $n = 10$:

```
move_tower(10, 'A', 'B', 'C')
```

Ο θρύλος των πύργων του Hanoi

Το τέλος του κόσμου θα έρθει όταν μια ομάδα μοναχών καταφέρει να μετακινήσει 64 χρυσούς δίσκους σε 3 διαμαντένους στύλους.

- Έστω T_n το πλήθος των μετακινήσεων που χρειάζονται για την μετακίνηση n δίσκων. Τότε:

  $$T_n = 2T_{n-1} + 1 \text{ για } n > 1, \quad \text{με } T_1 = 1$$

  Αποδεικνύεται *επαγωγικά* ότι:

  $$T_n = 2^n - 1$$

- Αν οι μοναχοί μετακινούν 1 δίσκο ανά δευτερόλεπτο, τότε χρειάζονται:
  - για 20 δίσκους, περισσότερο από 1 βδομάδα
  - για 30 δίσκους, περισσότερα από 31 χρόνια
  - για 40 δίσκους, 348 αιώνες
  - για 64 δίσκους, περισσότερο από 584 δισεκατομμύρια χρόνια!
- Υπολογιστής που εκτελεί 10^9 πράξεις το δευτερόλεπτο χρειάζεται αιώνες!

Προβλήματα με αναδρομή

- Η παράλειψη της βασικής περίπτωσης, π.χ. αναδρομικός υπολογισμός $n!$

  ```
  def factorial(n):
      return n*factorial(n-1)
  ```

  Εξαιτίας της παράλλειψης της βασικής περίπτωσης έχουμε ”υπερχείλιση στοίβας (stack overflow error)” και η κλήση `factorial(1)` για παράδειγμα, οδηγεί σε 988 αναδρομικές κλήσης και η συνάρτηση τερματίζει με το μήνυμα:

  ```
    [Previous line repeated 988 more times]
  RecursionError: maximum recursion depth exceeded
  ```

- Ακόμα και σωστά γραμμένη αναδρομική συνάρτηση (με βασική περίπτωση), π.χ. υπολογισμός n-στου αρμονικού αριθμού $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$:

```python
def harmonic(n):
    if n == 1:
        return n
    return 1/n + harmonic(n-1)
```

θα εξαντλήσει τη μνήμη που είναι αφιερωμένη στην υλοποίηση των αναδρομικών κλήσεων και θα έχουμε πάλι stack overflow error. Έτσι η κλήση `harmonic(992)` θα τερματίσει επίσης με το μήνυμα:

```
  [Previous line repeated 988 more times]
RecursionError: maximum recursion depth exceeded
```

Αντίθετα ο αντίστοιχος *επαναληπτικός* αλγόριθμος

```python
def harm(n):
    H = 0
    for k in range(1, n+1):
        H += 1/k
    return H
```

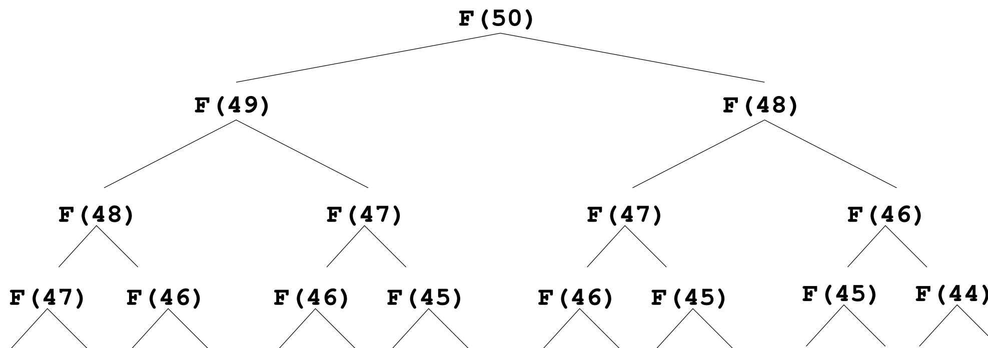
χρειάζεται ελάχιστη μνήμη και ”δουλεύει” για πολύ μεγάλα n χωρίς πρόβλημα.

- Περιττή επανάληψη υπολογισμών. Π.χ. ο αναδρομικός αλγόριθμος για αριθμούς Fibonacci

$$F_n = F_{n-1} + F_{n-1},\, n > 2 \quad \text{με } F_1 = 1, F_2 = 2$$

```python
def F(n):
    if n == 1 or n == 2:
        return n
    return F(n-1) + F(n-2)
```

Οι αναδρομικές κλήσεις της `F` έχουν τη δομή δυαδικού δέντρου. Για τον υπολογισμό F_{50} με αναδρομή:



Τελικά η `F(1)` θα κληθεί $F_{50} = 20.365.011.074$ φορές.

Ταξινόμηση με συγχώνευση (merge sort)

Η αναδρομή είναι σημαντική στην ανάπτυξη αλγορίθμων ταξινόμησης, γιατί είναι ευκολότερο να προγραμματιστεί από τον αντίστοιχο επαναληπτικό αλγόριθμο.

Η βασική ιδέα είναι ότι μπορούμε να συγχωνεύσουμε 2 ήδη ταξινομημένες λίστες `u` και `v` σε μια νέα λίστα `w` που περιέχει ταξινομημένα συνολικά τα στοιχεία των `u` και `v`.

Έστω `u = [20, 30, 50, 65]` και `v = [10, 30, 45, 75, 80]`. Θέλουμε να πάρουμε τη συνδυασμένη, ταξινομημένη λίστα:

`w = [10, 20, 30, 30, 45, 50, 65, 75, 80]`

Αρχικά η `w` είναι κενή. Ένα βήμα της διαδικασίας συγχώνευσης περιλαμβάνει τη σύγκριση της μικρότερης “εναπομείνασας” τιμής στη `u` με τη μικρότερη εναπομείνασα τιμή στη `v` και την αποθήκευση της μικρότερης από τις δύο στη λίστα `w`. Παρακολουθούμε τη διαδικασία με τους δείκτες `i` και `j` που διατρέχουν τις λίστες `u` και `v` αντίστοιχα. Αν το `u[i]` είναι η επόμενη τιμή που θα επιλεχθεί από το `u`, ο δείκτης `i` μετατοπίζεται μια θέση δεξιότερα. Αντίστοιχα για τα `v` και `j`. Δηλαδή οι μεταβλητές `i` και `j` παραπέμπουν στο πού βρισκόμαστε στις λίστες `u` και `v` σε κάθε βήμα. Οταν εξαντληθούν τα στοιχεία μιας από τις 2 λίστες, τα εναπομείναντα στοιχεία της άλλης επισυνάπτονται στη `w`.

`u`	`i`	`v`	`j`	`min(u[i],v[j])`	`w`
20, 30, 50, 65	0	**10**, 30, 45, 75, 80	0	10	`10`
20, 30, 50, 65	0	~~10~~, **30**, 45, 75, 80	1	20	`10, 20`
~~20~~, **30**, 50, 65	1	~~10~~, **30**, 45, 75, 80	1	30	`10, 20, 30`
~~20~~, **30**, 50, 65	1	~~10, 30~~, **45**, 75, 80	2	30	`10, 20, 30, 30`
~~20, 30~~, **50**, 65	2	~~10, 20~~, **45**, 75, 80,	2	45	`10, 20, 30, 30, 45`
~~20, 30~~, **50**, 65	2	~~10, 30, 45~~, **75**, 80	3	50	`10, 20, 30, 30, 45, 50`
~~20, 30, 50~~, **65**	3	~~10, 30, 45~~, **75**, 80	3	65	`10, 20, 30, 30, 45, 50, 65`
~~20, 30, 50, 65~~	-	~~10, 30, 45~~, **75, 80**	3	75, 80	`10, 20, 30, 30, 45, 50, 65, 75, 80`

Η συνάρτηση `merge` χρησιμοποιεί τα παραπάνω για να κατασκευάσει την ταξινομημένη λίστα `w` συγχωνεύοντας δυο ταξινομημένες λίστες `u` και `v`.

```python
def merge(u, v):
    # Συγχώνευση 2 ταξινομημένων λιστών
    w = []
    i = j = 0
    while i < len(u) and j < len(v):
        if u[i] < v[j]:
            w.append(u[i])
            i += 1
        else:
            w.append(v[j])
            j += 1
    w += u[i:]
    w += v[j:]
    return w
```

Αν έχουμε τη δυνατότητα συγχώνευσης 2 ταξινομημένων λιστών, μπορούμε να κατασκευάσουμε τον

αναδρομικό αλγόριθμο συγχώνευσης. Υποδιαιρούμε συνεχώς την αρχική λίστα μέχρις ότου καταλήξουμε σε λίστες μήκους 1 (βασική περίπτωση). Μια λίστα με ένα μόνο στοιχείο είναι διατεταγμένη. Στην συνέχεια συνδυάζουμε με τη `merge` διαδοχικά επι μέρους διατεταγμένες λίστες μέχρι να καταλήξουμε στο ζητούμενο.

Η ταξινόμηση γίνεται με τη συνάρτηση `mergesort` που υποδιαιρεί *αναδρομικά* τη λίστα εισόδου και στη συνέχεια κάθε βήμα συγχωνεύει μερικώς ταξινομημένες λίστες χρησιμοποιώντας τη `merge` μέχρι να ολοκληρωθεί η διάταξη της λίστας εισόδου.

```
def mergesort_r(x):
    # Αναδρομική ταξινόμηση με συγχώνευση
    n = len(x)
    if n == 1:
        y = x
    else:
        m = n // 2
        y1 = mergesort_r(x[:m])
        y2 = mergesort_r(x[m:])
        y = merge(y1, y2)
    return y
```