



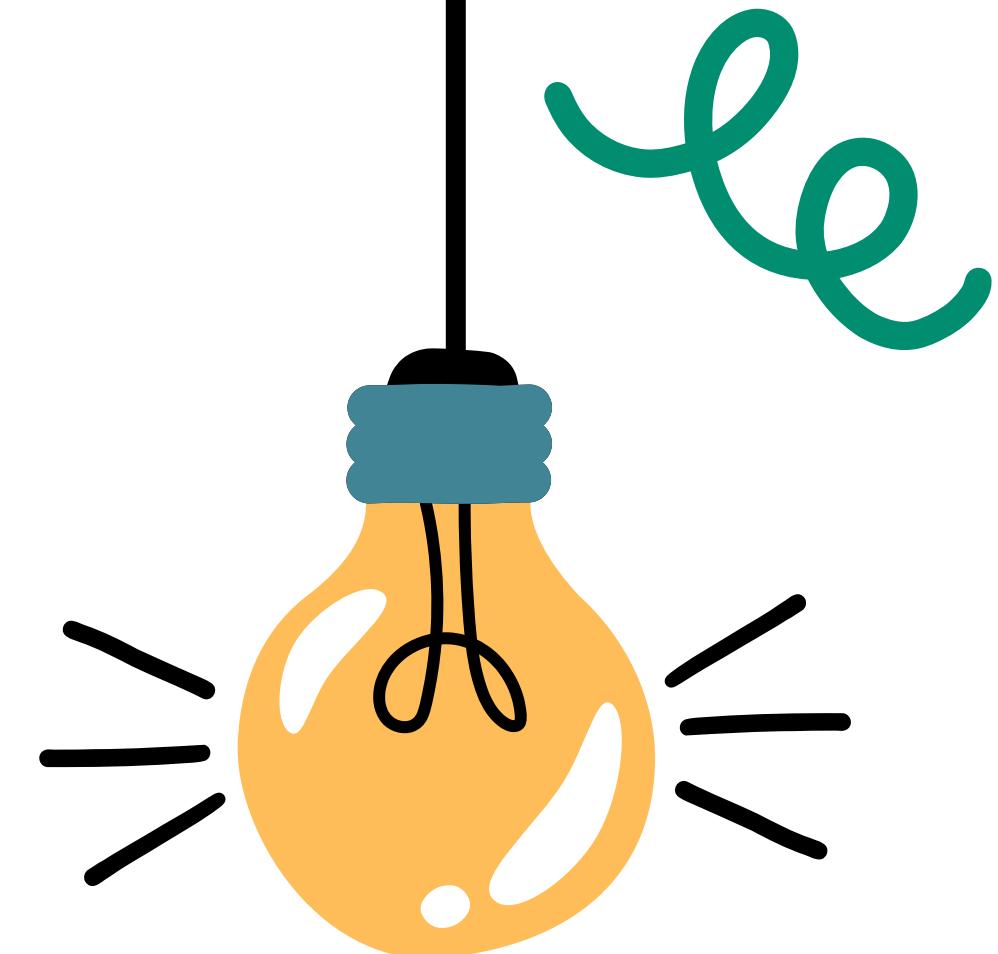
# IEEEEXTREME

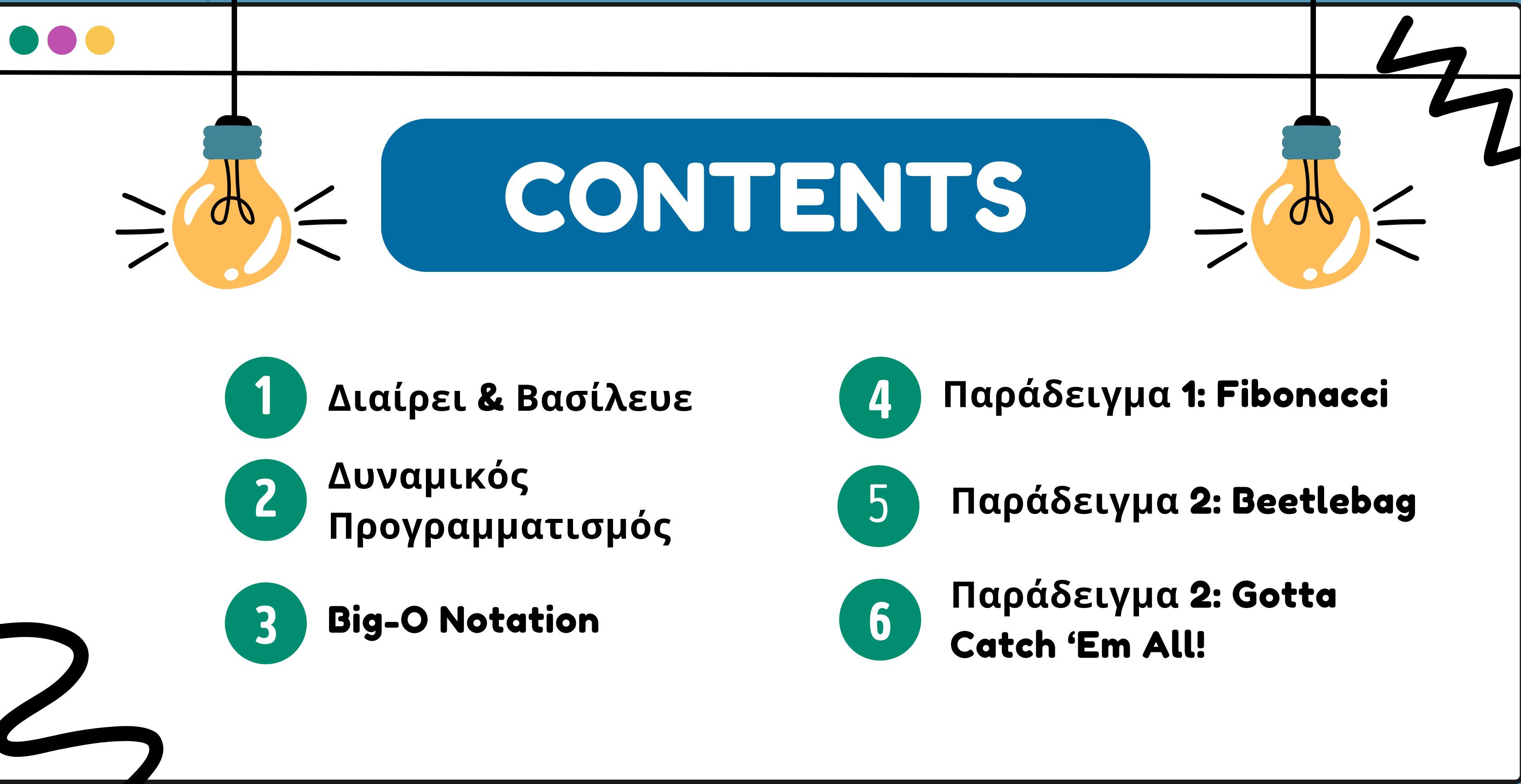
## WORKSHOP

- Παρουσίαση: Σπύρος Σταμούλης
- Διαφάνειες: Χάρης Ζαχαριάδης



**IEEE** Student Branch  
University of Thessaly







# DIVIDE & CONQUER

**Διαίρει:** Αν το πρόβλημα είναι μεγάλο για να τα διαχειριστείς απευθείας , διαίρεσέ τα σε μικρότερα υποπρόβλημα.

**Βασίλευε:** “Βασίλευε” αναδρομικά για να λύσεις το κάθε υποπρόβλημα

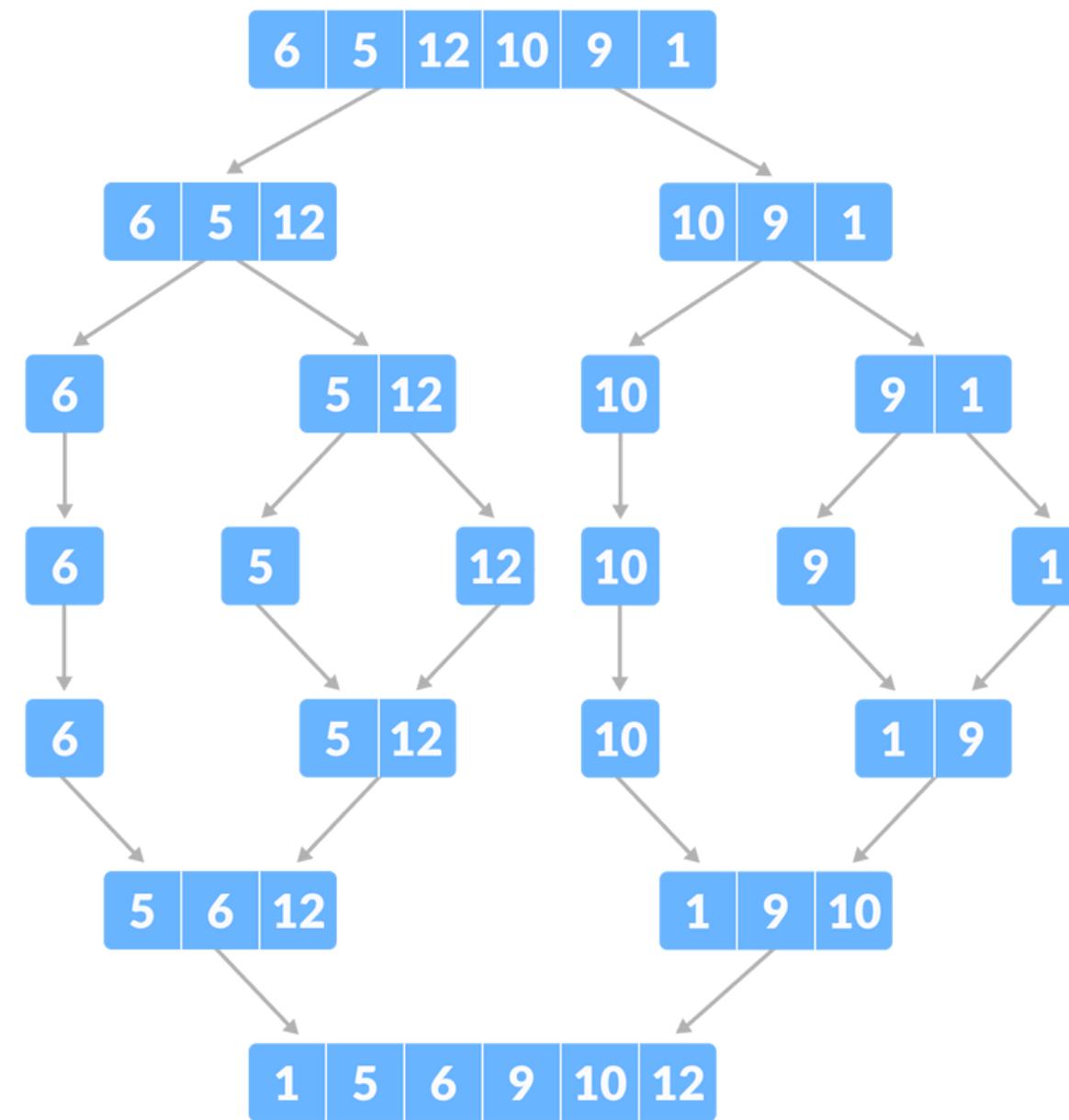
**Ένωσε:** Πάρε τις λύσεις από κάθε υποπρόβλημα και “ένωσέ” τες σε μια εννιαία λύση που είναι η λύση του αρχικού προβλήματος.





# DIVIDE & CONQUER

- Για παράδειγμα, ο αλγόριθμος **Merge Sort** είναι ένα χαρακτηριστικό παράδειγμα αυτής της τακτικής.



```
Merge-Sort(A, p, r)
  if p < r then
    q←(p+r)/2
    Merge-Sort(A, p, q)
    Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
```

- Όλα τα υποπροβλήματα όπως φαίνεται και στο παράδειγμα είναι **ανεξάρτητα** και **διαφορετικά**, ώστόσο ο συνδιασμός των λύσεων τους οδηγεί στη λύση του αρχικού προβλήματος.



# DYNAMIC PROGRAMMING (DP)

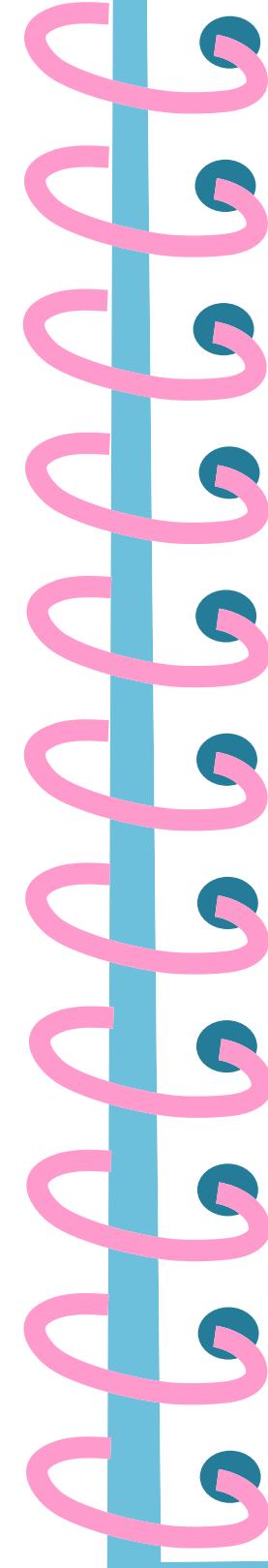
- Η μέθοδος “Διαίρει και Βασίλευε” δεν είναι πολύ αποτελεσματική καθώς χρησιμοποιεί τις ίδιες αναδρομικές κλήσεις για ίδια υποπροβλήματα.
- Ο Δυναμικός Προγραμματισμός αξιοποιεί την εμφάνιση των ίδιων υποπροβλημάτων **αποθηκέυοντας την πρώτη φορά τη λύση ενός υποπροβλήματος σε έναν πίνακα** ώστε να μην χρειαστεί να υπολογιστεί ξανά η λύση του.
- Άρα, ο Δυναμικός Προγραμματισμός εφαρμόζεται σε υποπροβλήματα τα οποία μοιράζονται υποπροβλήματα με άλλα υποπροβλήματα (υπάρχει δηλαδή **εξάρτηση** μεταξύ τους).

## Αρχή Βελτιστοποίησης

«Κάθε τμήμα μιας βέλτιστης διαδρομής είναι επίσης βέλτιστο.»



# DYNAMIC PROGRAMMING (DP)

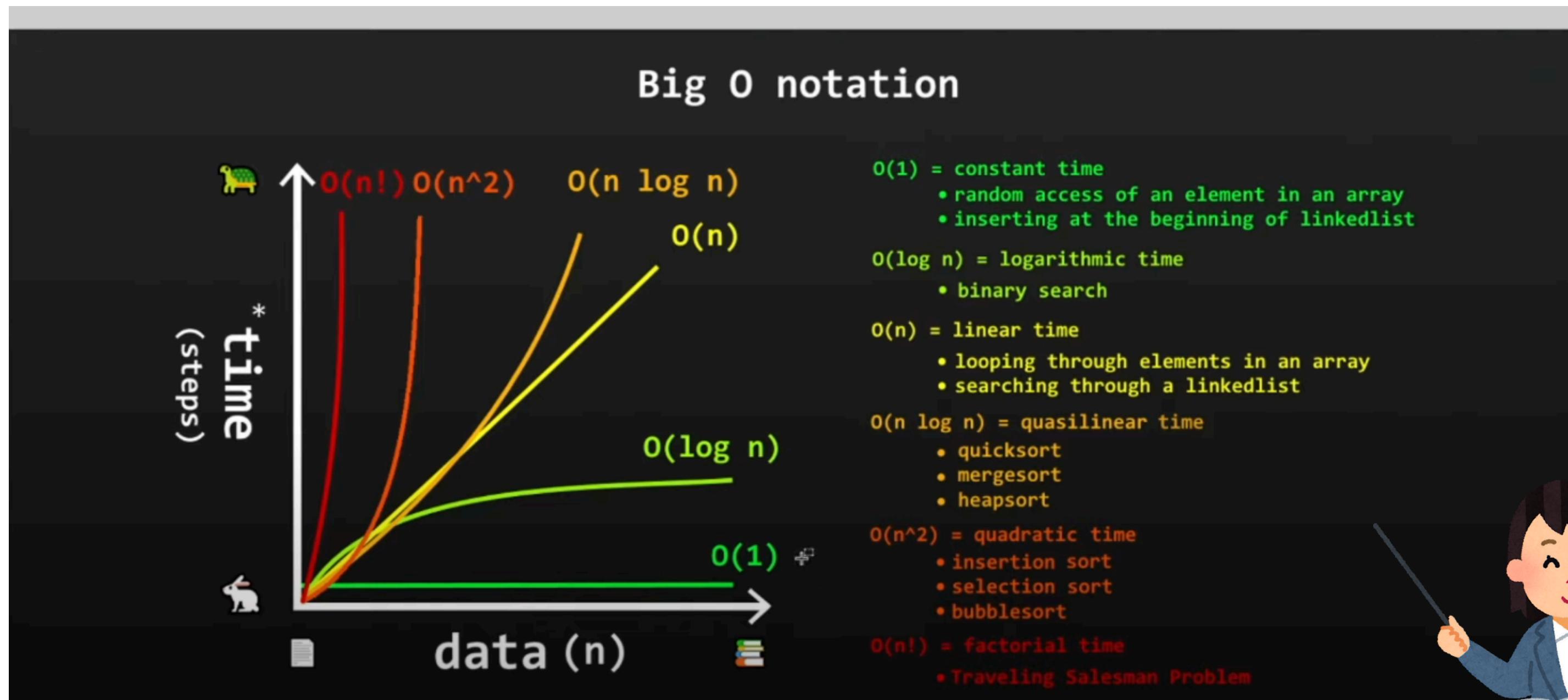


## Σχεδιασμός του Αλγορίθμου:

1. Προσδιόρισε τις μεταβλητές του προβλήματος.
2. Διατύπωσε ξεκάθαρα τη σχέση επαναληψιμότητας (recurrence relation).
3. Προσδιόρισε τις αρχικές περιπτώσεις (base cases).
4. Αποφάσισε αν θέλεις να το υλοποιήσεις επαναληπτικά ή αναδρομικά.
5. Προσδιόρισε την πολυπλοκότητα χρόνου.



# BIG-O NOTATION



Ολοκληρωμένη παρουσίαση



# Fibonaci



## Εκφώνηση (από IEEEXtreme 11.0)

### Overview:

Dr. Fibonacci models population growth using a sequence:

- Bacteria population:
  - 11 in minute 1, 22 in minute 2, 33 in minute 3, 55 in minute 4, etc.
- Formula:

$$\text{Population}[n] = \text{Population}[n-1] + \text{Population}[n-2]$$

### Problem Setup:

When a disaster occurs in generation m, the survivors are calculated using:

- Population % 10

### Challenge:

- Dr. Xtreme wants to adapt this model for human population growth.
- A disaster scenario is introduced where most humans are wiped out.

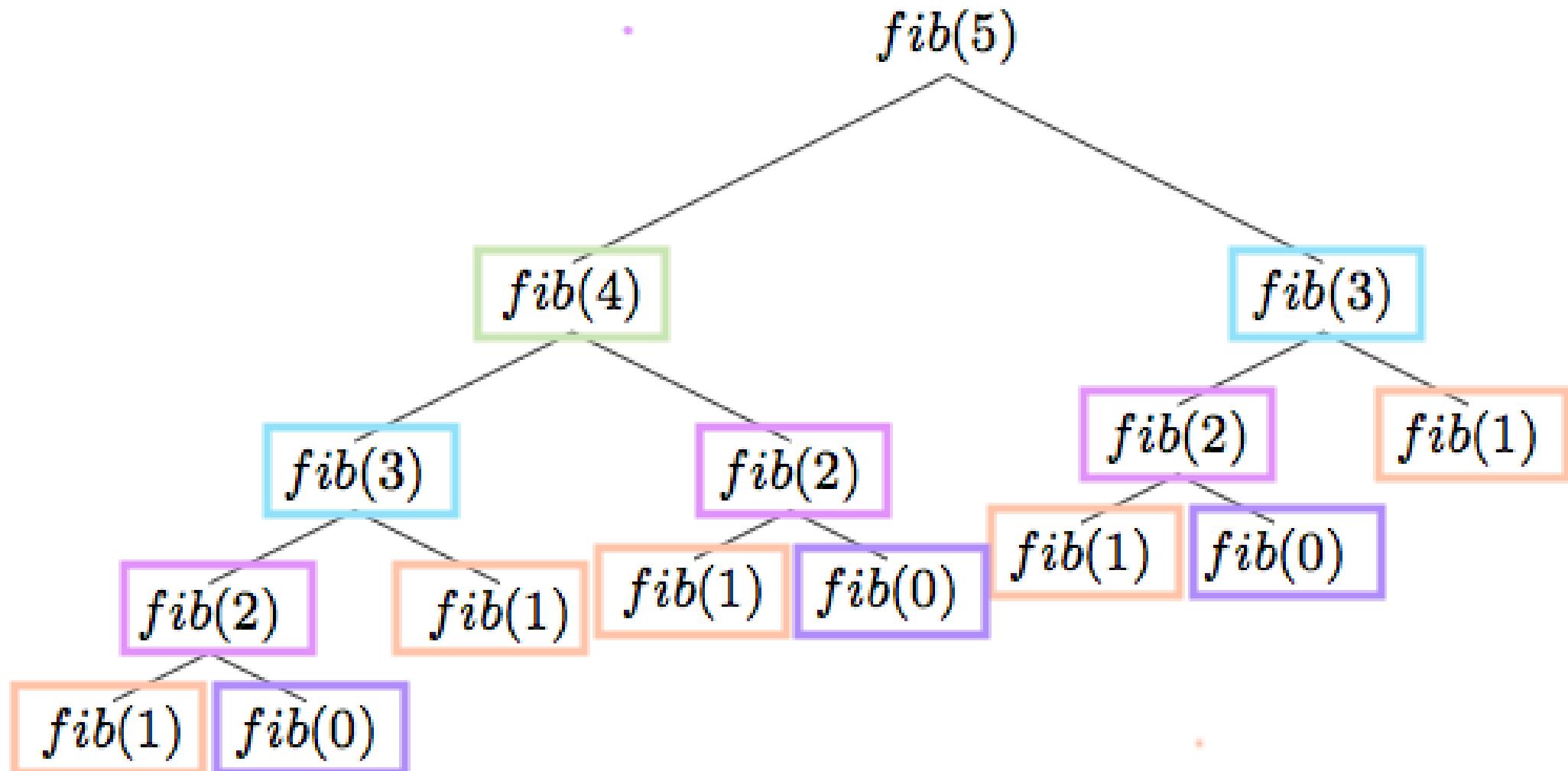
### Goal:

- Develop the population model that includes disaster effects to calculate the number of survivors after a disaster in any generation.

1,1,2,3,5,8,13,21...



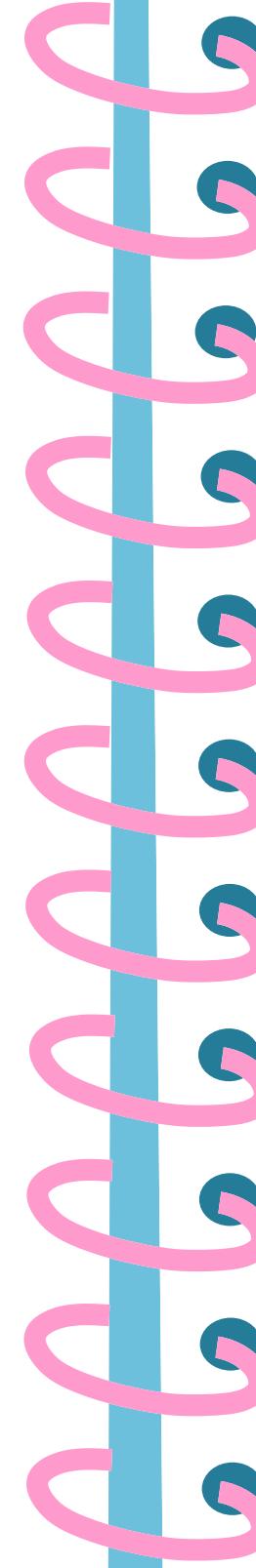
# Fibonacci

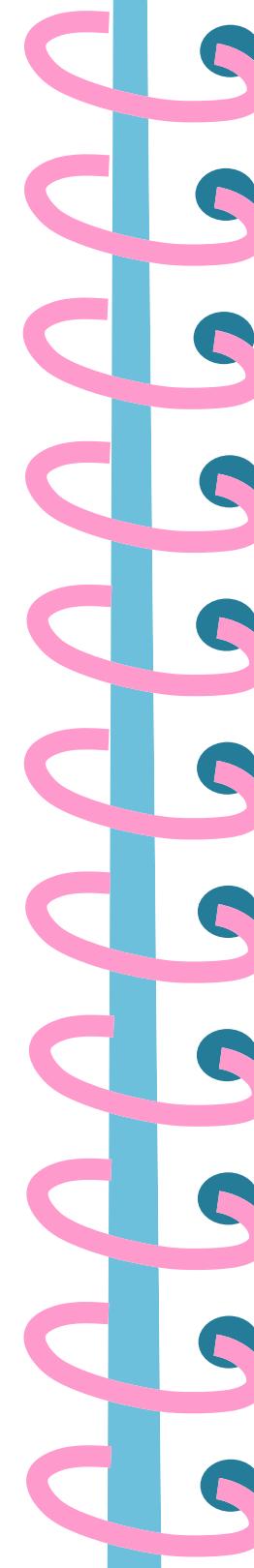


- Παρατηρούμε επαναλαμβανόμενα υποπροβλήματα.
- Επομένως έχουμε πρόβλημα **Δυναμικού Προγραμματισμού!**

## Σχεδιασμός του Αλγορίθμου:

1. Προσδιόρισε τις μεταβλητές του προβλήματος.
  - **t**: Αριθμός πειραμάτων
  - **m**:  $n$ -th Fibonacci number του κάθε πειράματος





## Σχεδιασμός του Αλγορίθμου:

2. Διατύπωσε ξεκάθαρα τη σχέση επαναληψιμότητας
  - $\text{Fib}[n] = \text{Fib}[n-1] + \text{Fib}[n-2]$



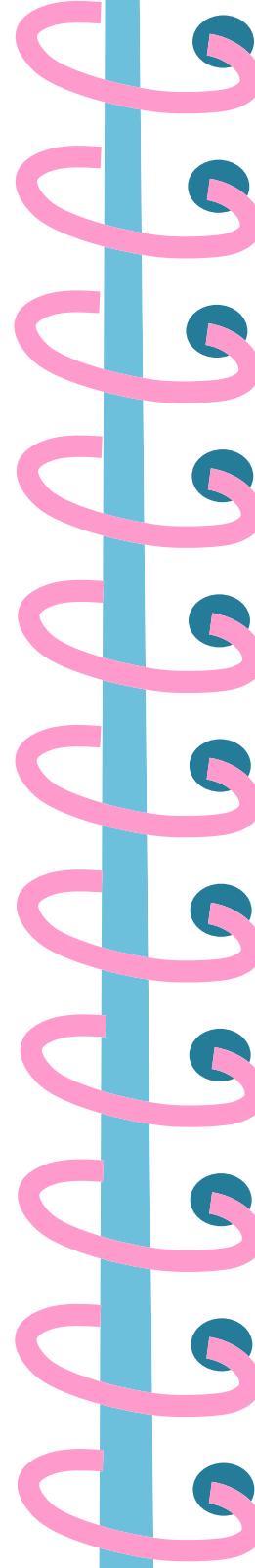
## Σχεδιασμός του Αλγορίθμου:

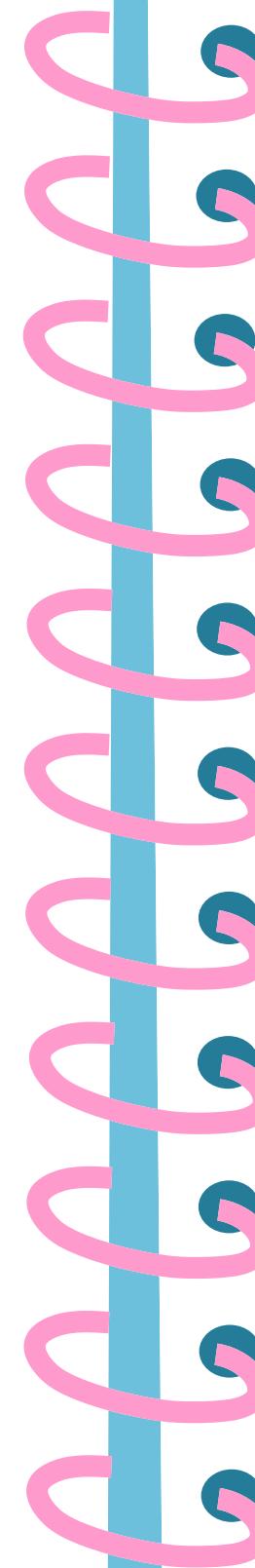
3. Προσδιόρισε τις αρχικές περιπτώσεις (base cases).
  - $n=1 \rightarrow \text{result}=1$
  - $n=2 \rightarrow \text{result}=2$
  - $n=3 \rightarrow \text{result}=3$

## Σχεδιασμός του Αλγορίθμου:

4. Αποφάσισε αν θέλεις να το υλοποιήσεις:

- **Επαναληπτικά**
- **Αναδρομικά**





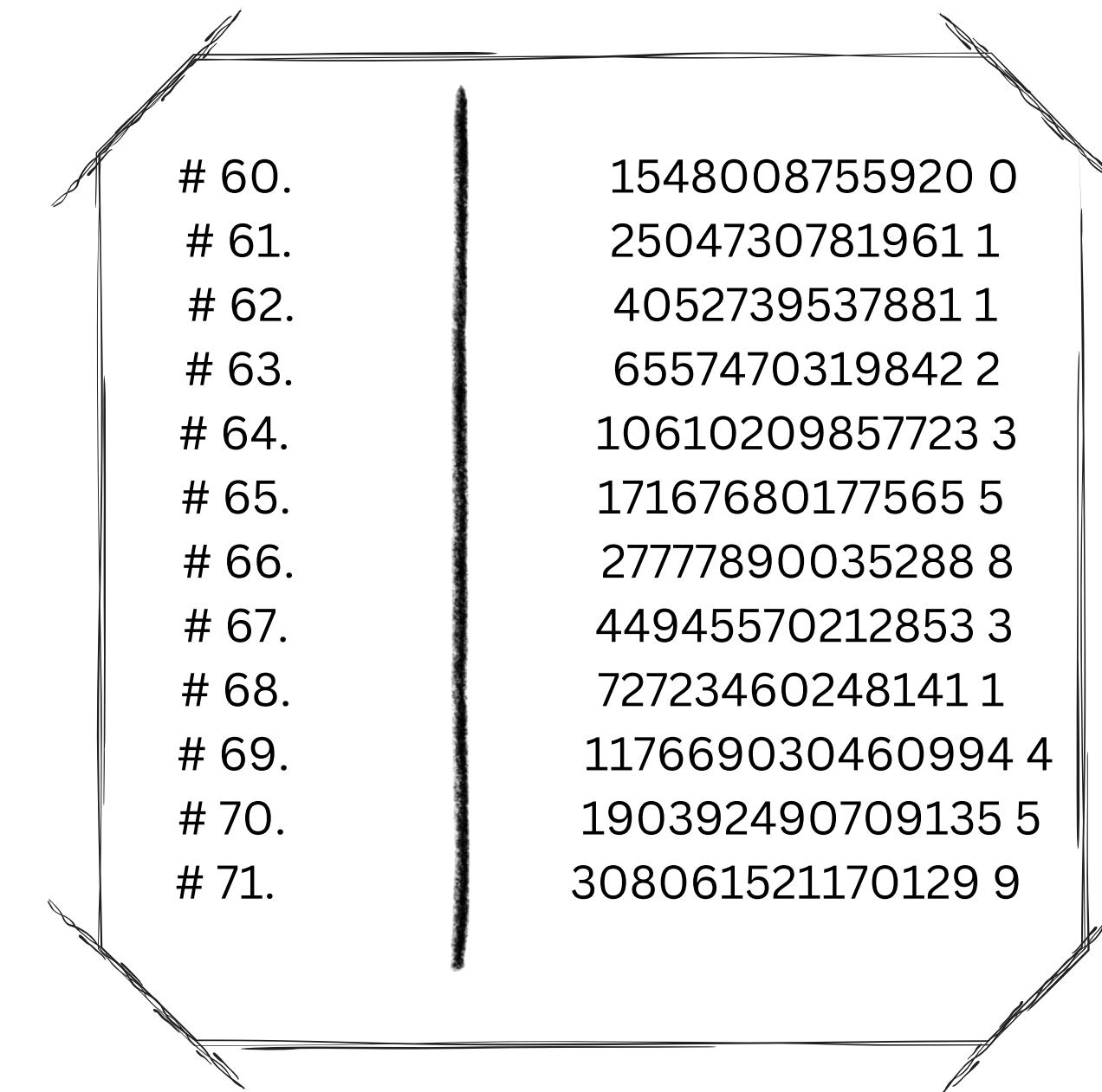
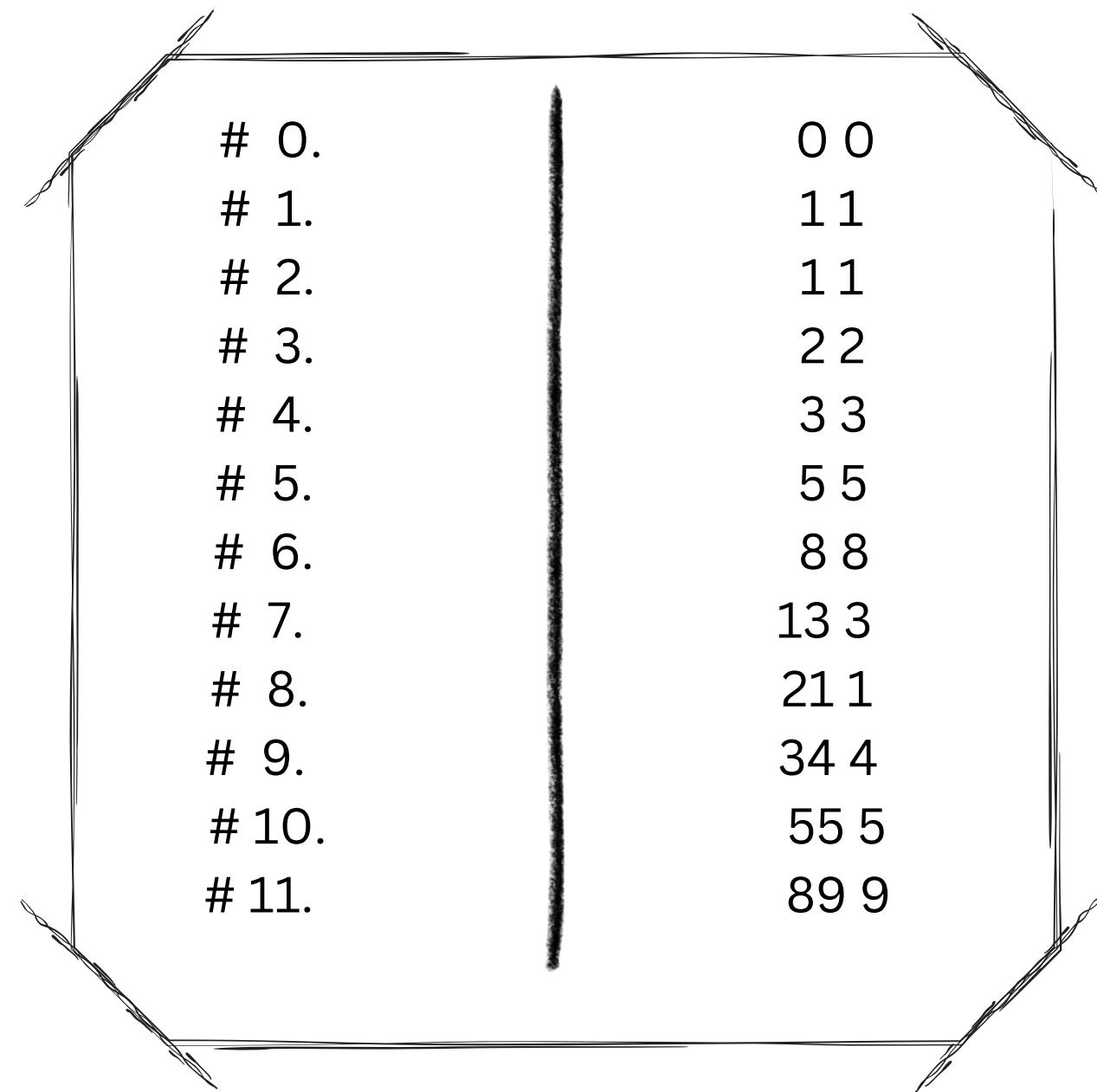
## Σχεδιασμός του Αλγορίθμου:

5. Προσδιόρισε την πολυπλοκότητα χρόνου:

- **Επαναληπτικά (tabulation):**  $O(n)$
- **Αναδρομικά (memoization):**  $O(n)$
- **Αναδρομικά (όχι memoization):**  $O(2^n)$

# Fibonacci

- Παρατηρούμε ένα μοτίβο:



[Source](#)

# BeetleBag



## Εκφώνηση (από IEEEXtreme 11.0)

### Overview:

- Beetleman joined the Strangers, a superhero team protecting the cyber world. To enhance his fighting power, Copperman granted Beetleman access to his lab's gadgets.

### Problem Setup:

- Beetleman has limited space in his hero bag. Each gadget has a specific weight and fighting power.

### Challenge:

- Beetleman must select gadgets that maximize his fighting power without exceeding his bag's capacity.

### Goal:

- Help Beetleman choose the optimal set of gadgets to maximize his fighting power within the bag's weight limit, based on multiple test cases.

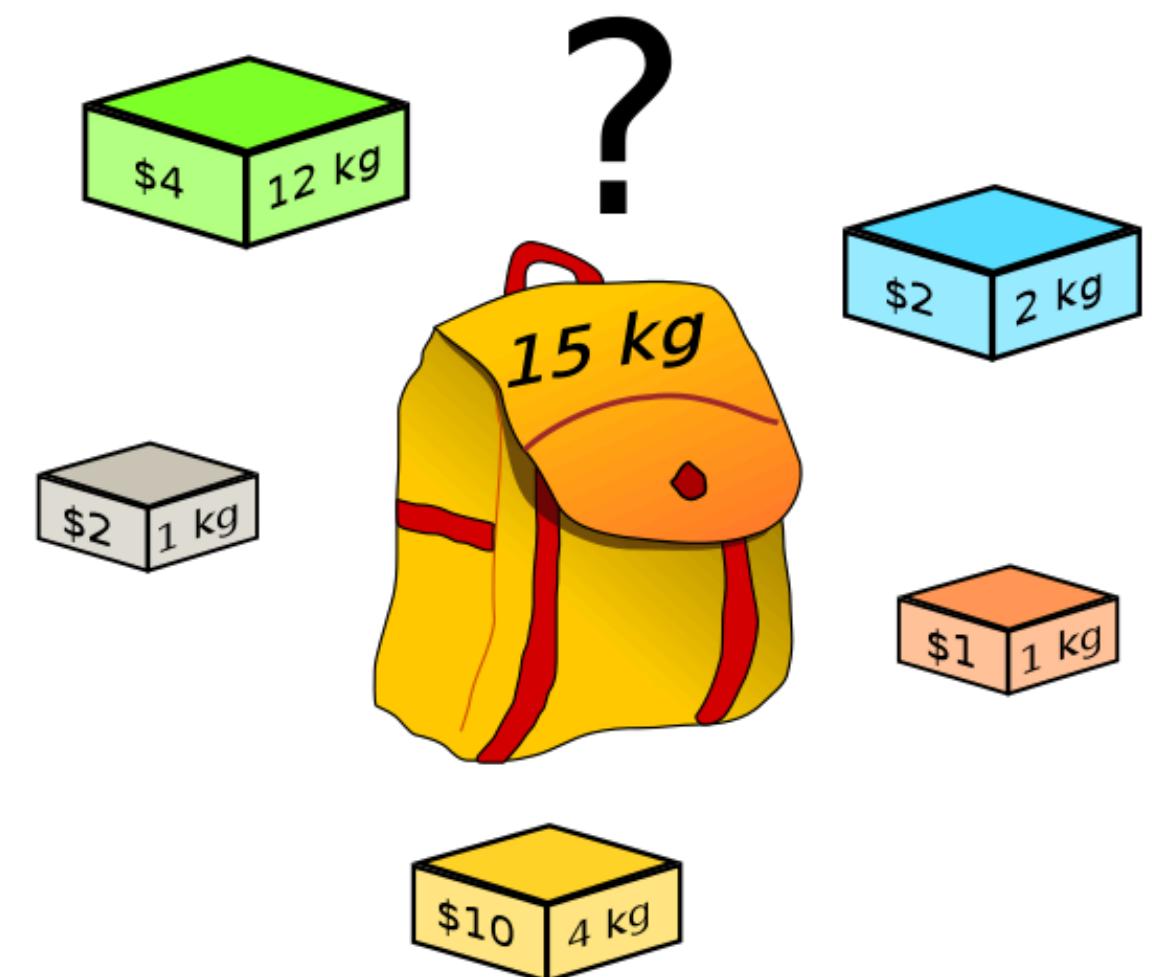


# 0-1 Knapsack

Ας υποθέσουμε ότι τα  $w_1, w_2, \dots, w_n, W$  είναι αυστηρά θετικοί ακέραιοι αριθμοί. Ορίζουμε το  $m[i, w]$  ως τη **μέγιστη τιμή που μπορεί να επιτευχθεί με βάρος μικρότερο ή ίσο με  $w$**  χρησιμοποιώντας αντικείμενα μέχρι το  $i$  (τα πρώτα  $i$  αντικείμενα).

Μπορούμε να ορίσουμε το  $m[i, w]$  αναδρομικά ως εξής:

- $m[0, w] = 0$
- $m[i, w] = m[i-1, w]$ , αν  $w_i > w$  (**το νέο αντικείμενο είναι μεγαλύτερο από το τρέχον όριο βάρους**)
- $m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i)$ , αν  $w_i \leq w$ .





# 0-1 Knapsack

$$m[0, w] = 0$$

		Capacity							
		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1								
$v_2=2, w_2=1$	2								
$v_3=4, w_3=3$	3								
$v_4=5, w_4=4$	4								
$v_5=3, w_5=2$	5								

# 0-1 Knapsack

$$m[i, w] = m[i-1, w], \text{ if } w_i > w$$

	Capacity							
	0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0						
$v_2=2, w_2=1$	2							
$v_3=4, w_3=3$	3							
$v_4=5, w_4=4$	4							
$v_5=3, w_5=2$	5							

# 0-1 Knapsack

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i), \text{ if } w_i \leq w$$

	Capacity							
	0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	0	0	0	0	0	0	0	0
$v_2=2, w_2=1$	0	0	0	0	0	0	0	0
$v_3=4, w_3=3$	0	0	0	0	0	0	0	0
$v_4=5, w_4=4$	0	0	0	0	0	0	0	0
$v_5=3, w_5=2$	0	0	0	0	0	0	0	0

Diagram illustrating the 0-1 Knapsack problem using a dynamic programming table. The table has rows for items (0 to 5) and columns for capacity (0 to 7). The value  $v_i$  and weight  $w_i$  for each item are listed on the left. The table entries represent the maximum value  $m[i, w]$ . Blue arrows show the path from the bottom-left to the cell at row 1, column 3.

# 0-1 Knapsack

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i), \text{ if } w_i \leq w$$

		Capacity							
		0	1	2	3	4	5	6	7
Empty		0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2						
$v_3=4, w_3=3$	3								
$v_4=5, w_4=4$	4								
$v_5=3, w_5=2$	5								

A diagram showing a 0-1 Knapsack DP table. The columns represent capacity from 0 to 7. The rows represent items: item 1 (v1=2, w1=3), item 2 (v2=2, w2=1), item 3 (v3=4, w3=3), item 4 (v4=5, w4=4), and item 5 (v5=3, w5=2). The value at each cell (i, w) is the maximum value of either not taking item i or taking item i. An arrow points to the cell (2, 2) which contains 2, indicating that item 2 should be included in the knapsack to achieve a value of 2 at capacity 2.

# 0-1 Knapsack

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i), \text{ if } w_i \leq w$$

		Capacity							
		0	1	2	3	4	5	6	7
Empty		0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	2				
$v_3=4, w_3=3$	3								
$v_4=5, w_4=4$	4								
$v_5=3, w_5=2$	5								

A diagram illustrating the 0-1 Knapsack problem. The table shows the maximum value that can be obtained for a given capacity  $w$  by including items  $i$  or not. The columns represent capacities from 0 to 7. The rows represent items  $i$  with values  $v_i$  and weights  $w_i$ . Blue arrows point to the cell at capacity 3, row  $v_2=2, w_2=1$ , which contains 2, indicating that item 2 can be included in the knapsack to achieve a value of 2 at capacity 3.

# 0-1 Knapsack

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i), \text{ av } w_i \leq w$$

		Capacity							
		0	1	2	3	4	5	6	7
Empty		0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	2	4			
$v_3=4, w_3=3$	3								
$v_4=5, w_4=4$	4								
$v_5=3, w_5=2$	5								

A 0-1 Knapsack DP table for a knapsack capacity of 7. The columns represent capacity from 0 to 7. The rows represent items 0 to 5. The value at  $m[i, w]$  is the maximum value of either not taking item  $i$  or taking item  $i$  and reducing the capacity by its weight  $w_i$ . The value is also the sum of the item's value  $v_i$  and the previous state's value  $m[i-1, w-w_i]$ .

The table shows the following values:

- Row 0 (Empty): [0, 0, 0, 0, 0, 0, 0, 0, 0]
- Row 1 ( $v_1=2, w_1=3$ ): [0, 0, 0, 0, 2, 2, 2, 2, 2]
- Row 2 ( $v_2=2, w_2=1$ ): [2, 0, 2, 2, 2, 4]
- Row 3 ( $v_3=4, w_3=3$ ): [3]
- Row 4 ( $v_4=5, w_4=4$ ): [4]
- Row 5 ( $v_5=3, w_5=2$ ): [5]

An arrow points from the value 2 in the cell  $m[1, 4]$  to the cell  $m[0, 4]$ , illustrating the recurrence relation  $m[1, 4] = \max(m[0, 4], m[0, 4-3] + v_1)$ .



# 0-1 Knapsack

$$m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i), \text{ av } w_i \leq w$$

	0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0
$v_1=2, w_1=3$	1	0	0	0	2	2	2	2
$v_2=2, w_2=1$	2	0	2	2	2	4	4	4
$v_3=4, w_3=3$	3	0	2	2	4	6	6	8
$v_4=5, w_4=4$	4	0	2	2	4	6	7	9
$v_5=3, w_5=2$	5	0	2	3	5	6	7	9



# 0-1 Knapsack

## Code:

```
for i from 1 to n do:  
    for j from 1 to W do:  
        if w[i] > j then:  
            m[i, j] := m[i-1, j]  
        else:  
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

# Gotta catch 'em all!



## Εκφώνηση (από IEEExtreme 11.0)

### Overview:

- Blue, an aspiring Pokémon master, embarks on his journey to Viridian City for his first gym battle.
- To reach the city, he must pass through the Viridian Forest, where his rival, Red, has set traps using enemy Pokémon.

### Game Setup:

- The forest is represented as a grid ( $r \times c$ ), where Blue starts at the top left and must exit at the bottom right.
- Blue can only move right or down through the grid.
- Each cell contains either:
  - A health potion (positive integer) that increases Pikachu's health.
  - An enemy Pokémon (negative integer) that decreases Pikachu's health.

### Challenge:

- Blue's Pikachu must fight these enemy Pokémon, losing health in each battle.
- However, health potions scattered throughout the forest can restore Pikachu's health.

### Goal:

- Plan the journey for him so that Pikachu trains until he increases his health capacity to the minimum health needed to defeat all the Pokémon in their journey to Viridian city.





## 1. Εύρεση Αναδρομική Συσχέτισης:

- Παρατηρώ ότι για κάθε κελί  $(i, j)$  το hp που χρειάζεται το πίκατσου είναι το ελάχιστο από το hp για να μετακινηθεί δεξιά  $(i, j+1)$  και από το hp για να μετακινηθεί κάτω  $(i+1, j)$ .
- Αν θέσουμε κάθε στοιχείο  $dp[i][j]$  το ελάχιστο hp που χρειάζεται για να πάει το πίκατσου στην κάτω δεξιά γωνία από το  $(i, j)$  κελί βρίσκουμε αυτή τη σχέση:

$$dp[i][j] = \max(1, \min(dp[i][j+1], dp[i+1][j]) - grid[i][j])$$

(Γιατί πρέπει να έχει τουλάχιστον 1 hp)

(Από την 1η παρατήρηση)

(Αν είναι αρνητικό πρέπει να αυξηθεί το hp αν είναι θετικό μπορούμε να μειώσουμε το hp)

## 2. Base Case:

Στην πιο απλή περίπτωση η αρχή είναι ίδια με το τέλος ára θέλουμε απλά 1 hp, οπότε:

$$dp[r][c] = 1$$

## 3. Γεμίζουμε τον πίνακα dp από κάτω δεξιά μέχρι την αφετηρία.

# ΠΑΡΑΔΕΙΓΜΑ ΕΚΤΕΛΕΣΗΣ



INF	INF	INF
INF	INF	INF
INF	INF	1

Red arrow from (1,3) to (2,3) labeled (5)

INF	INF	INF
INF	INF	INF
INF	1	1

Red arrow from (1,2) to (2,2) labeled (-8)

INF	INF	INF
INF	INF	INF
9	1	1

Red arrow from (2,3) to (3,3) labeled (-4)

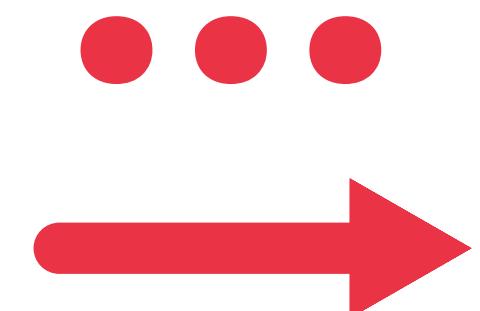
0	1	2
-2	-9	-4
-8	5	0

**grid[r][c]**

INF	INF	INF
INF	INF	5
9	1	1

Red arrow from (2,3) to (3,3) labeled (-9)

INF	INF	INF
INF	10	5
9	1	1



2	2	3
11	10	5
9	1	1

**dp[r][c]**



**ANY  
QUESTIONS?**

