Εισαγωγή στον Προγραμματισμό Η/Υ

Διδάσκοντες: Ι. Πολενάκης, Κ. Σκιαδόπουλος, Δ. Μουρατίδου

Τμήμα Πληροφορικής, Ιόνιο Πανεπιστήμιο

7. Αναδρομή

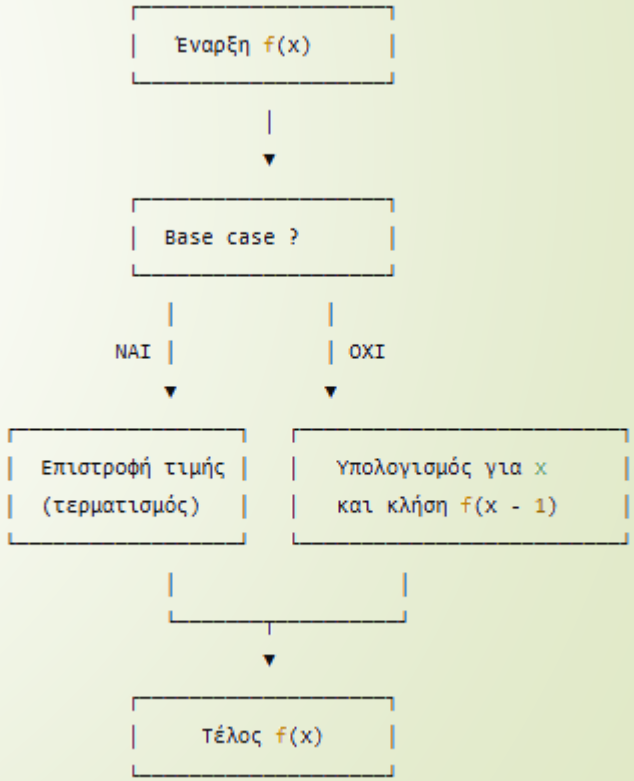
- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

- Κάθε αναδρομική συνάρτηση χρειάζεται:

  - **Recursive Case (Αναδρομική Κλήση)**
    Η περίπτωση όπου το πρόβλημα μειώνεται και η συνάρτηση καλεί τον εαυτό της..

  - **Base Case (Συνθήκη Τερματισμού)**
    Η πιο απλή περίπτωση του προβλήματος όπου η συνάρτηση *δεν καλεί* τον εαυτό της.



7. Αναδρομή

- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

- Χρησιμοποιείται όταν ένα πρόβλημα μπορεί να «σπάσει» σε μικρότερα, όμοια υποπροβλήματα.

- Χρειάζεται πάντα **συνθήκη τερματισμού**.
  - Αποτρέπει την άπειρη αναδρομή.
  - Είναι η απλούστερη περίπτωση του προβλήματος (base case).
  - Χωρίς αυτήν θα έχουμε RecursionError.
  - **Παράδειγμα:** Αν υπολογίζω το παραγοντικό *n!*, η πιο απλή περίπτωση είναι: $0! = 1$

- Ιδανική για: δέντρα, δομές, μαθηματικά μοτίβα.

7. Αναδρομή

- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

- Αναδρομή vs Επανάληψη
  - Η αναδρομή είναι συχνά πιο κομψή και πιο κοντά στον τρόπο που περιγράφουμε λογικά το πρόβλημα.
  - Η επανάληψη (loops) είναι συνήθως πιο αποδοτική σε μνήμη και ταχύτητα.
  - Η Python έχει όριο βάθους αναδρομής (περίπου 1000 επίπεδα).
  - Η αναδρομή μπορεί να αντικατασταθεί με while/for loops στις περισσότερες περιπτώσεις.

7. Αναδρομή

- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

Πότε χρησιμοποιούμε αναδρομή;

- Όταν το πρόβλημα είναι φυσικά αναδρομικό.

Παραδείγματα:

- Δέντρα (traversal)
- Αναζήτηση σε φακέλους
- Πύργοι του Ανόι
- Διαίρεση και κατάκτηση (merge sort, quicksort)

7. Αναδρομή

- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

Πότε χρησιμοποιούμε αναδρομή;

- Όταν το πρόβλημα είναι φυσικά αναδρομικό.

Κίνδυνοι & Συμβουλές

Προσοχή:

- Έλλειψη συνθήκης τερματισμού
- Πολλές αναδρομικές κλήσεις → μεγάλη χρήση μνήμης

Συμβουλή:

- Να ξεκινάς γράφοντας πρώτα το “base case”
- Έπειτα την αναδρομική σχέση

7. Αναδρομή

- **Αναδρομή**

Η αναδρομή είναι τεχνική κατά την οποία μια συνάρτηση καλεί **τον εαυτό της** για να λύσει μικρότερες εκδοχές του ίδιου προβλήματος.

<u>Πότε χρησιμοποιούμε αναδρομή;</u>

- Όταν το πρόβλημα είναι φυσικά αναδρομικό.

Κίνδυνοι & Συμβουλές

<u>Προσοχή</u>:

- Έλλειψη συνθήκης τερματισ
- Πολλές αναδρομικές κλήσ

<u>Συμβουλή</u>:

- Να ξεκινάς γράφοντας πρώτα
- Έπειτα την αναδρομική σχέση

- Η αναδρομή είναι ισχυρό εργαλείο.
- Χρειάζεται σωστή συνθήκη τερματισμού.
- Έχει περιορισμούς στη μνήμη.
- Απαραίτητη σε πολλές αλγοριθμικές τεχνικές.

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Παραγοντικό**

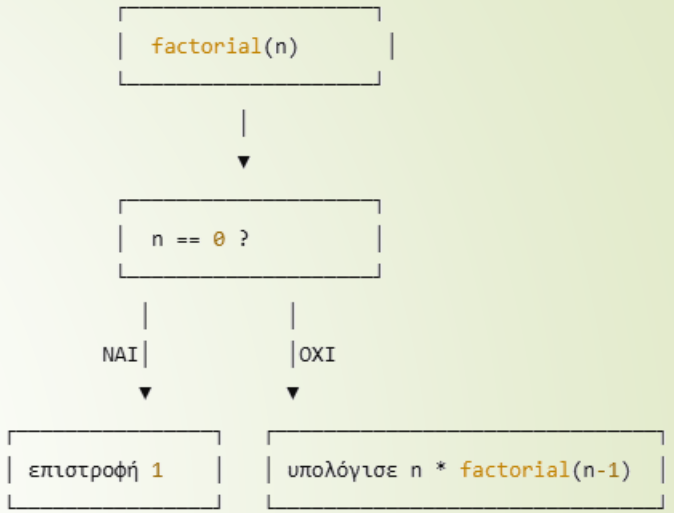
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
factorial(4)
= 4 * factorial(3)
            = 3 * factorial(2)
                        = 2 * factorial(1)
                                    = 1 * factorial(0)
                                                = 1
-----------------------------------------------------
= 24
```

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Παραγοντικό**

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```




```
factorial(4)
= 4 * factorial(3)
            = 3 * factorial(2)
                        = 2 * factorial(1)
                                    = 1 * factorial(0)
                                                = 1
-------------------------------------------------------
= 24
```

```
factorial(4)
    |
4 * factorial(3)
    |
3 * factorial(2)
    |
2 * factorial(1)
    |
1 * factorial(0)
    |
    1   ← base case
```

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Παραγοντικό**

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
factorial(4)
= 4 * factorial(3)
            = 3 * factorial(2)
                        = 2 * factorial(1)
                                    = 1 * factorial(0)
                                                = 1
------------------------------------------------------------
= 24
```

στοίβα κλήσεων **τη στιγμή που "βαθαίνει"** η αναδρομή.

Φάση 1 — Κλήση factorial(4)
TOP → | factorial(4) |

Φάση 2 — factorial(4) καλεί factorial(3)
TOP → | factorial(3) |
| factorial(4) |

Φάση 3 — factorial(3) → factorial(2)
TOP → | factorial(2) |
| factorial(3) |
| factorial(4) |

Φάση 4 — factorial(2) → factorial(1)
TOP → | factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

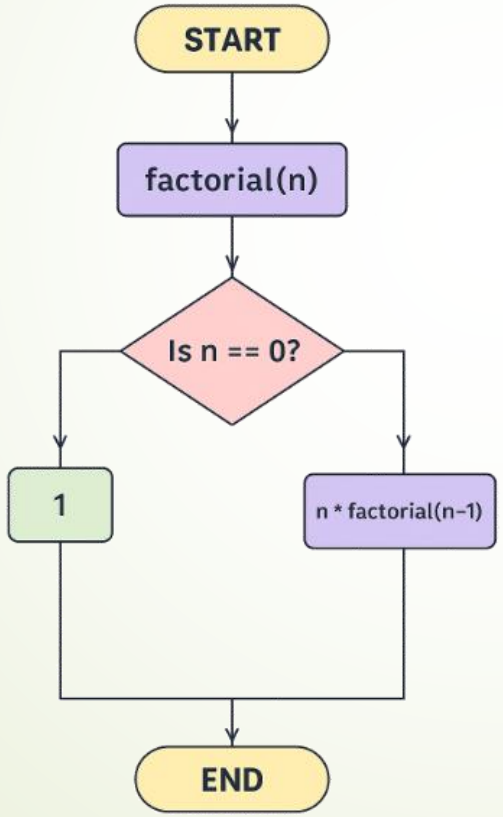
Φάση 5 — factorial(1) → factorial(0)
TOP → | factorial(0) |
| factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

Base case — factorial(0) επιστρέφει 1 και απομακρύνεται από τη στοίβα
TOP → | factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

και συνεχίζεται μέχρι να καθαρίσει η στοίβα.

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Παραγοντικό**



στοίβα κλήσεων **τη στιγμή που "βαθαίνει"** η αναδρομή.

Φάση 1 — Κλήση factorial(4)
TOP → | factorial(4) |

Φάση 2 — factorial(4) καλεί factorial(3)
TOP → | factorial(3) |
| factorial(4) |

Φάση 3 — factorial(3) → factorial(2)
TOP → | factorial(2) |
| factorial(3) |
| factorial(4) |

Φάση 4 — factorial(2) → factorial(1)
TOP → | factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

Φάση 5 — factorial(1) → factorial(0)
TOP → | factorial(0) |
| factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

Base case — factorial(0) επιστρέφει 1 και απομακρύνεται από τη στοίβα
TOP → | factorial(1) |
| factorial(2) |
| factorial(3) |
| factorial(4) |

και συνεχίζεται μέχρι να καθαρίσει η στοίβα.

7. Αναδρομή

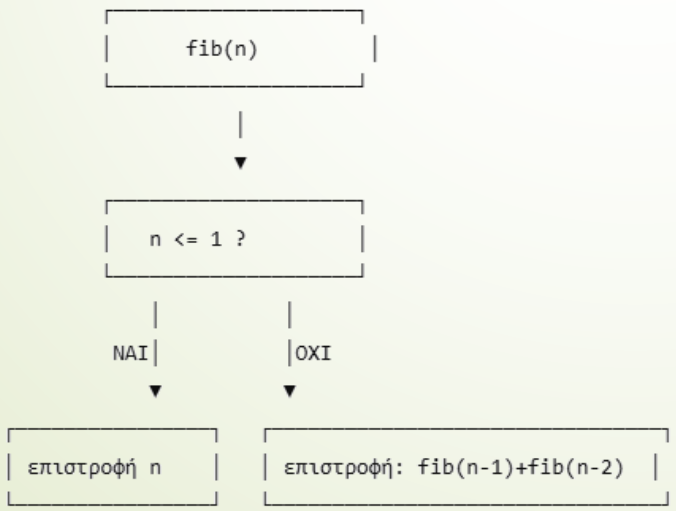
- **Παράδειγμα Εφαρμογής: Fibonacci**

Αναλυτική Αναδρομική Δομή

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Ο αριθμός κλήσεων αυξάνεται πολύ:

- η Fibonacci αναδρομή είναι αργή!



7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Fibonacci**

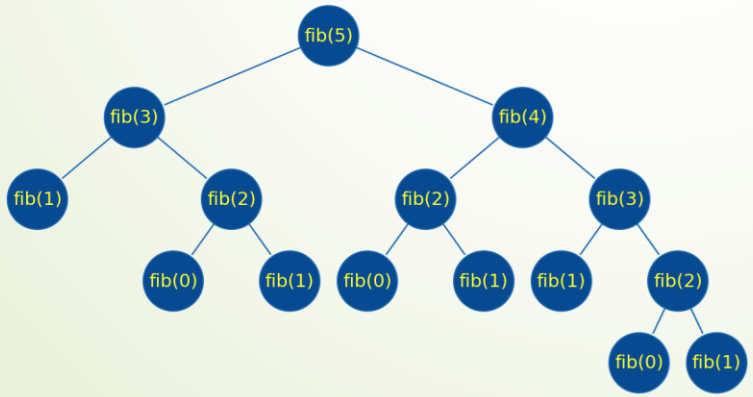
Αναλυτική Αναδρομική Δομή

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

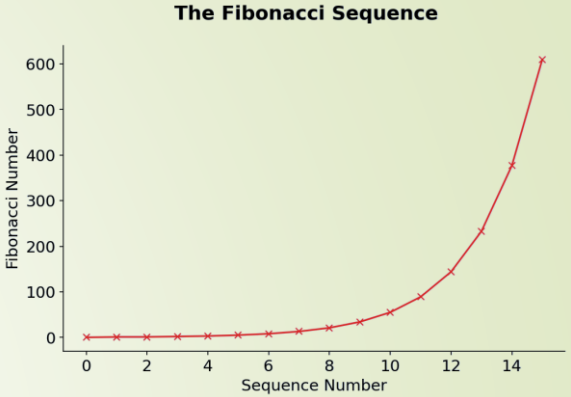
Ο αριθμός κλήσεων αυξάνεται πολύ:

- η Fibonacci αναδρομή είναι αργή!

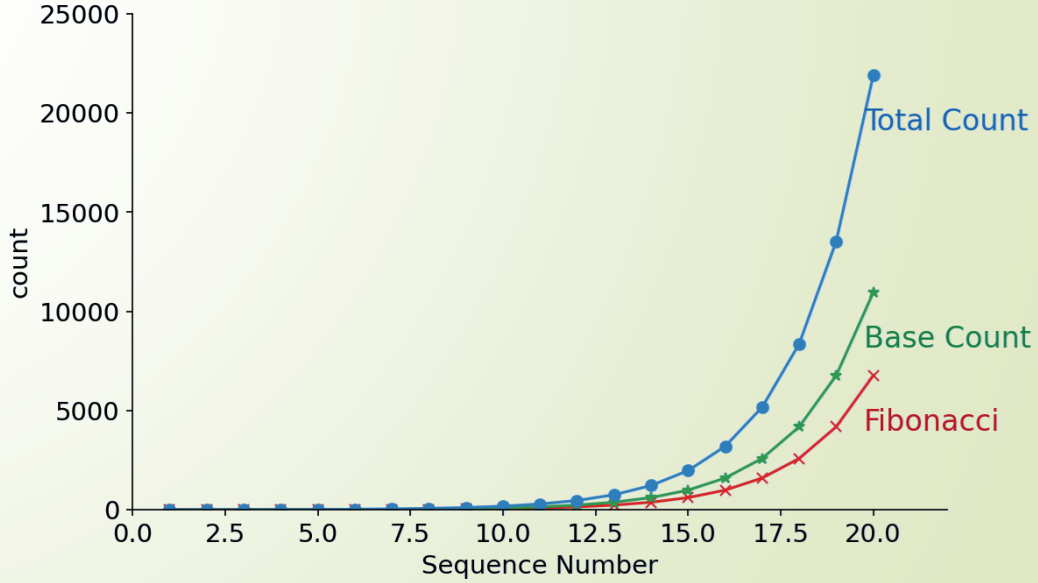
Recursive calls to calculate the fifth Fibonacci number



Index	Fibonacci	Base Count	Total Count
1	1	1	1
2	1	2	3
3	2	3	5
4	3	5	9
5	5	8	15
6	8	13	25
7	13	21	41
8	21	34	67
9	34	55	109
10	55	89	177
11	89	144	287
12	144	233	465
13	233	377	753
14	377	610	1,219
15	610	987	1,973
16	987	1,597	3,193
17	1,597	2,584	5,167
18	2,584	4,181	8,361
19	4,181	6,765	13,529
20	6,765	10,946	21,891



Function calls to calculate Fibonacci sequence recursively



7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Fibonacci**

- **Fibonacci** με **Δυναμικό Προγραμματισμό** (Bottom-Up)

- **Ιδέα**: Φτιάχνουμε έναν πίνακα dp όπου: dp[i] = fib(i)
  και τα υπολογίζουμε από το 0 προς τα πάνω, αποφεύγοντας την αναδρομή.

```
def fib_dp(n):
    dp = [0] * (n + 1)
    if n > 0:
        dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Πίνακας Υπολογισμού για n = 6

i :	0	1	2	3	4	5	6
dp:	0	1	1	2	3	5	8

```
dp[0] = 0
dp[1] = 1

for i from 2 to n:
    dp[i] = dp[i-1] + dp[i-2]
```

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Fibonacci**

Πίνακας Υπολογισμού για n = 6

i :	0	1	2	3	4	5	6
dp:	0	1	1	2	3	5	8

- **Fibonacci** με **Δυναμικό Προγραμματισμό** (Bottom-Up)
- **Ιδέα**: Φτιάχνουμε έναν πίνακα dp όπου: dp[i] = fib(i)
  και τα υπολογίζουμε από το 0 προς τα πάνω, αποφεύγοντας την αναδρομή.

```
def fib_dp(n):
    dp = [0] * (n + 1)
    if n > 0:
        dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

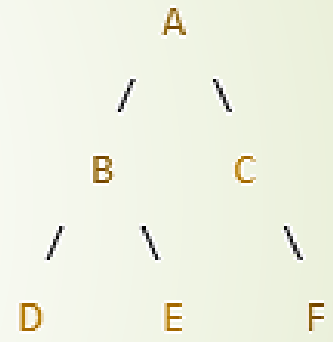
```
dp[0] = 0
dp[1] = 1

for i from 2 to n:
    dp[i] = dp[i-1] + dp[i-2]
```

```
dp[0] → 0
dp[1] → 1
dp[2] → dp[1] + dp[0] = 1
dp[3] → dp[2] + dp[1] = 2
dp[4] → dp[3] + dp[2] = 3
dp[5] → dp[4] + dp[3] = 5
dp[6] → dp[5] + dp[4] = 8
```

7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Preorder Διάσχιση Δέντρου**

```
def preorder(node):
    if not node:
        return
    print(node.value)
    preorder(node.left)
    preorder(node.right)
```



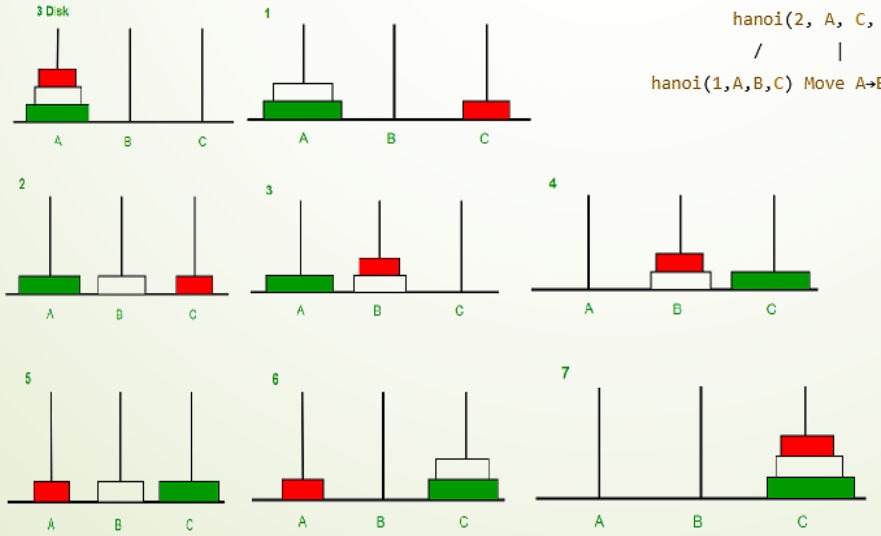
A → B → D → E → C → F

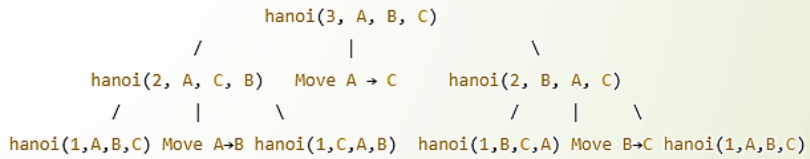
7. Αναδρομή

- **Παράδειγμα Εφαρμογής: Πύργοι του Ανόι (Hanoi Towers)**

```
def hanoi(n, start, mid, end):
    if n == 1:
        print(f"Move {start} -> {end}")
        return
    hanoi(n-1, start, end, mid)
    print(f"Move {start} -> {end}")
    hanoi(n-1, mid, start, end)
```

```
Push hanoi(3)
 Push hanoi(2)
  Push hanoi(1)
  Pop
 Move A→B
  Push hanoi(1)
  Pop
Pop hanoi(2)

Move A→C

 Push hanoi(2)
   Push hanoi(1)
   Pop
 Move B→C
   Push hanoi(1)
   Pop
 Pop hanoi(2)

Pop hanoi(3)

Empty stack
```





1. Αρχική κλήση
TOP → | hanoi(3, A, B, C) |
---

2. Καλεί hanoi(2, A, C, B)
TOP → | hanoi(2, A, C, B) |
| hanoi(3, A, B, C) |
---

3. Καλεί hanoi(1, A, B, C)
TOP → | hanoi(1, A, B, C) |
| hanoi(2, A, C, B) |
| hanoi(3, A, B, C) |

Base case → γίνεται το move:
Move A → C
Pop:
TOP → | hanoi(2, A, C, B) |
| hanoi(3, A, B, C) |
---

4. Επιστρέφουμε σε hanoi(2...) → move A → B
(move A → B)
5. Καλεί hanoi(1, C, A, B)
Push:
TOP → | hanoi(1, C, A, B) |
| hanoi(2, A, C, B) |
| hanoi(3, A, B, C) |

Base case → move:
(move C → B)
Pop:
TOP → | hanoi(2, A, C, B) |
| hanoi(3, A, B, C) |
---

hanoi(2,...) τελειώνει → pop:
TOP → | hanoi(3, A, B, C) |
---

6. Επιστρέφουμε σε hanoi(3...) → move A → C
(move A → C)
7. Καλεί hanoi(2, B, A, C)
Push:
TOP → | hanoi(2, B, A, C) |
| hanoi(3, A, B, C) |
---

8. Καλεί hanoi(1, B, C, A)
Push:
TOP → | hanoi(1, B, C, A) |
| hanoi(2, B, A, C) |
| hanoi(3, A, B, C) |

Base case → move:
(move B → A)
Pop stack:
TOP → | hanoi(2, B, A, C) |
| hanoi(3, A, B, C) |
---

9. Move B → C
(move B → C)
10. Καλεί hanoi(1, A, B, C)
Push:
TOP → | hanoi(1, A, B, C) |
| hanoi(2, B, A, C) |
| hanoi(3, A, B, C) |

Base → move:
(move A → C)
Pop:
TOP → | hanoi(2, B, A, C) |
| hanoi(3, A, B, C) |

Pop:
TOP → | hanoi(3, A, B, C) |
---

11. Τελικό Pop — Stack empty
TOP → [empty stack]

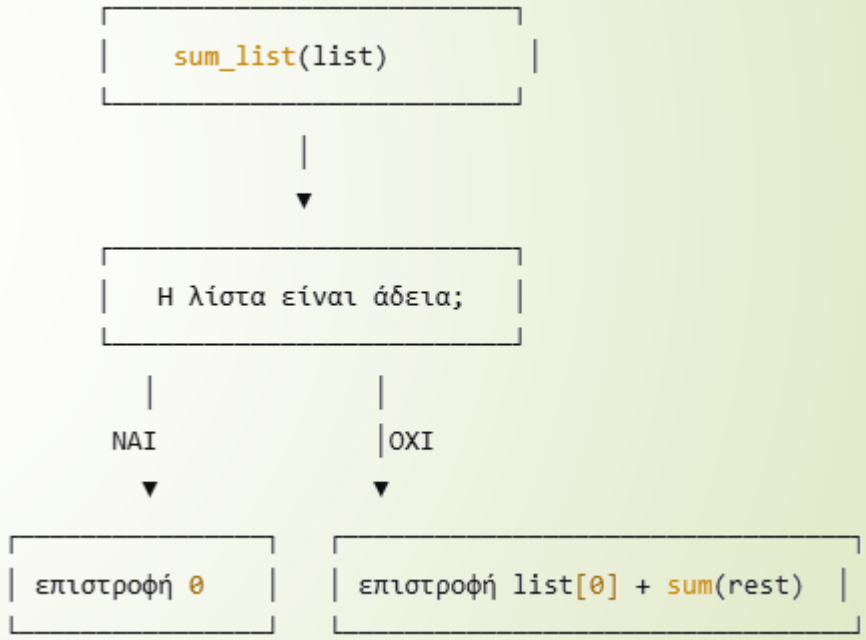
7. Αναδρομή

- **Παράδειγμα Εφαρμογής: …Lists**

Άθροισμα Λίστας

```
def sum_list(lst):
    if not lst:
        return 0
    return lst[0] + sum_list(lst[1:])
```

```
sum_list([3,1,4])
= 3 + sum_list([1,4])
            = 1 + sum_list([4])
                        = 4 + sum_list([])
                                    = 0
```



7. Αναδρομή

- **Παράδειγμα Εφαρμογής: …Lists**

Άθροισμα Λίστας

```
def sum_list(lst):
    if not lst:
        return 0
    return lst[0] + sum_list(lst[1:])
```

```
sum_list([5, 2, 1])
  /          \
5 +      sum_list([2, 1])
           /        \
         2 +     sum_list([1])
                     |
                 1 + sum_list([])
                     |
                     0  ← base case
```

```
sum_list([3,1,4])
= 3 + sum_list([1,4])
          = 1 + sum_list([4])
                    = 4 + sum_list([])
                              = 0
```

Μέγιστο Λίστας

```
def max_rec(lst):
    if len(lst) == 1:
        return lst[0]
    m = max_rec(lst[1:])
    return lst[0] if lst[0] > m
else m
```

```
max_rec([3,8,2])
→ compare 3 with max_rec([8,2])
                    ↓
          compare 8 with max_rec([2])
                              ↓
                    base case → 2
-----------------------------------------
max = 8
```

7. Αναδρομή

Ασκήσεις…

- **Άσκηση 1:** Γράψε αναδρομική συνάρτηση που υπολογίζει το άθροισμα των αριθμών από 1 έως n.
  Υπόδειξη: `sum(n) = n + sum(n-1)`

7. Αναδρομή

Ασκήσεις…

- **Άσκηση 1:** Γράψε αναδρομική συνάρτηση που υπολογίζει το άθροισμα των αριθμών από 1 έως n.
  Υπόδειξη: `sum(n) = n + sum(n-1)`

```
def sum_to_n(n):
    if n == 0:
        return 0
    return n + sum_to_n(n-1)
```

7. Αναδρομή

Ασκήσεις…

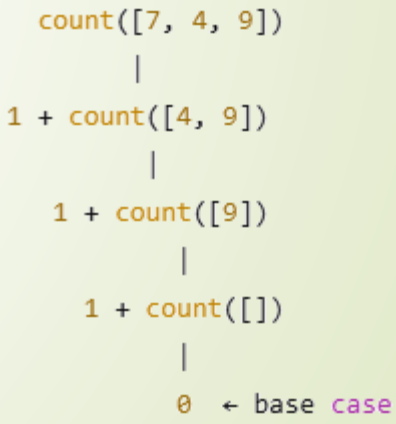
- **Άσκηση 2:** Γράψε αναδρομική συνάρτηση που μετράει πόσα στοιχεία έχει μια λίστα.

7. Αναδρομή

Ασκήσεις…

- **Άσκηση 2:** Γράψε αναδρομική συνάρτηση που μετράει πόσα στοιχεία έχει μια λίστα.

```
def count(lst):
    if not lst:
        return 0
    return 1 + count(lst[1:])
```



7. Αναδρομή

Ασκήσεις…

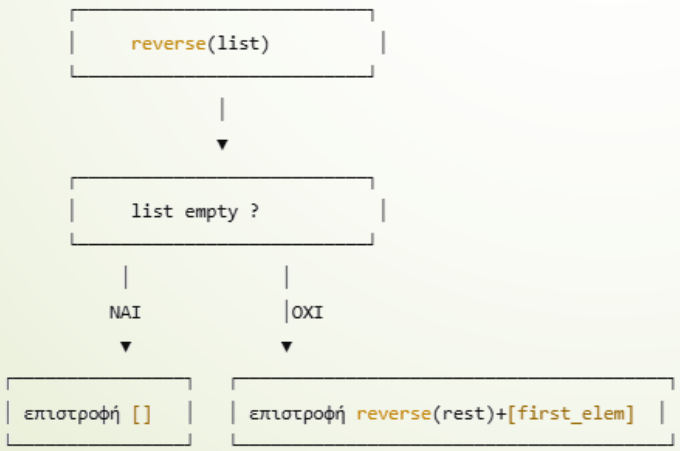
- **Άσκηση 3:** Γράψε αναδρομική συνάρτηση που αντιστρέφει μια λίστα.

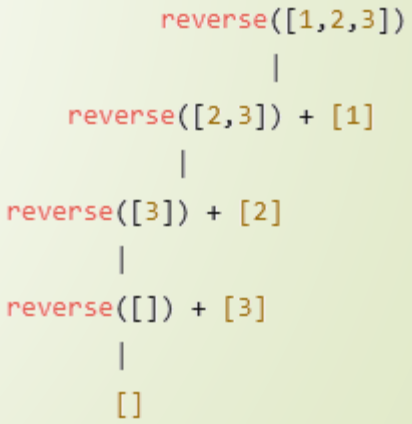
7. Αναδρομή

Ασκήσεις…

- **Άσκηση 3:** Γράψε αναδρομική συνάρτηση που αντιστρέφει μια λίστα.

```
def reverse(lst):
    if not lst:
        return []
    return reverse(lst[1:]) + [lst[0]]
```

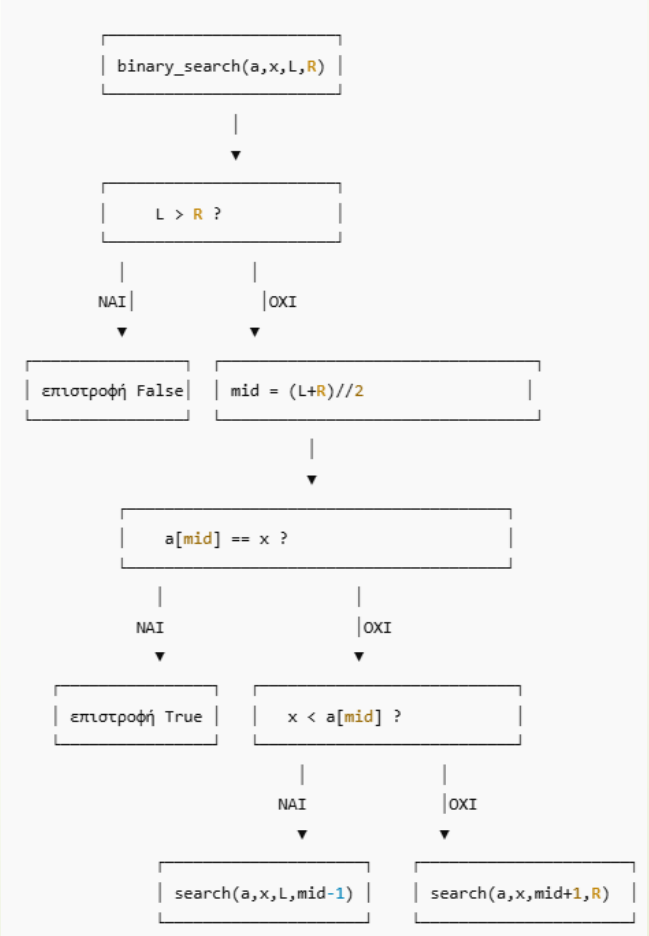




7. Αναδρομή

Ασκήσεις…

- **Άσκηση 4:** Υλοποίησε αναδρομική δυαδική αναζήτηση (binary search).

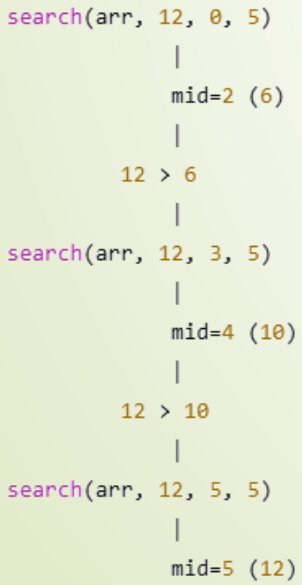


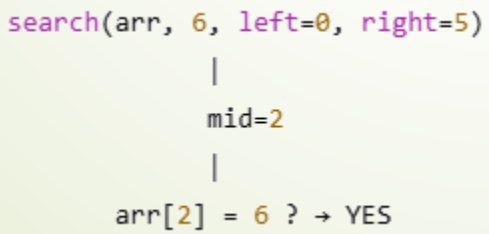
7. Αναδρομή

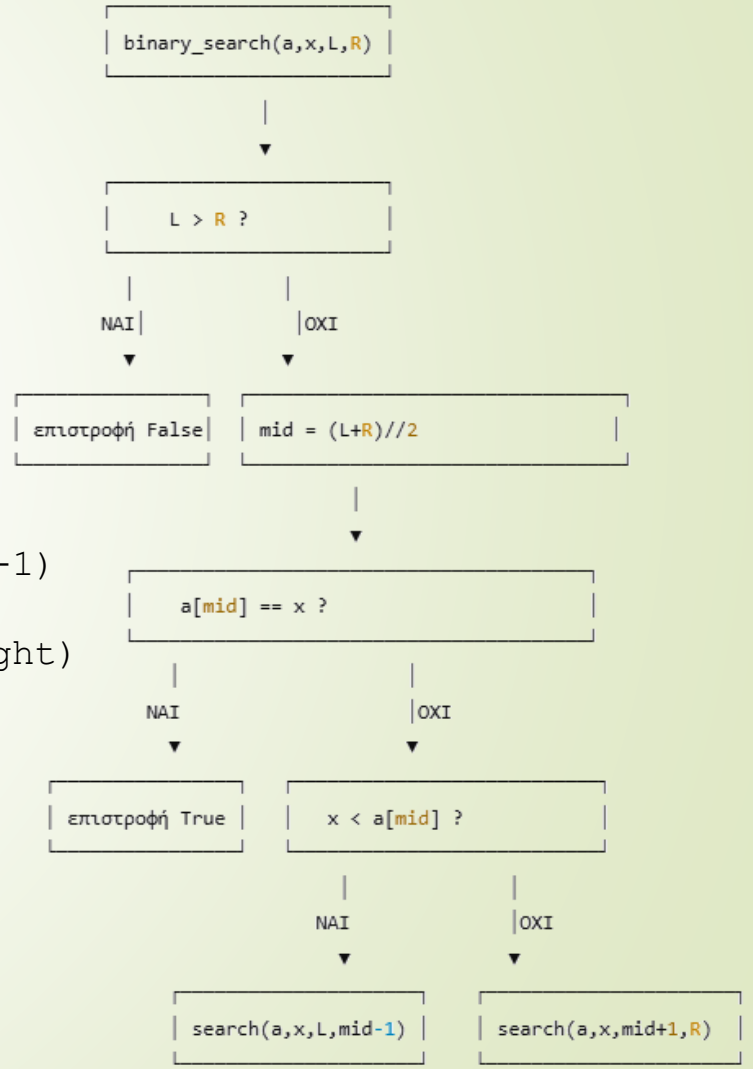
Ασκήσεις…

- **Άσκηση 4:** Υλοποίησε αναδρομική δυαδική αναζήτηση (binary search).

```
def binary_search(arr, target, left, right):
    if left > right:
        return False
    mid = (left + right) // 2
    if arr[mid] == target:
        return True
    elif target < arr[mid]:
        return binary_search(arr, target, left, mid-1)
    else:
        return binary_search(arr, target, mid+1, right)
```







7. Αναδρομή

Ασκήσεις…

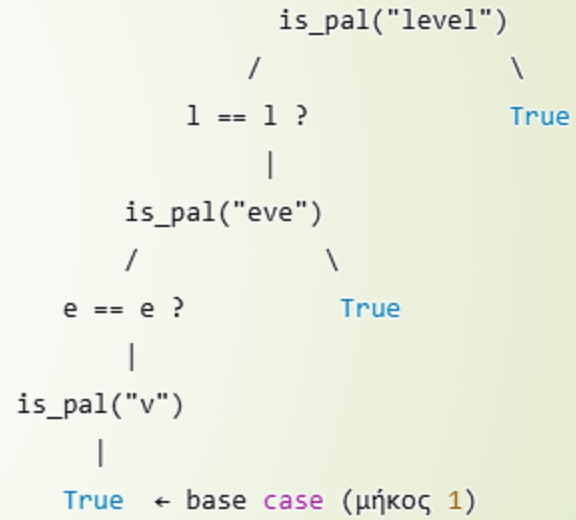
- **Άσκηση 5:** Γράψε συνάρτηση που ελέγχει αναδρομικά αν μια λέξη είναι παλίνδρομο.

7. Αναδρομή

Ασκήσεις…

- **Άσκηση 5:** Γράψε συνάρτηση που ελέγχει αναδρομικά αν μια λέξη είναι παλίνδρομο.

```
def is_pal(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_pal(s[1:-1])
```

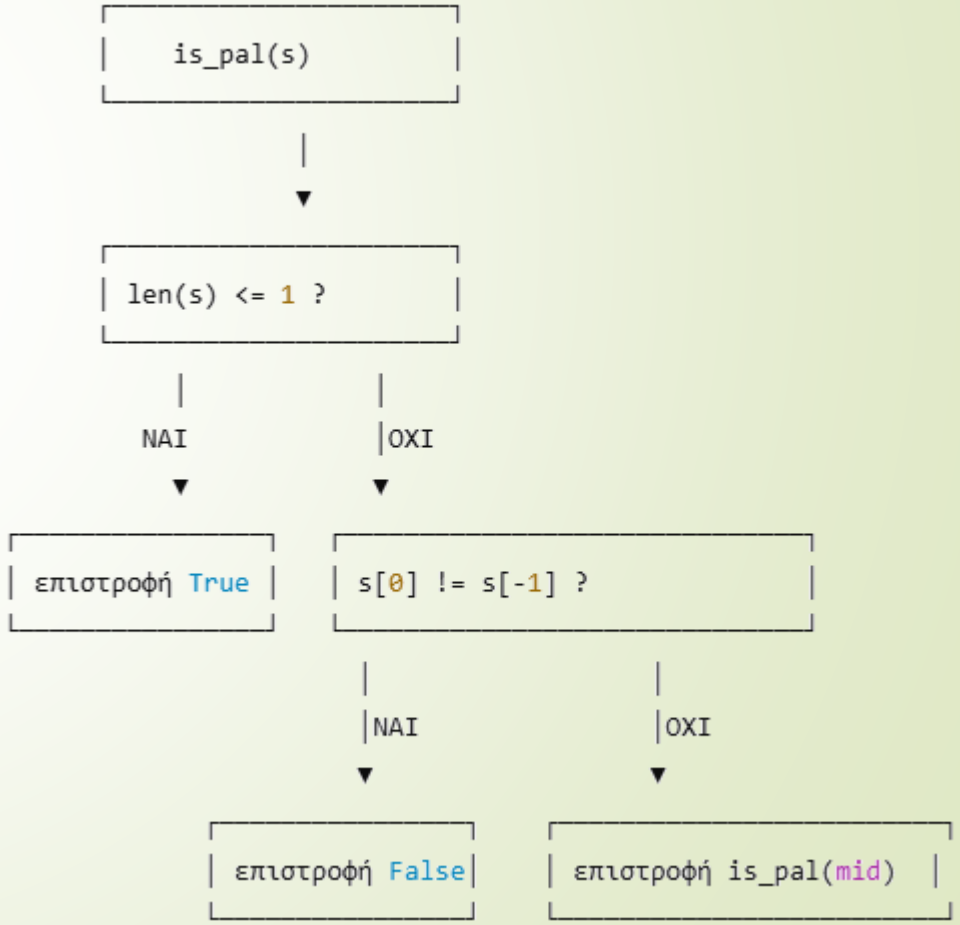


7. Αναδρομή

Ασκήσεις…

- **Άσκηση 5:** Γράψε συνάρτηση που ελέγχει αναδρομικά αν μια λέξη είναι παλίνδρομο.

```
def is_pal(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_pal(s[1:-1])
```



7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)
  - Δίνεται ένα πλέγμα n × m.
  - Ξεκινάς από τη θέση (0, 0) και θέλεις να φτάσεις στο (n–1, m–1).
  - Κανόνας κίνησης: → Μπορείς να πας ΜΟΝΟ Δεξιά ή Κάτω.

Ζητούμενο:

Γράψε μια αναδρομική συνάρτηση που επιστρέφειτο σύνολο των πιθανών μονοπατιών από την αρχή στο τέλος.

Παράδειγμα:

Για πλέγμα 2×2, τα μονοπάτια είναι:

Right → Down

Down → Right

Άρα: 2 μονοπάτια.

7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)
  - Δίνεται ένα πλέγμα n × m.
  - Ξεκινάς από τη θέση (0, 0) και θέλεις να φτάσεις στο (n–1, m–1).
  - Κανόνας κίνησης: → Μπορείς να πας ΜΟΝΟ Δεξιά ή Κάτω.

Ζητούμενο:

Γράψε μια αναδρομική συνάρτηση που επιστρέφειτο σύνολο των πιθανών μονοπατιών από την αρχή στο τέλος.

Παράδειγμα:

Για πλέγμα 2×2, τα μονοπάτια είναι:

Right → Down

Down → Right

Άρα: 2 μονοπάτια.

Αν είμαι σε κελί (r, c), δύο επιλογές:

1. Πάω **κάτω** → (r+1, c)\
2. Πάω **δεξιά** → (r, c+1)

Έτσι: paths(r, c) = paths(r+1, c) + paths(r, c+1)

Base case 1:

Αν φτάσω στο τελευταίο κελί → επιστροφή 1 (βρήκα μονοπάτι)

Base case 2:

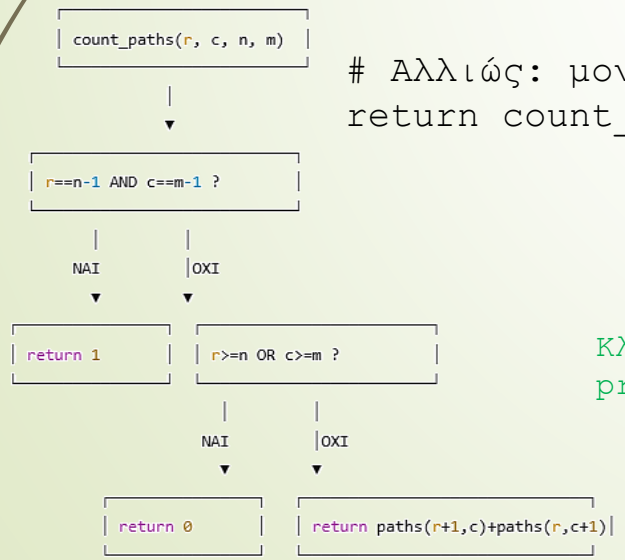
Αν βγω έξω από το πλέγμα → επιστροφή 0

7. Αναδρομή

Εφαρμογή …

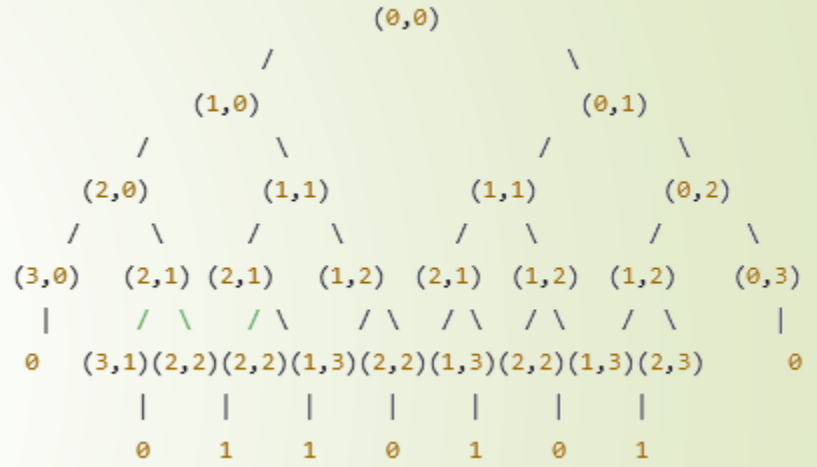
- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

```
def count_paths(r, c, n, m):
    # Αν φτάσαμε στον προορισμό => 1 μονοπάτι
    if r == n - 1 and c == m - 1:
        return 1

    # Αν βγήκαμε εκτός πλέγματος => 0 μονοπάτια
    if r >= n or c >= m:
        return 0

    # Αλλιώς: μονοπάτια δεξιά + μονοπάτια κάτω
    return count_paths(r + 1, c, n, m) + count_paths(r, c + 1, n, m)
```




```
Κλήση για πλέγμα 3×3:
print(count_paths(0, 0, 3, 3))    # Απάντηση: 6
```

- **Κάθε (2,2) επιστρέφει 1 (στόχος)**
- **Όσα ξεφεύγουν από το πλέγμα (3,) ή (,3) → 0**
- **Έχουμε ΠΟΛΛΕΣ επαναλήψεις γι' αυτό είναι δύσκολο χωρίς DP/memoization**

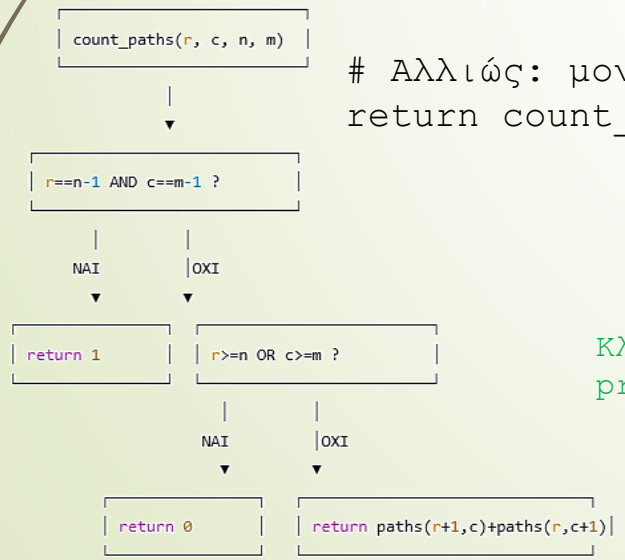


7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

```
def count_paths(r, c, n, m):
    # Αν φτάσαμε στον προορισμό => 1 μονοπάτι
    if r == n - 1 and c == m - 1:
        return 1

    # Αν βγήκαμε εκτός πλέγματος => 0 μονοπάτια
    if r >= n or c >= m:
        return 0

    # Αλλιώς: μονοπάτια δεξιά + μονοπάτια κάτω
    return count_paths(r + 1, c, n, m) + count_paths(r, c + 1, n, m)
```




```
Κλήση για πλέγμα 3×3:
print(count_paths(0, 0, 3, 3))    # Απάντηση: 6
```

Κλήση 1
count_paths(0,0)
Καλεί δύο:
TOP → | count_paths(1,0) |
| count_paths(0,0) |

Κλήση 2
count_paths(1,0)
→ καλεί:
TOP → | count_paths(2,0) |
| count_paths(1,0) |
| count_paths(0,0) |

Κλήση 3
count_paths(2,0)
→ καλεί:
TOP → | count_paths(3,0) |
| count_paths(2,0) |
| count_paths(1,0) |
| count_paths(0,0) |
Base case → επιστροφή 0
Pop:
TOP → | count_paths(2,0) |
| count_paths(1,0) |
| count_paths(0,0) |
Τώρα καλεί count_paths(2,1):
TOP → | count_paths(2,1) |
| count_paths(2,0) |
| count_paths(1,0) |
| count_paths(0,0) |
Αυτό συνεχίζεται μέχρι να συναντήσει (2,2) → base case 1.

7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

Top-Down Dynamic Programming (Memoization)

```
def count_paths_memo(r, c, n, m, memo):
    # Αν υπάρχει στο memo → επέστρεψέ το
    if (r, c) in memo:
        return memo[(r, c)]
    # Base case: φτάσαμε στον προορισμό
    if r == n - 1 and c == m - 1:
        return 1
    # Base case: έξω από πλέγμα
    if r >= n or c >= m:
        return 0
    # Αλλιώς: υπολογισμός και αποθήκευση
    memo[(r, c)] = (
        count_paths_memo(r + 1, c, n, m, memo) +
        count_paths_memo(r, c + 1, n, m, memo)
    )
    return memo[(r, c)]
# Κλήση
memo = {}
print(count_paths_memo(0, 0, 3, 3, memo))    # Απάντηση: 6
```

7. Αναδρομή

Εφαρμογή …

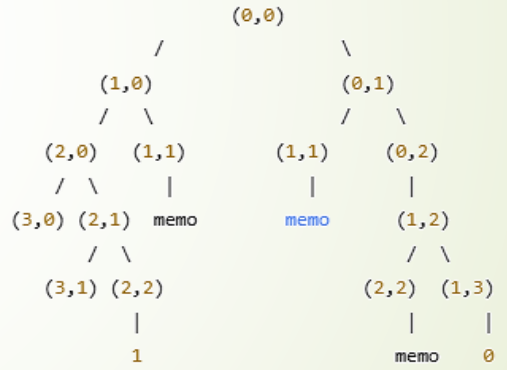
- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

Top-Down Dynamic Programming (Memoization)

```
def count_paths_memo(r, c, n, m, memo):
    # Αν υπάρχει στο memo → επέστρεψέ το
    if (r, c) in memo:
        return memo[(r, c)]
    # Base case: φτάσαμε στον προορισμό
    if r == n - 1 and c == m - 1:
        return 1
    # Base case: έξω από πλέγμα
    if r >= n or c >= m:
        return 0
    # Αλλιώς: υπολογισμός και αποθήκευση
    memo[(r, c)] = (
        count_paths_memo(r + 1, c, n, m, memo) +
        count_paths_memo(r, c + 1, n, m, memo)
    )
    return memo[(r, c)]
# Κλήση
memo = {}
print(count_paths_memo(0, 0, 3, 3, memo))    # Απάντηση: 6
```

Χρησιμοποιούμε την ίδια αναδρομική λογική, αλλά **αποθηκεύουμε** κάθε υπολογισμένο αποτέλεσμα στο memo.

✓ Αποφεύγουμε επαναληπτικούς υπολογισμούς
✓ Το recursion tree γίνεται *γραμμικό*
✓ Τεράστια επιτάχυνση



- Το (1,1) υπολογίζεται **μία** φορά
- Το (2,2) υπολογίζεται **μία** φορά
- Πολλοί κόμβοι οδηγούν σε **memo hits**

```
count_paths_memo(0,0)
  → (1,0)
      → (2,0)
          → (3,0) → 0
          → (2,1)
              → (3,1) → 0
              → (2,2) → 1
      → (1,1)
          → (2,1) (memo hit)
          → (1,2)
              → (2,2) (memo hit)
              → (1,3) → 0
  → (0,1)
      (1,1) (memo hit)
      (0,2)
          → (1,2) (memo hit)
          → (0,3) → 0
```

7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

Bottom-Up Dynamic Programming (Tabulation)

```
def count_paths_bottomup(n, m):
    # Δημιουργούμε πίνακα n×m με μηδενικά
    dp = [[0]*m for _ in range(n)]
    # Το τελευταίο κελί έχει πάντα 1 μονοπάτι
    dp[n-1][m-1] = 1
    # Γέμισμα πίνακα
    for r in range(n-1, -1, -1):
        for c in range(m-1, -1, -1):
            # Παραλείπουμε το τελευταίο κελί
            if r == n-1 and c == m-1:
                continue
            right = dp[r][c+1] if c+1 < m else 0
            down  = dp[r+1][c] if r+1 < n else 0
            dp[r][c] = right + down
    return dp[0][0]
```

7. Αναδρομή

Εφαρμογή …

- Πλήθος μονοπατιών σε πλέγμα (Grid Paths)

Bottom-Up Dynamic Programming (Tabulation)

```
def count_paths_bottomup(n, m):
    # Δημιουργούμε πίνακα n×m με μηδενικά
    dp = [[0]*m for _ in range(n)]
    # Το τελευταίο κελί έχει πάντα 1 μονοπάτι
    dp[n-1][m-1] = 1
    # Γέμισμα πίνακα
    for r in range(n-1, -1, -1):
        for c in range(m-1, -1, -1):
            # Παραλείπουμε το τελευταίο κελί
            if r == n-1 and c == m-1:
                continue
            right = dp[r][c+1] if c+1 < m else 0
            down  = dp[r+1][c] if r+1 < n else 0
            dp[r][c] = right + down
    return dp[0][0]
```

```
goal (2,2) = 1

(2,1) = 1
(2,0) = 1

(1,2) = 1
(1,1) = (1 + 1) = 2
(1,0) = (2 + 1) = 3

(0,2) = 1
(0,1) = (1 + 2) = 3
(0,0) = (3 + 3) = 6
```

Γεμίζουμε έναν πίνακα dp[n][m] από το τέλος:

- Το τελευταίο κελί = 1
- Τα υπόλοιπα = άθροισμα δεξιάς + κάτω γειτονικής τιμής
- Γεμίζουμε τον πίνακα από κάτω δεξιά προς πάνω αριστερά

Γεμίζουμε από κάτω προς τα πάνω:

Βήμα 1 — Αρχικό

```
0 0 0
0 0 0
0 0 1
```

Βήμα 2 — Συμπλήρωση τελευταίας σειράς & τελευταίας στήλης

```
0 0 1
0 0 1
0 1 1
```

Βήμα 3 — Πλήρης πίνακας

```
6 3 1
3 2 1
1 1 1
```

Παρατήρηση

- Το dp[0][0] = 6
- Αντιστοιχεί στο πλήθος μονοπατιών
- Είναι ίδιο με το αποτέλεσμα της αναδρομής αλλά πολύ πιο γρήγορο