Σχεδίαση και Ανάλυση Αλγορίθμων

Ενότητα 2.0

Πολυπλοκότητα Αλγορίθμων

Ασυμπτωτική Πολυπλοκότητα

Αναδρομικές Σχέσεις

Σταύρος Δ. Νικολόπουλος | 2017-18

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Webpage: www.cs.uoi.gr/~stavros

- **Ας ξεκινήσουμε…**

Αλγόριθμος

Ας ξεκινήσουμε…

- Για κάθε επιλύσιμο πρόβλημα Π, υπάρχει ένα σύνολο αλγορίθμων $\{A_1, A_2, \ldots, A_k\}$ που το επιλύουν, $k \geq 1$.
- Υποθέτουμε ότι διαθέτουμε έναν αλγόριθμο A_1 ο οποίος ταξινομεί 100.000 στοιχεία σε 30 sec, σε έναν μεσαίου μεγέθους Η/Υ.
- Είναι πολύ πιθανό !!!

  εάν χρησιμοποιήσουμε έναν άλλο αλγόριθμο ταξινόμησης A_2, ο χρόνος επίλυσης του προβλήματος (ταξινόμηση 100.000 στοιχείων) να αυξηθεί απίστευτα, έστω και αν ο Η/Υ στον οποίο εκτελείται ο A_2 είναι χιλιάδες φορές γρηγορότερος από αυτόν του A_1!!!

Δημιουργούνται επομένως Ερωτήματα !!!

- ✓ **Είναι** ο αλγόριθμος A αποτελεσματικότερος του αλγορίθμου B, δηλ. επιλύει ο αλγόριθμος Α το πρόβλημα Π σε λιγότερο χρόνο από τον αλγόριθμο Β;
- ✓ **Πόσο** θα αυξηθεί ο χρόνος εκτέλεσης του αλγορίθμου A εάν διπλασιάσουμε τα δεδομένα εισόδου (μέγεθος του προβλήματος);
- ✓ **Μπορώ** να χρησιμοποιήσω τον αλγόριθμο A όταν n>>, δηλ. όταν το μέγεθος n του προβλήματος Π που επιλύει είναι πολύ μεγάλο;
- ✓ **Υπάρχει** αλγόριθμος (ή μπορώ να σχεδιάσω έναν) ο οποίος επιλύει το πρόβλημα Π σε δεδομένο χρόνο T(n);

Τι θα Θέλαμε !!!

- ✓ Θα θέλαμε να διαθέτουμε αρκετούς αλγορίθμους $A_1, A_2, \ldots, A_k$ για ένα Π, ώστε να είμαστε σε θέση να επιλέξουμε τον καλύτερο!!!

Πρόβλημα Π

$$A_1, A_2, \ldots, A_k$$

- **Όλοι** οι αλγόριθμοι θα πρέπει να είναι **σωστοί** (correct) !!!
- **Όλοι** οι αλγόριθμοι παρουσιάζουν **θεωρητικό ενδιαφέρον** !!!

Τι θα Θέλαμε !!!

- ✓ Θα θέλαμε να διαθέτουμε αρκετούς αλγορίθμους A_1, A_2, …, A_k για ένα Π, ώστε να είμαστε σε θέση να επιλέξουμε τον καλύτερο!!!

Ας θυμηθούμε τι είναι Αλγόριθμος !

II. Αλγόριθμος είναι μια *υπολογιστική διαδικασία* για την *επίλυση ενός προβλήματος*, βασιζόμενη στην εκτέλεση μιας *καλώς ορισμένης ακολουθίας απλών ενεργειών*

Αλγόριθμος… Μέθοδος για την επίλυση ενός προβλήματος

Δομή Δεδομένων… Μέθοδος αποθήκευσης δεδομένων

Αλγόριθμος Πολλαπλασιασμού

- Είναι σε όλους γνωστός ο αλγόριθμος πολλαπλασιασμού δύο ακέραιων.
- Υπάρχει όμως και ένας άλλος, λιγότερο γνωστός, αλγόριθμος πολλαπλασιασμού (Al-Khwarizmi):

✓ **Παράδειγμα**

Έστω ότι θέλουμε να πολλαπλασιάσουμε το ζεύγος ακεραίων (45, 19).
Ο αλγόριθμος εργάζεται ως εξής:

περιττός	45	19	19
άρτιος	22	38	--
:	11	76	76
:	5	152	152
	2	304	--
	1	608	608
			855

Αλγόριθμος Πολλαπλασιασμού

- Είναι σωστός ο αλγόριθμος πολλαπλασιασμού του Al-Khwarizmi;

→	11	13	13
→	5	26	26
→	2	52	--
→	1	104	104
			143

```
                1  1  0  1     = 13
          x     1  0  1  1     = 11
          ------------------
                1  1  0  1     1101 x 1
             1  1  0  1        1101 x 1, << 1
          0  0  0  0           1101 x 0, << 2
    +  1  1  0  1              1101 x 1, << 3
    ------------------------
    1  0  0  0  1  1  1  1     = 143
```

- Ο αλγόριθμος του Al-Khwarizmi είναι ένα συναρπαστικό μείγμα δεκαδικού και δυαδικού συστήματος !!!

Αλγόριθμος ΜΚΔ

Αλγόριθμος του Ευκλείδη

- Ο αλγόριθμος του Ευκλείδη για ΜΚΔ:

  gcd(x,y) = gcd(y,x mod y)

  όπου **x≥ y** είναι θετικοί ακέραιοι.

Ευκλείδης (300 πΧ)

Παράδειγμα

```
Euclid(128,40)  =
  Euclid(40,8)  =
    Euclid(8,0) = 8
```

- Είναι σωστός ο αλγόριθμος εύρεσης ΜΚΔ του Ευκλείδη;
  - ✓ Αρκεί να δείξουμε τον απλούστερο κανόνα **gcd(x,y) = gcd(x-y,y)** από τον οποίο προκύπτει ο παραπάνω κανόνας με επανειλημμένη αφαίρεση του y από τον x.

Ορθότητα

- **Όλοι** οι αλγόριθμοι θα πρέπει να είναι **σωστοί** (correct) !!!
  - ✓ Για να αποδείξουμε ότι ένας αλγόριθμος είναι "λάθος" το μόνο που χρειαζόμαστε είναι να βρούμε ένα στιγμιότυπο του προβλήματος που επιλύει, για το οποίο δε δίνει σωστή απάντηση.
  - ✓ Από την άλλη μεριά, είναι πολύ πιο δύσκολο να αποδείξουμε την "ορθότητα" ενός αλγορίθμου.

Πολυπλοκότητα

- **Όλοι** οι αλγόριθμοι παρουσιάζουν **θεωρητικό ενδιαφέρον** !!!

✓ Όμως, πρακτικό ενδιαφέρον παρουσιάζουν οι αλγόριθμοι οι οποίοι είναι ***αποτελεσματικοί*** (efficient), δηλ. οι αλγόριθμοι οι οποίοι ελαχιστοποιούν:

- τον **χρόνο** που εκτελούνται
- τον **χώρο, τους επεξεργαστές, την ενέργεια** που χρησιμοποιούν

✓ Θα λέμε ότι ένας αλγόριθμος είναι:

- ***αποτελεσματικός*** (efficient) ⇔ ***μικρή πολυπλοκότητα*** (low-complexity)
- ***μη-αποτελεσματικός*** (inefficient) ⇔ ***μεγάλη πολυπλοκότητα*** (high-com.)

Πολυπλοκότητα

- **Στόχος μας είναι**:

  (α) η ανάλυση και ο υπολογισμός της ***πολυπλοκότητας χρόνου*** και ***χώρου*** (time and space complexity) των αλγορίθμων, και

  (β) ο έλεγχος για το εάν ένας αλγόριθμος είναι ***βέλτιστος*** (optimal), δηλ. εάν είναι ο πιο αποτελεσματικός για το πρόβλημα για το οποίο σχεδιάστηκε.

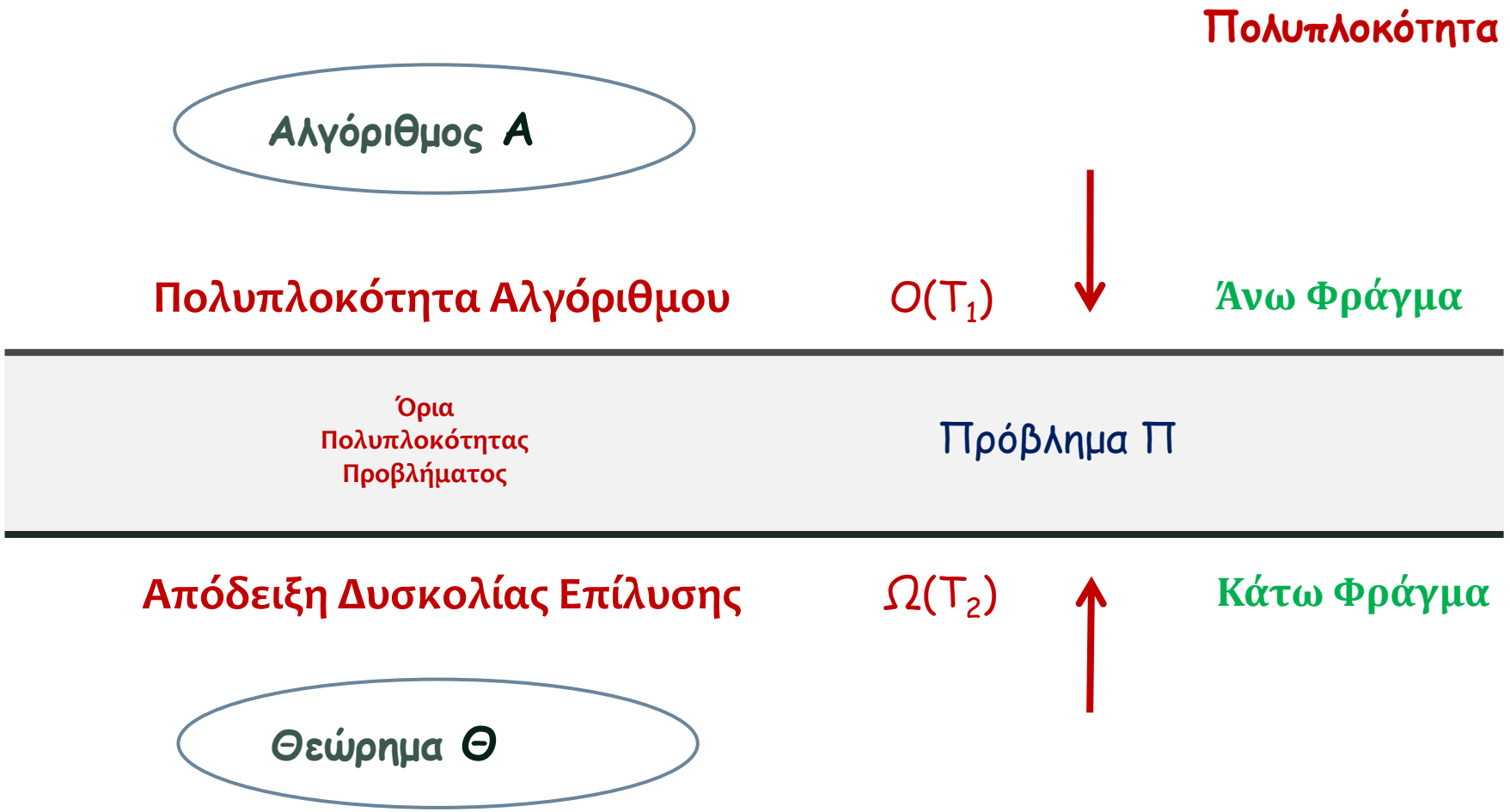
Πολυπλοκότητα Αλγόριθμου ⇒ **Άνω Φράγμα του Π**

II

Απόδειξη ενός ελάχιστου χρόνου για την επίλυση του προβλήματος Π ⇒ **Κάτω Φράγμα του Π**

Βέλτιστος (optimal) Αλγόριθμος

Βελτιστότητα !

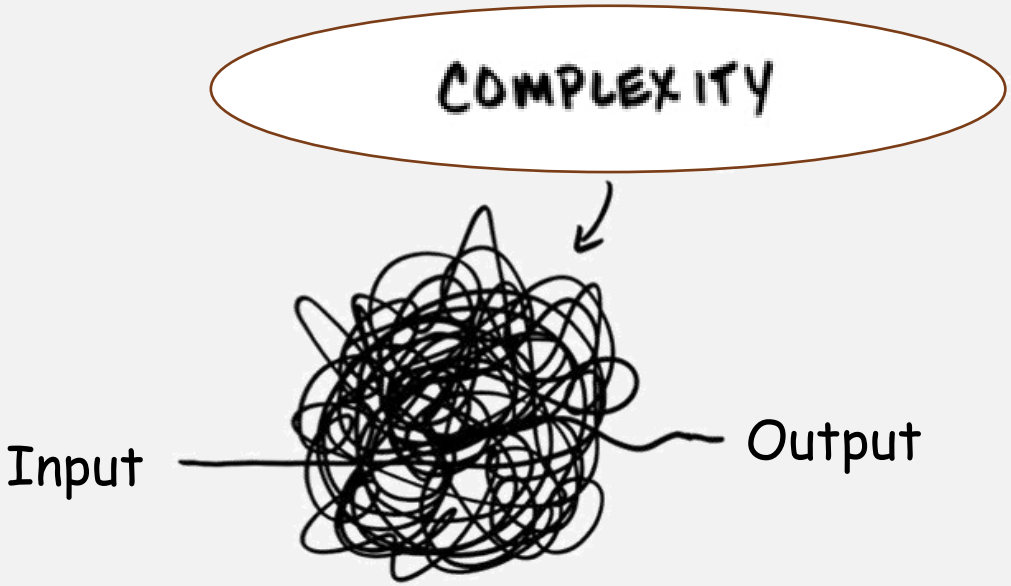


Βελτιστότητα !

Πολυπλοκότητα Αλγόριθμου

- **Ας συνεχίσουμε…**

Αλγόριθμος



Πολυπλοκότητα: Εμπειρική και Θεωρητική

- Διαθέτοντας ένα σύνολο αλγορίθμων $\{A_1, A_2, \ldots, A_k\}$, $k \geq 2$, οι οποίοι επιλύουν το ίδιο πρόβλημα Π, πως θα αποφασίσουμε ποιος αλγόριθμος είναι ο πιο αποτελεσματικός, δηλ. ποιος αλγόριθμος έχει την μικρότερη πολυπλοκότητα.

Υπάρχουν δύο προσεγγίσεις

- ***Εμπειρική*** (a posteriori)
- ***Θεωρητική*** (a priori)

Πολυπλοκότητα: Εμπειρική και Θεωρητική

- ***Εμπειρική*** (a posteriori)

  Η εμπειρική υπολογίζεται μετρώντας τον χρόνο εκτέλεσης του αλγορίθμου σε συγκεκριμένη μηχανή.

- ***Θεωρητική*** (a priori)

  Η θεωρητική καθορίζει μαθηματικά τον Χρόνο (κυρίως) και τον Χώρο που απαιτεί ο αλγόριθμος, ως συνάρτηση του μεγέθους των εξεταζόμενων στιγμιότυπων.

Πολυπλοκότητα: Εμπειρική και Θεωρητική

- ***Εμπειρική*** (a posteriori)

  Η εμπειρική υπολογίζεται μετρώντας τον χρόνο εκτέλεσης του αλγορίθμου σε συγκεκριμένη μηχανή.

- ***Θεωρητική*** (a priori)

  Στιγμιότυπο ⟺ Είσοδος αλγορίθμου

  Μέγεθος Στιγμιοτύπου ⟺ Μέγεθος Εισόδου Αλγορίθμου

Πολυπλοκότητα

Θεωρητική

Τα πλεονεκτήματα της θεωρητικής προσέγγισης του υπολογισμού της αποτελεσματικότητας ενός αλγορίθμου είναι ότι:

- δεν εξαρτάται από τον Η/Υ
- δεν εξαρτάται από τη γλώσσα προγραμματισμού
- δεν εξαρτάται από τις ικανότητες του προγραμματιστή

Πολυπλοκότητα

- Η απάντηση δίνεται από την "**αρχή της σταθερότητας**" (principle of invariance)

  Δυο διαφορετικές εφαρμογές ή υλοποιήσεις (implementations) του ίδιου αλγορίθμου Α, δηλ.

  - όταν εκτελούνται σε διαφορετικούς Η/Υ,
  - όταν γράφονται σε διαφορετικές γλώσσες,
  - όταν κωδικοποιούνται από διαφορετικούς προγραμματιστές, κλπ.

  δεν διαφέρουν στην αποτελεσματικότητά τους περισσότερο από σταθερό πολλαπλάσιο !!!

Πολυπλοκότητα

- Η απάντηση δίνεται από την "**αρχή της σταθερότητας**" (principle of invariance)

Δυο διαφορετικές εφαρμογές ή υλοποιήσεις (implementations) του ίδιου αλγορίθμου Α, δεν διαφέρουν στην αποτελεσματικότητά τους περισσότερο από σταθερό πολλαπλάσιο !!!

Εάν E_1 είναι η αποτελεσματικότητα μιας εφαρμογής ενός Α και E_2 η αποτελεσματικότητα μιας άλλης εφαρμογής του, τότε ισχύει:

$$E_1 = c \cdot E_2$$

όπου **c** μια σταθερά.

Πολυπλοκότητα

- Η απάντηση δίνεται από την "**αρχή της σταθερότητας**" (principle of invariance)

Σήμερα μας ενδιαφέρει κυρίως η αποτελεσματικότητα (χρόνος) εκτέλεσης των αλγορίθμων μας!!!

Εστιάζω στον Χρόνο…

Εάν $t_1(n)$ και $t_2(n)$ είναι οι χρόνοι εκτέλεσης δύο εφαρμογών του ίδιου αλγόριθμου Α, τότε υπάρχει πάντα σταθερά **c** τέτοια ώστε:

$$t_1(n) = c \cdot t_2(n)$$

όπου **n** το μέγεθος του προβλήματος που επιλύουν.

Πολυπλοκότητα

Χρόνος εκτέλεσης

Θα λέμε ότι ο αλγόριθμος Α **εκτελείται σε χρόνο t(n)**, εάν υπάρχει θετική σταθερά **c** και μια εφαρμογή (ή υλοποίηση) του Α τέτοια ώστε:

για κάθε είσοδο μήκους ***n*** *(μέγεθος προβλήματος), ο χρόνος εκτέλεσης του αλγορίθμου να* ***φράσσεται άνω*** *από* ***c · t(n)***

✓ Αργότερα θα δούμε ότι η παραπάνω έκφραση αποτελεί τον **ασυμπτωτικό συμβολισμό** (asymptotic notation) της πολυπλοκότητας των αλγορίθμων.

Πολυπλοκότητα

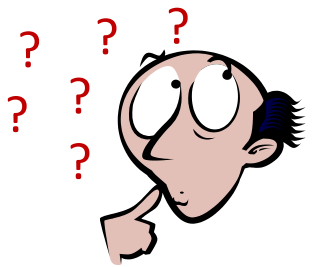
- **Εύλογο ερώτημα !!!**

ΠΟΙΑ ΜΟΝΑΔΑ θα χρησιμοποιήσουμε για να εκφράσουμε θεωρητικά την αποτελεσματικότητα ενός αλγορίθμου;

Για παράδειγμα,

θα την εκφράσουμε σε **sec**; θα την εκφράσουμε σε **βήματα**; κλπ.

✓ Θα χρησιμοποιήσουμε:
Θεμελιώδη ή **Βασική Πράξη** (elementary operation).



Βασική Πράξη

- Βασική πράξη ονομάζεται:

> η πράξη της οποίας ο χρόνος εκτέλεσης φράσσεται άνω (bounded above) από μια σταθερά η οποία εξαρτάται μόνο από τη χρησιμοποιούμενη εφαρμογή (Η/Υ, γλώσσα προγραμματισμού, ικανότητα προγραμματιστή, κλπ.)

- Επομένως, για την ανάλυση των αλγορίθμων μας θα χρησιμοποιήσουμε:

> **ΜΟΝΟ** το **πλήθος των ΒΠ** που εκτελούνται από τον αλγόριθμο !!!
>
> **ΌΧΙ** τον **ακριβή χρόνο** που απαιτεί κάθε μια από τις πράξεις αυτές !!!

Βασική Πράξη

- Προφανώς, ισχύει:

  - $\textit{Πλήθος Βασικών Πράξεων} = f(\textit{Μέγεθος Εισόδου})$

Παράδειγμα

Θεωρούμε το πρόβλημα της εύρεσης του ελάχιστου (min) στοιχείου μιας ακολουθίας S, δηλ.

$$x \leftarrow \min\{S[i] \mid 1 \le i \le n\}$$ … **n-1 βασικές πράξεις**

Είναι προφανές ότι, το πλήθος των βασικών πράξεων αυξάνει με την αύξηση του n.

Βασική Πράξη

- Προσοχή !!!... Υπάρχουν όμως μαθηματικές πράξεις οι οποίες είναι αρκετά πολύπλοκες για να χαρακτηριστούν βασικές πράξεις.

Παράδειγμα

Το θεώρημα του Wilson το οποίο μας επιτρέπει να πούμε ότι ο ακέραιος **n** είναι πρώτος αριθμός εάν διαιρείται ακριβώς με το **(n-1)!+1**

```
function Wilson(n)
begin
      if n divides (n-1)!+1 exactly then n is a prime
                                    else n is not a prime
end;
```

Βασική Πράξη

Παράδειγμα

Πρόβλημα διερεύνησης εάν ένα δοθέν στοιχείο x ανήκει σε μια ακολουθία S μήκους n ή όχι.

```
Algorithm Linear-Search(S,x)
begin
    index ← 1
    while S[index] ≠ x and index ≤ n do
            index ← index + 1
    end
    if index > n then return "x ∉ S"
                 else return index
end
```

?

x = 5

α) [3,1,5,2,9,7]

b) [9,3,7,2,1,5]

c) [4,7,1,2,9,3]

d) [7,5,2,4,3,1]

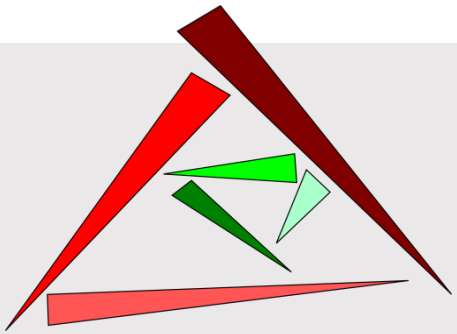
Βασικές πράξεις:

S[index] ≠ x

index ≤ n

Χειρίστη & Μέση Πολυπλοκότητα

Πολυπλοκότητα Αλγόριθμου



- **Χειρότερης περίπτωσης** (worst case):
  με αυτήν ασχολούμαστε συνήθως.

- **Μέσης περίπτωσης** (average case):
  με βάση κατανομή πιθανότητας στιγμιότυπων (instances) του προβλήματος. Συνήθως δύσκολο να οριστεί σωστά.

- Αντισταθμιστική (amortized):
  εκφράζει την μέση αποδοτικότητα σε μια σειρά επαναλήψεων του αλγορίθμου.

Πολυπλοκότητα Χρόνου

Επειδή, ο χρόνος εκτέλεσης μιας Βασικής Πράξης φράσσεται άνω από μια σταθερά, ισχύει:

- $\textit{Πολυπλοκότητα Χρόνου} = f(\textit{Μέγεθος Εισόδου})$

  - **Ανάλυση Χείριστης Περίπτωσης** (Worst-case Analysis)
  - **Ανάλυση Μέσης Περίπτωσης** (Average-case Analysis)

Χείριστη Περίπτωση (Worst-case Analysis)

- Έστω D_n το σύνολο όλων των εισόδων μεγέθους n.

  Για κάθε I∈D_n, ας είναι t(I) το πλήθος των Βασικών Πράξεων που εκτελούνται από τον αλγόριθμο με είσοδο I, δηλ.

$$t : D_n \rightarrow N$$

- Ορίζουμε την πολυπλοκότητα Χείριστης Περίπτωσης (Worst-case Complexity) ως εξής:

$$W(n) = \max \{ t(I) \mid I \in D_n \}$$

Μέση Περίπτωσης (Average-case Analysis)

- Υποθέτουμε ότι μπορούμε να αντιστοιχίσουμε μια **πιθανότητα** $p(\mathrm{I})$ σε κάθε είσοδο $\mathrm{I} \in D_n$, έτσι ώστε:

$$p(\mathrm{I}) = \textit{Πιθανότητα εμφάνισης της εισόδου } \mathrm{I}$$

- Ορίζουμε την πολυπλοκότητα Μέσης-Περίπτωσης (Average-case Complexity) ως εξής:

$$A(n) = \sum_{I \in D_n} p(I)t(I)$$

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Παράδειγμα 1. (Γραμμική Διερεύνηση – Linear Search)

Δοθείσης μιας ακολουθίας στοιχείων S μήκους n και ενός στοιχείου x,
θέλουμε να ελέγξουμε εάν το στοιχείο x ανήκει στην S και σε ποια θέση ή όχι.

```
Algorithm Linear-Search(S,x)
begin
    index ← 1
    while S[index] ≠ x and index ≤ n do
            index ← index + 1
    end
    if index > n then index ← 0
    return index
end
```

Βασικές Πράξεις: `S[index] ≠ x`

`index ≤ n`

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Παράδειγμα 1. (Γραμμική Διερεύνηση – Linear Search)

Δοθείσης μιας ακολουθίας στοιχείων S μήκους n και ενός στοιχείου x,
θέλουμε να ελέγξουμε εάν το στοιχείο x ανήκει στην S και σε ποια θέση ή όχι.

```
Algorithm Linear-Search(S,x)
begin
    index ← 1
    while S[index] ≠ x and index ≤ n do
          index ← index + 1
    end
    if index > n then index ← 0
    return index
end
```

Βασικές Πράξεις: `S[index] ≠ x`

`index ≤ n`

Αρκεί μία ΜΟΝΟ Βασική Πράξη:

`S[index] ≠ x ή index ≤ n`

Γιατί;

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Χειρίστη Πολυπλοκότητα

```
Algorithm Linear-Search(S,x)
begin
    index ← 1
    while S[index] ≠ x and index ≤ n do
          index ← index + 1
    end
    if index > n then index ← 0
    return index
end
```

x = 5

α) [3,1,5,2,9,7]

b) [9,3,7,2,1,5]

c) [4,7,1,2,9,3]

d) [7,5,2,4,3,1]

Βασική Πράξη: **S[index] ≠ x**

Ανάλυση Χείριστης-Περίπτωσης: $W(n) = n$

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Μέση Πολυπλοκότητα

Υποθέτουμε ότι ισχύουν τα κάτωθι:

- όλα τα στοιχεία είναι διαφορετικά,
- το στοιχείο αναζήτησης υπάρχει στην S, δηλ. x ∈ S,
- όλες οι θέσεις εμφάνισης του x έχουν την ίδια πιθανότητα.

Έστω I_i η είσοδος για την οποία ισχύει x = S[i], 1 ≤ i ≤ n. Τότε

$$p(I_i) = 1/n \qquad \text{και} \qquad t(I_i) = i$$

Επομένως

$$A(n) = \sum_{i=1}^{n} p(I_i)t(I_i) = \sum_{i=1}^{n} \frac{1}{n} i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Μέση Πολυπλοκότητα

Ας αποσύρουμε τώρα την υπόθεση ότι το στοιχείο αναζήτησης x ∈ S.

Σε αυτή την περίπτωση θα συμβολίζουμε με I_e κάθε είσοδο μήκους n η οποία δεν περιέχει το στοιχείο x, δηλ. x ≠ S[i] ∀ i, 1 ≤ i ≤ n.

Προφανώς ισχύει

$$t(I_e) = n$$

Έστω q η πιθανότητα το ζητούμενο στοιχείο x να βρίσκεται στην S. Τότε

$$p(I_i) = q \cdot (1/n) \qquad p(I_e) = 1 - q$$

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Μέση Πολυπλοκότητα

Επομένως για τη μέση περίπτωση θα ισχύει:

$$A(n) = (\sum_{i=1}^{n} p(I_i)t(I_i)) + p(I_e)t(I_e)$$

$$= (\sum_{i=1}^{n} \frac{q}{n} i) + (1-q)n \quad = \quad (\frac{q}{n} \sum_{i=1}^{n} i) + (1-q)n$$

$$= \frac{q}{n} \frac{n(n+1)}{2} + (1-q)n \quad = \quad \frac{q(n+1)}{2} + (1-q)n$$

Παράδειγμα Χειρίστης & Μέσης Περίπτωσης

Μέση Πολυπλοκότητα

- ✓ Εάν q=1 (δηλ. το στοιχείο x βρίσκεται στην S), τότε

$$A(n) = (n+1)/2 = (1/2)n + 1/2$$

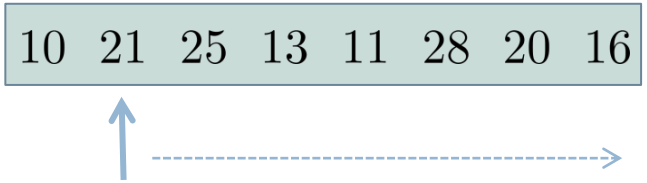
- ✓ Εάν q=1/2, τότε

$$A(n) = (n+1)/4 + n/2 = (3/4)n + 1/4$$

Επομένως, στη περίπτωση αυτή, πρέπει να διερευνηθούν κατά μέσο όρο περίπου τα ¾ της S.

1. Εύρεση Max στοιχείου (I)

```
Algorithm Find_Max-I
begin
    max ← S[1]; posmax ← 1
    for i ← 2 to n do
        if S[i]>max then
            max ← S[i]
            posmax ← i
        end
    end
    return max, posmax
end
```




```
max ← 21; posmax ← 1
return 28, 5
```

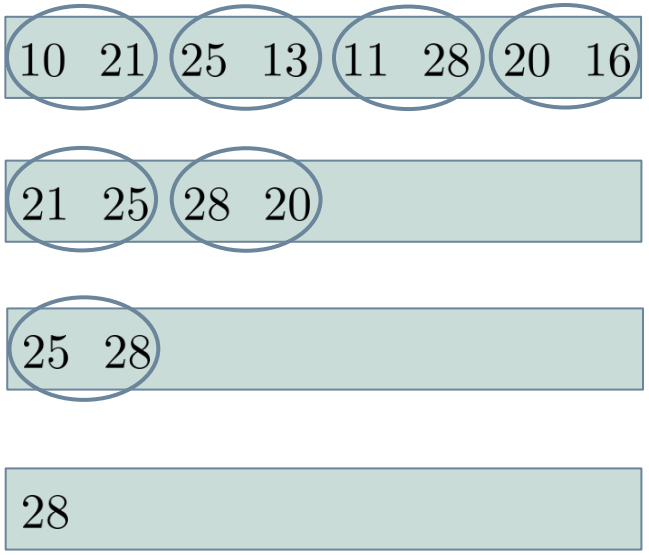
Βασική Πράξη: `S[i] > max`

Ανάλυση Χείριστης-Περίπτωσης: $W(n) =$ `n-1`

Ανάλυση Μέσης-Περίπτωσης: $A(n) =$ `n-1`

2. Εύρεση Max στοιχείου (II)

```
Algorithm Find_Max-II
begin
      for i ← 1 to logn do
          k ← 1
          for j ← 1 to n/2^(i-1) step 2 do
              if S[j] < S[j+1]
                  then S[k] ← S[j+1]
                  else S[k] ← S[j]
              k ← k+1;
          end
      end
      return max ← S[1]
end
```



3. Εύρεση Max στοιχείου (III)

```
Algorithm Find_Max-III
begin
      for i ← 1 to logn do
          k ← 1
          for j ← 1 to n/2^(i-1) step 2 do
              if S[j] < S[j+1]
                  then S[k] ← S[j+1]
                  else S[k] ← S[j]
              k ← k+1;
          end
      end
      return max ← S[1]
end
```

10	21	25	13	11	28	20	16

28

ΒΠ: **S[j] > S[j+1]**

Χείριστη-Περίπτωση:

$W(n) = \mathbf{n-1}$

n/2+n/4+n/8+…+1 =
n(1/2+1/4+…+1/n) =
n(n-1)/n = n-1

Μέση-Περίπτωση: $A(n) = \mathbf{n-1}$

4. Εύρεση Min και Max στοιχείων

```
Algorithm Find_Min_Max
begin
      min ← max ← S[1]
      for i ← 2 to n do
          if S[i] < min then
              min ← S[i];
          else
              if S[i] > max then
                  max ← S[i]
          end
      end
      return min, max
end
```

$I_1 = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$

$I_2 = 9\ 5\ 8\ 4\ 7\ 3\ 6\ 2\ 1$

ΒΠ: **S[i] < min, S[i] > max**

Χείριστη-Περίπτωση:

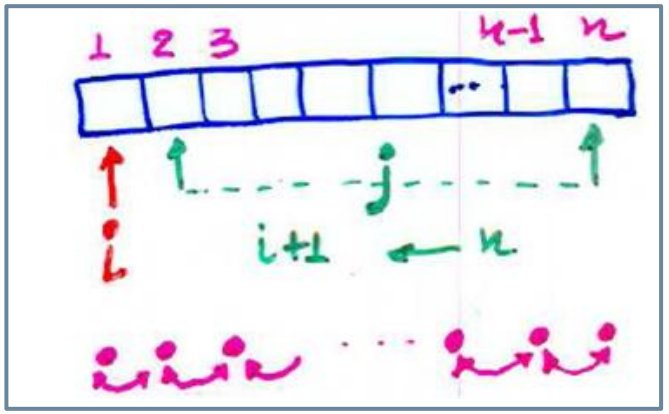
$W(n) =$ **(n-1)+(n-1)= 2n-2**

Μέση-Περίπτωση:

$A(n) =$ **(n-1)+1/2(n-1)= 3/2(n-1)**

5. Ταξινόμηση (Αλγόριθμος Φυσαλίδας)

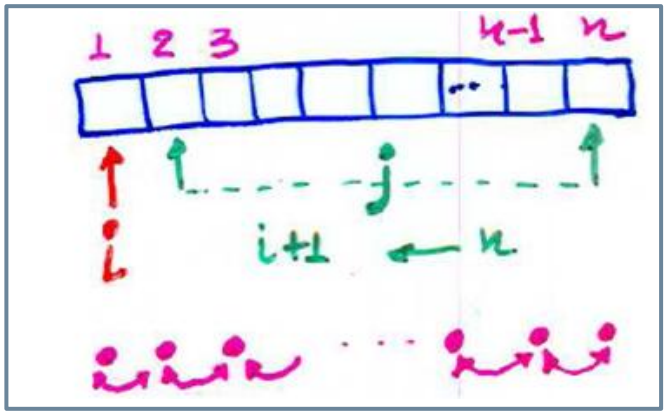
```
Algorithm Bubble-sort
begin
     for i ← 1 to n-1 do
       for j ← n downto i+1 do
          if S[j] < S[j-1] then
             Swap(S[j], S[j-1])
          end
     end
     return S
end
```



Ορθότητα

5. Ταξινόμηση (Αλγόριθμος Φυσαλίδας)

```
Algorithm Bubble-sort
begin
     for i ← 1 to n-1 do
       for j ← n downto i+1 do
          if S[j] < S[j-1] then
             Swap(S[j], S[j-1])
          end
     end
     return S
end
```



Πολυπλοκότητα

ΒΠ: `S[j] < S[j-1]`

5. Ταξινόμηση (Αλγόριθμος Φυσαλίδας)

```
Algorithm Bubble-sort
begin
     for i ← 1 to n-1 do
       for j ← n downto i+1 do
          if S[j] < S[j-1] then
             Swap(S[j], S[j-1])
          end
     end
     return S
end
```

ΒΠ: `S[j] < S[j-1]`

Χείριστη-Περίπτωση:

$$W(n) = \sum_{i=1}^{n-1}(n-i) =$$

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 =$$

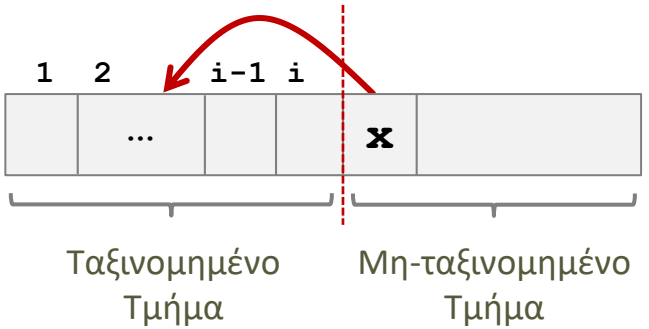
$$\frac{n(n-1)}{2}$$

Μέση-Περίπτωση:

$A(n) =$ `n(n-1)/2`

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

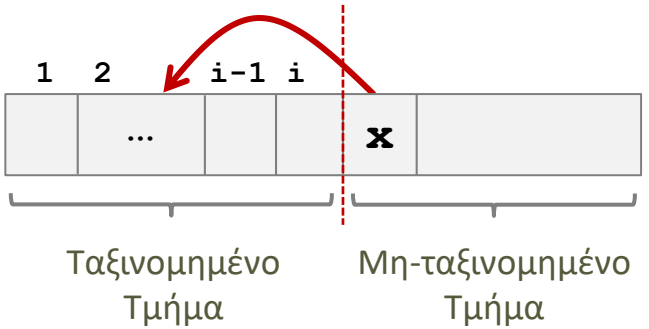
```
Algorithm Insertion-sort
begin
     for i ← 2 to n do
       x ← S[i];
       j ← i-1
       while x < S[j] and j >= 1 do
             S[j+1] ← S[j]
             j ← j-1
       end
       S[j+1] ← x
     end
     return S
end
```



Ορθότητα

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

```
Algorithm Insertion-sort
begin
     for i ← 2 to n do
       x ← S[i];
       j ← i-1
       while x < S[j] and j >= 1 do
             S[j+1] ← S[j]
             j ← j-1
       end
       S[j+1] ← x
     end
     return S
end
```

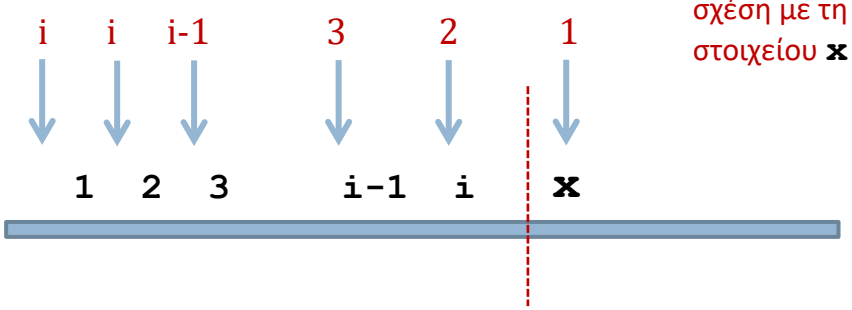


Πολυπλοκότητα

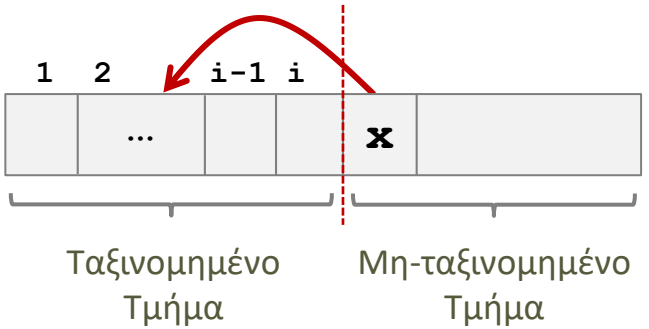
ΒΠ: `x < S[j]`

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Χειρίστη-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**



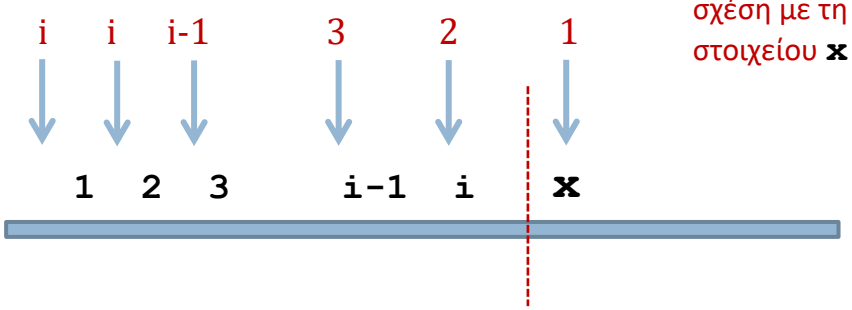
$$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Χειρίστη-Περίπτωση:

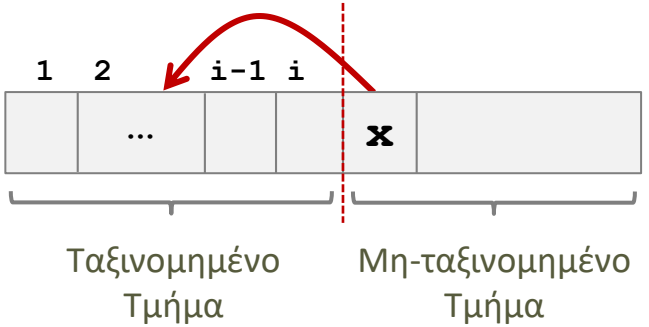
W(n) = **n(n-1)/2**

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**



Το στοιχείο **x** μπορεί να μεταφερθεί σε **i+1** θέσεις !!!

Η πιθανότητα το **x** να μεταφερθεί σε μία από αυτές τις θέσεις είναι **1/i+1** !!!

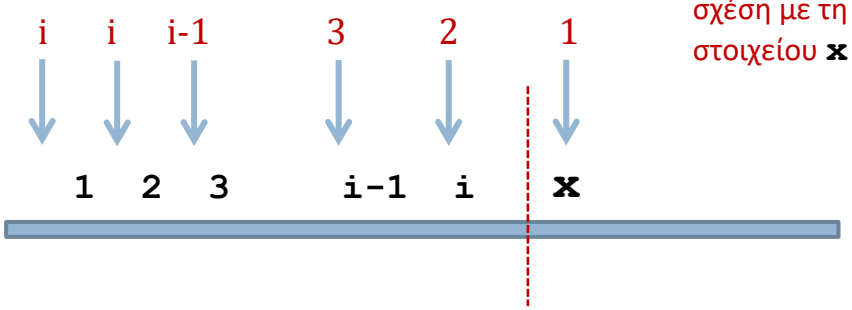
Υποθέτουμε ότι:

(α) τα στοιχεία του S είναι όλα διαφορετικά μεταξύ του, και

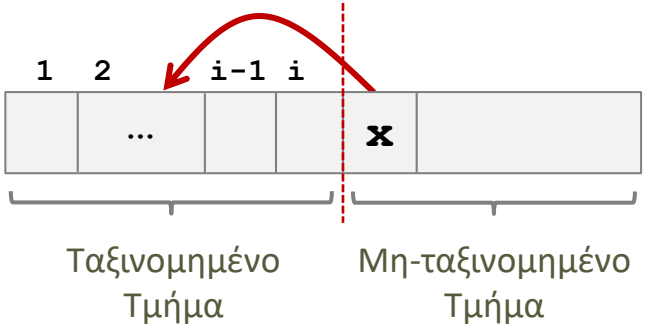
(β) όλες οι είσοδοι έχουν την ίδια πιθανότητα εμφάνισης

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

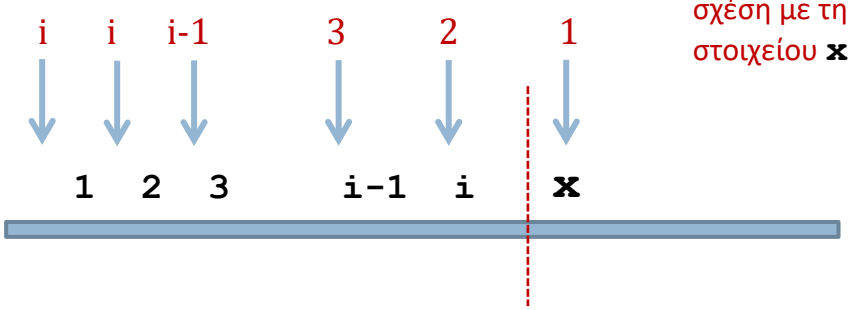


Επομένως,
με πιθανότητα **1/i+1** καταλαμβάνει την **k-οστη** θέση και απαιτεί **k** συγκρίσεις:

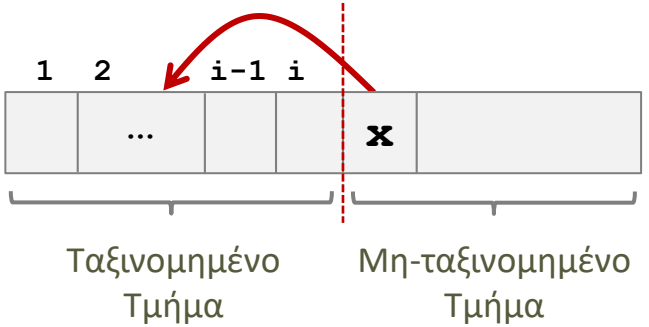
$$A(n) = \sum_{k=1}^{i} \frac{1}{i+1}\ k\ +\ \frac{1}{i+1}\ i$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

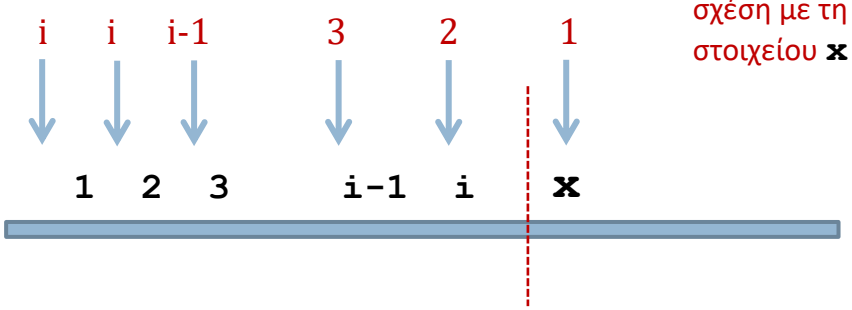


Επομένως,
με πιθανότητα **1/i+1** καταλαμβάνει την **k-οστη** θέση και απαιτεί **k** συγκρίσεις:

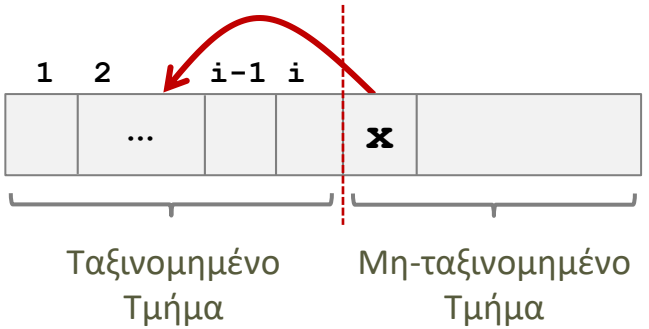
$$A(n) = \sum_{k=1}^{i} \frac{1}{i+1}\, k \;+\; \frac{1}{i+1}\, i \;=\; \frac{i}{2} + 1 - \frac{1}{i+1}$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

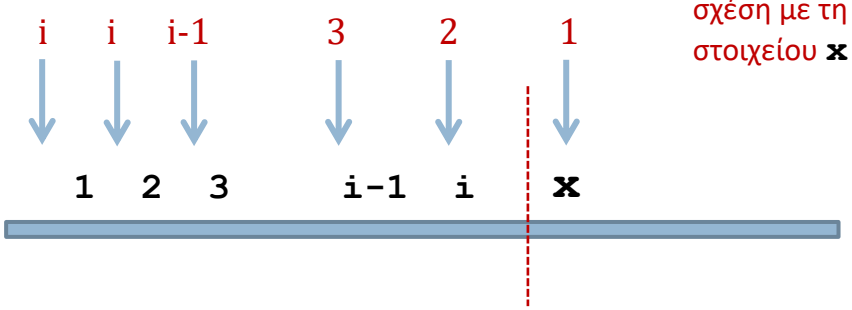


Τώρα, προσθέτω τα κόστη όλων των n-1 παρεμβολών :

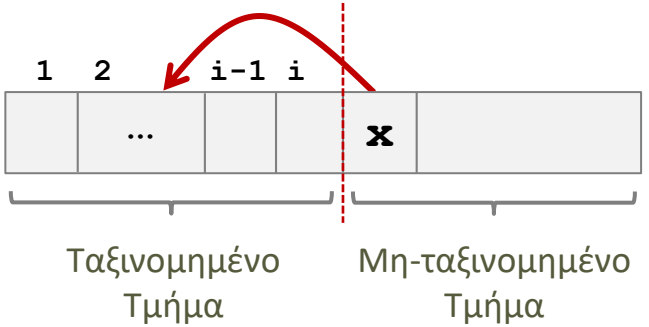
$$A(n) = \sum_{i=1}^{n-1} (\frac{i}{2} + 1 - \frac{1}{i+1})$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

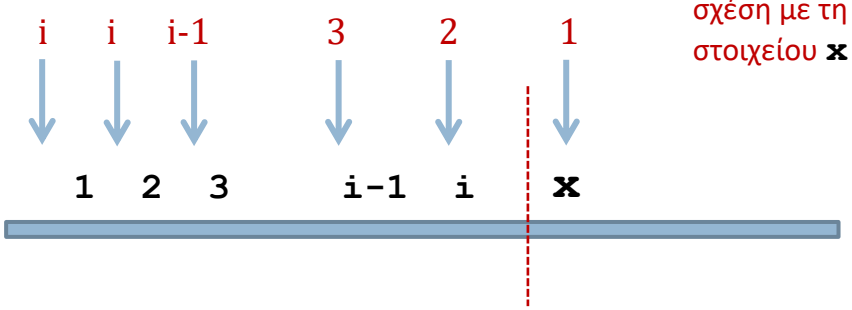


Τώρα, προσθέτω τα κόστη όλων των n-1 παρεμβολών :

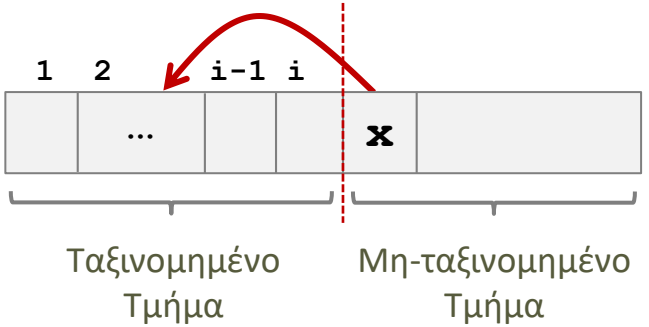
$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right)$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**



Τώρα, προσθέτω τα κόστη όλων των n-1 παρεμβολών :

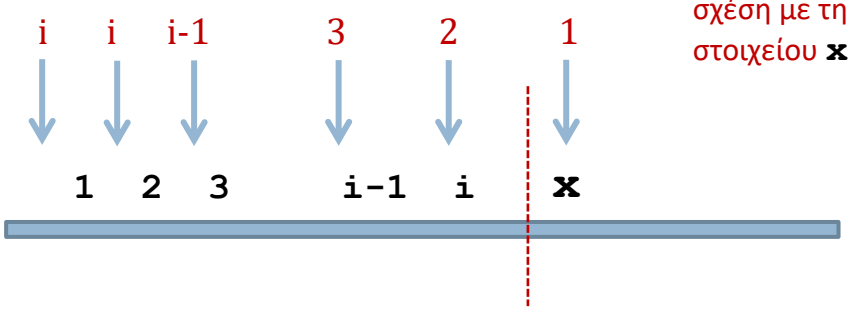
$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n} \frac{1}{j}$$

Ισχύει:

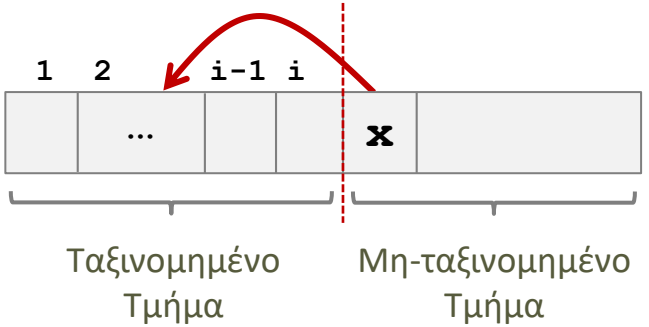
$$\sum_{j=1}^{n} \frac{1}{j} \approx \ln n$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

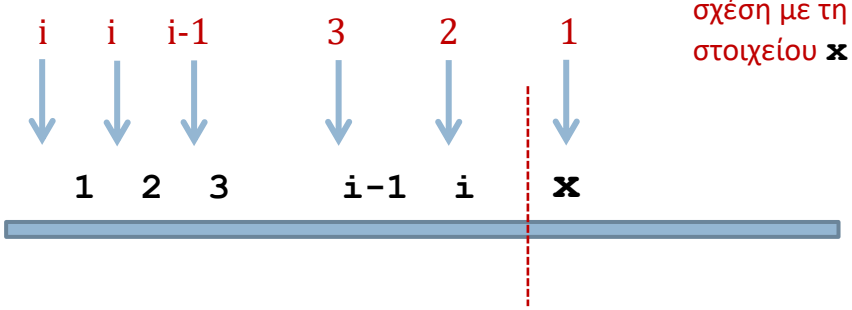


Αντικαθιστώντας, παίρνουμε:

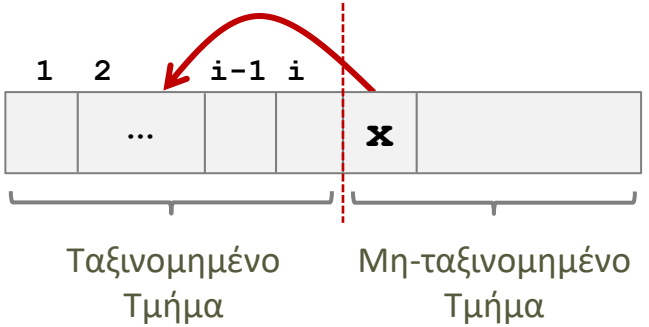
$$A(n) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n} \frac{1}{j} \approx \frac{n(n-1)}{4} + n - 1 - \ln n$$

6. Ταξινόμηση (Αλγόριθμος Παρεμβολής)

Μέση-Περίπτωση



Πλήθος συγκρίσεων σε σχέση με τη θέση του στοιχείου **x**

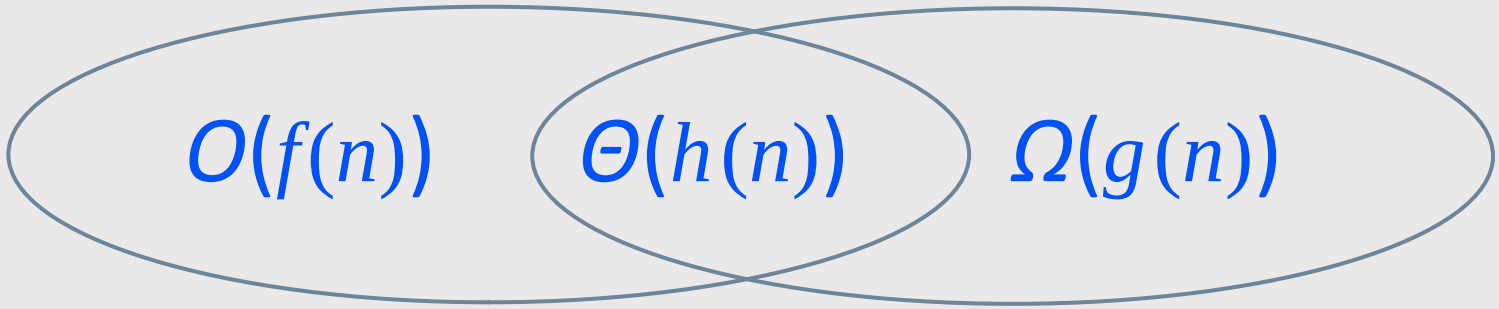


Αντικαθιστώντας, παίρνουμε:

$$A(n) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n} \frac{1}{j} \approx \frac{n(n-1)}{4} + n - 1 - \ln n \approx \frac{n^2}{4}$$

Ασυμπτωτική Πολυπλοκότητα

Πολυπλοκότητα Αλγόριθμου



Τι είδαμε ως τώρα ;

Πλήθος Βασικών Πράξεων ενός αλγόριθμου – Πολυπλοκότητες W(n) και A(n)

Ερώτημα… **Μήπως σε αυτήν την προσέγγιση της πολυπλοκότητας υπεισέρχονται στον υπολογισμό μας παράγοντες οι οποίοι δεν μας διευκολύνουν να εκτιμήσουμε την αποτελεσματικότητα ενός αλγόριθμου;**

Επόμενο Εύλογο Ερώτημα… **Είναι οι παράγοντες αυτοί τόσο σημαντικοί ώστε να μην μπορούν να αγνοηθούν;**

Ας αναφερθούμε στο πρώτο ερώτημα !!!

Ερώτημα... **Μήπως σε αυτή την προσέγγιση υπεισέρχονται παράγοντες οι οποίοι δεν μας διευκολύνουν στην εκτίμησή μας ;**

Έστω... Αλγόριθμος **A** $\Rightarrow$ $W(n) = \mathbf{n^3 / 4}$

Αλγόριθμος **B** $\Rightarrow$ $W(n) = \mathbf{10\ n^2}$

Ισχύει... $\mathbf{n^3 / 4 < 10\ n^2}$ για $\mathbf{n < 40}$

Τότε Ο αλγόριθμος A είναι αποτελεσματικότερος του B !!!

Όμως... Χωρίς δυσκολία, παρατηρούμε ότι για μέγεθος εισόδου n >> (αρκετά μεγάλο) *ο αλγόριθμος B είναι πολύ πιο αποτελεσματικός από τον A*.

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

n	Cray-1 Fortran[a] $3n^3$ νανο-δευτερόλεπτα	TRS-80 Basic[b] $19.500.000n$ νανο-δευτερόλεπτα
10		
100		
1000	?	
2500		
10.0000		
1.000.000		

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Ασυμπτωτική Πολυπλοκότητα

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Ασυμπτωτική Πολυπλοκότητα

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Ασυμπτωτική Πολυπλοκότητα

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα
2500	50.0 δευτερόλεπτα	50.0 δευτερόλεπτα

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Ασυμπτωτική Πολυπλοκότητα

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα
2500	50.0 δευτερόλεπτα	50.0 δευτερόλεπτα
10.0000	49.0 λεπτά	3.2 λεπτά

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Ασυμπτωτική Πολυπλοκότητα

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα
2500	50.0 δευτερόλεπτα	50.0 δευτερόλεπτα
10.0000	49.0 λεπτά	3.2 λεπτά
1.000.000	95.0 χρόνια	5.4 ώρες

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

	Cray-1 Fortran[a]	TRS-80 Basic[b]
n	$3n^3$ νανο-δευτερόλεπτα	$19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα
2500	50.0 δευτερόλεπτα	50.0 δευτερόλεπτα ← $\forall n \geq n_0 = 2500$
10.0000	49.0 λεπτά	3.2 λεπτά
1.000.000	95.0 χρόνια	5.4 ώρες

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

- **Ας αναφερθούμε στο πρώτο ερώτημα !!!**

Από το παραπάνω παράδειγμα γίνεται σαφές ότι:

Χρειαζόμαστε έναν τρόπο να εξασφαλίσουμε μερικές
κατηγορίες συναρτήσεων
οι οποίες

ΕΞΑΛΕΙΦΟΥΝ
ΜΗ-ΣΗΜΑΝΤΙΚΟΥΣ ΠΑΡΑΓΟΝΤΕΣ

όπως σταθερές, μικρού μεγέθους εισόδους, κλπ.

Συμβολισμοί Συνόλων

- $N = \{0, 1, 2, \ldots\}$
- $N^+ = \{1, 2, \ldots\}$
- R = Το σύνολο των πραγματικών
- R^+ = Το σύνολο των πραγματικών > 0
- R^* = Το σύνολο των πραγματικών ≥ 0

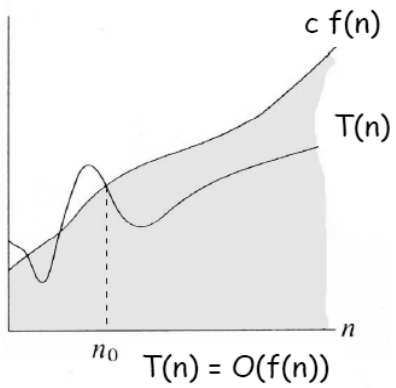
Συντομογραφίες Ο, Ω και Θ

- **Ορισμός:** Έστω μια συνάρτηση $f : N \to R^*$

$$\mathcal{O}(f(n)) = \{T : N \to R^* | \ (\exists\, c \in R^+)(\exists\, n_0 \in N)(\forall n \geq n_0)[T(n) \leq c\ f(n)]\}$$

Ορίζουμε να είναι το σύνολο όλων των συναρτήσεων T : N⟶R*, οι οποίες φράσσονται άνω από την f(n), με την προϋπόθεση ότι η τιμή του n είναι μεγαλύτερη από ή ίση με ένα κατώφλι n_0.

- Το σύνολο **O(f(n))** ονομάζεται **"τάξη της f(n)"**
- Εάν T(n) ∈ O(f(n)) θα λέμε ότι T(n) είναι τάξεως f(n)
- Συχνά, T(n) ∈ O(f(n)) γράφεται T(n) = O(f(n))



Συντομογραφίες O, Ω και Θ

- Σύμφωνα με την *αρχή της σταθερότητας*, εάν μια εφαρμογή E_1 ενός αλγορίθμου Α *εκτελείται σε χρόνο* $\mathbf{T_1(n)}$, τότε οποιαδήποτε άλλη εφαρμογή E_2 του ίδιου αλγορίθμου Α εκτελείται σε χρόνο τάξεως $\mathbf{T_2(n) = c\ T_1(n)}$, όπου **c** σταθερά και **n** το μέγεθος του προβλήματος.

- Θα λέμε ότι "*ο αλγόριθμος εκτελείται σε χρόνο T(n) τάξης f(n)*" εάν η συνάρτηση $f : N \longrightarrow R^*$ είναι τέτοια ώστε

$$T(n) \in O(f(n))$$

- Επίσης, θα είναι σωστό να γράφουμε $T(n) = n^3 + 3n^2 + n + 8 \in O(f(n))$

Ερώτηση: $T(n) \in O(n^3)$ **?**
$T(n) \in O(n^4)$ **?**
$T(n) \in O(n^2)$ **?**

Συντομογραφίες Ο, Ω και Θ

- Γενικά, εκφράζουμε την τάξη του χρόνου εκτέλεσης T(n) ενός αλγορίθμου χρησιμοποιώντας την απλούστερη δυνατή συνάρτηση f για την οποία ισχύει T(n) ∈ O(f(n)).

- Εάν T(n) ∈ O(f(n)), τότε

  "*η συνάρτηση T(n) αυξάνει όχι γρηγορότερα από τη συνάρτηση f(n)*"

  Ισχύει

$$T(n) \in \mathcal{O}(f(n)) \qquad \text{εάν-ν} \qquad \lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$$

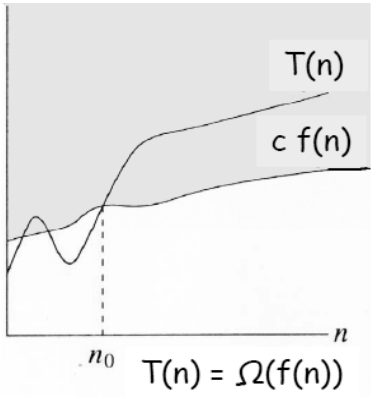
Συντομογραφίες Ο, Ω και Θ

- **Ορισμός:** Έστω μια συνάρτηση $f : N \to R^*$

$$\Omega(f(n)) = \{T : N \to R^* |\ \ (\exists\ \ c \in R^+)(\exists\ \ n_0 \in N)(\forall n \geq n_0)[T(n) \geq c\ f(n)]\}$$

Ορίζουμε να είναι το σύνολο όλων των συναρτήσεων T : N⟶R*, οι οποίες φράσσονται κάτω από την f(n), με την προϋπόθεση ότι η τιμή του n είναι μεγαλύτερη από ή ίση με ένα κατώφλι n_0.

- Το σύνολο **Ω(f(n))** ονομάζεται "το σύνολο ωμέγα της f(n)" (the omega of f(n)).
- Εάν T(n) ∈ *Ω*(f(n)), τότε "*η T(n) αυξάνει τουλάχιστον όσο γρήγορα αυξάνη και η συνάρτηση f(n)*"



Συντομογραφίες Ο, Ω και Θ

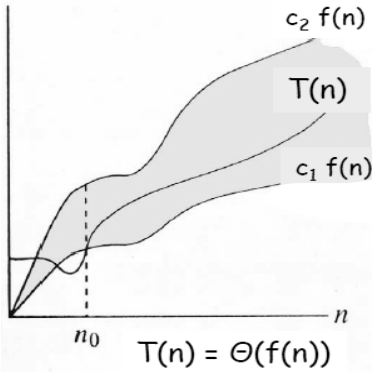
- **Ορισμός:** Έστω μια συνάρτηση $f : N \to R^*$

$$\Theta(f(n)) = \{T : N \to R^* |\ (\exists\ c_1, c_2 \in R^+)(\exists\ n_0 \in N)(\forall n \geq n_0)\,[c_1\ f(n) \leq T(n) \leq c_2\ f(n)]\}$$

Ορίζουμε να είναι το σύνολο όλων των συναρτήσεων T(n), δηλ. $T : N \to R^*$, οι οποίες φράσσονται άνω και κάτω από την f(n), με την προϋπόθεση ότι η τιμή του n είναι μεγαλύτερη από ή ίση με ένα κατώφλι n_0.

- Εάν $T(n) \in \Theta(f(n))$, τότε

"*η συνάρτηση T(n) αυξάνει ίδια με τη συνάρτηση f(n)*"



Συντομογραφίες Ο, Ω και Θ

- **Ισοδύναμος Ορισμός:** Ορίζουμε

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

και ονομάζουμε το σύνολο "ακριβή τάξη της f(n)" ("exact order of f(n)").

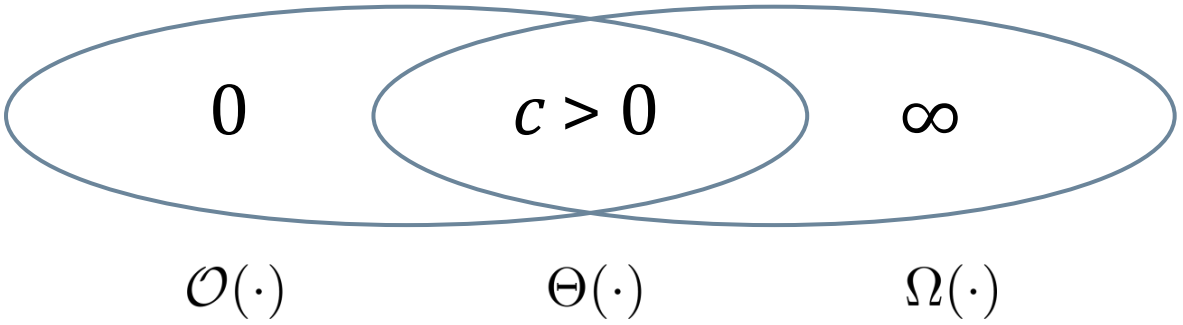
Ισχύει

$$T(n) \in \Omega(f(n)) \quad \text{εάν-ν} \quad \lim_{n\to\infty} \frac{T(n)}{f(n)} = +\infty$$

$$T(n) \in \Theta(f(n)) \quad \text{εάν-ν} \quad \lim_{n\to\infty} \frac{T(n)}{f(n)} = c, \quad c \in \mathcal{R}^+ \text{ και } \quad c \neq \infty$$

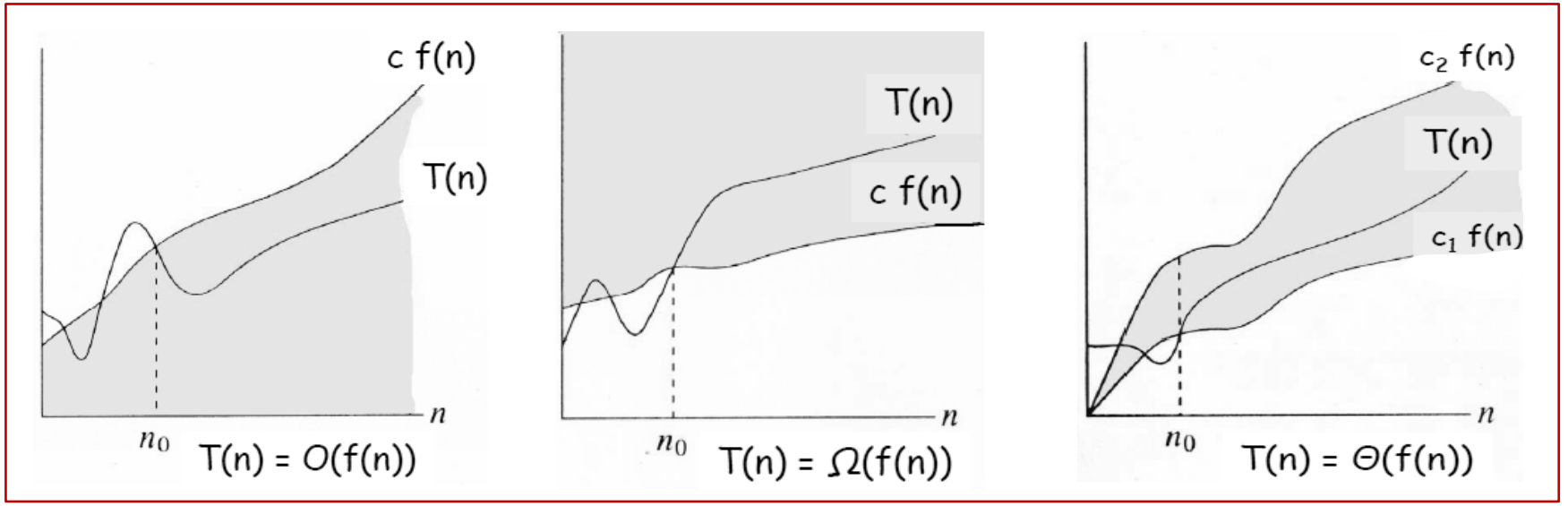
Βασικές Ιδιότητες των O, Ω και Θ

- Έστω f, g : N→R*, τότε ισχύουν οι παρακάτω προτάσεις:
  - $g(n) \in \mathcal{O}(f(n)) \quad \wedge \quad f(n) \in \mathcal{O}(h(n)) \quad \Longrightarrow \quad g(n) \in \mathcal{O}(h(n))$
  - $g(n) \in \mathcal{O}(f(n)) \quad \Longleftrightarrow \quad f(n) \in \Omega(g(n))$
  - $g(n) \in \Theta(f) \quad \Longrightarrow \quad f(n) \in \Theta(g(n))$



Βασικές Ιδιότητες των O, Ω και Θ

- Σχηματική Παράσταση



Βασικές Ιδιότητες των Ο, Ω και Θ

- Εάν f, g : N→R^*, τότε ισχύουν οι παρακάτω προτάσεις:

✓ $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathcal{R}^+ \quad \Longrightarrow \quad f(n) \in \Theta(g(n))$

✓ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \Longrightarrow \quad f(n) \in \mathcal{O}((g(n)) \quad \wedge \quad f(n) \notin \Theta(g(n))$

✓ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \quad \Longrightarrow \quad f(n) \in \Omega((g(n)) \quad \wedge \quad f(n) \notin \Theta(g(n))$

Βασικές Ιδιότητες των O, Ω και Θ

- Εάν f, g : N⟶R^*, ισχύει η σχέση:

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max[f(n), g(n)])$$

Η παραπάνω ιδιότητα χρησιμοποιείται συχνά για την απλοποίηση ασυμπτωτικών υπολογισμών.

Παράδειγμα

$$\begin{aligned} n^3 + 3n^2 + n + 5 \in \mathcal{O}(n^3 + 3n^2 + n + 5) \\ = \mathcal{O}(n^3 + (3n^2 + n + 5)) \\ = \mathcal{O}(max[n^3, (3n^2 + n + 5)]) \\ = \mathcal{O}(n^3) \end{aligned}$$

- **Τώρα Γνωρίζω !!!**

n	Cray-1 Fortran[a] $3n^3$ νανο-δευτερόλεπτα	TRS-80 Basic[b] $19.500.000n$ νανο-δευτερόλεπτα
10	3.0 μικρο-δευτερόλεπτα	0.2 δευτερόλεπτα
100	3.0 χιλιοστά-δευτερολέπτου	2.0 δευτερόλεπτα
1000	3.0 δευτερόλεπτα	20.0 δευτερόλεπτα
2500	50.0 δευτερόλεπτα	50.0 δευτερόλεπτα ← $\forall n \geq n_0 = 2500$
10.0000	49.0 λεπτά	3.2 λεπτά
1.000.000	95.0 χρόνια	5.4 ώρες

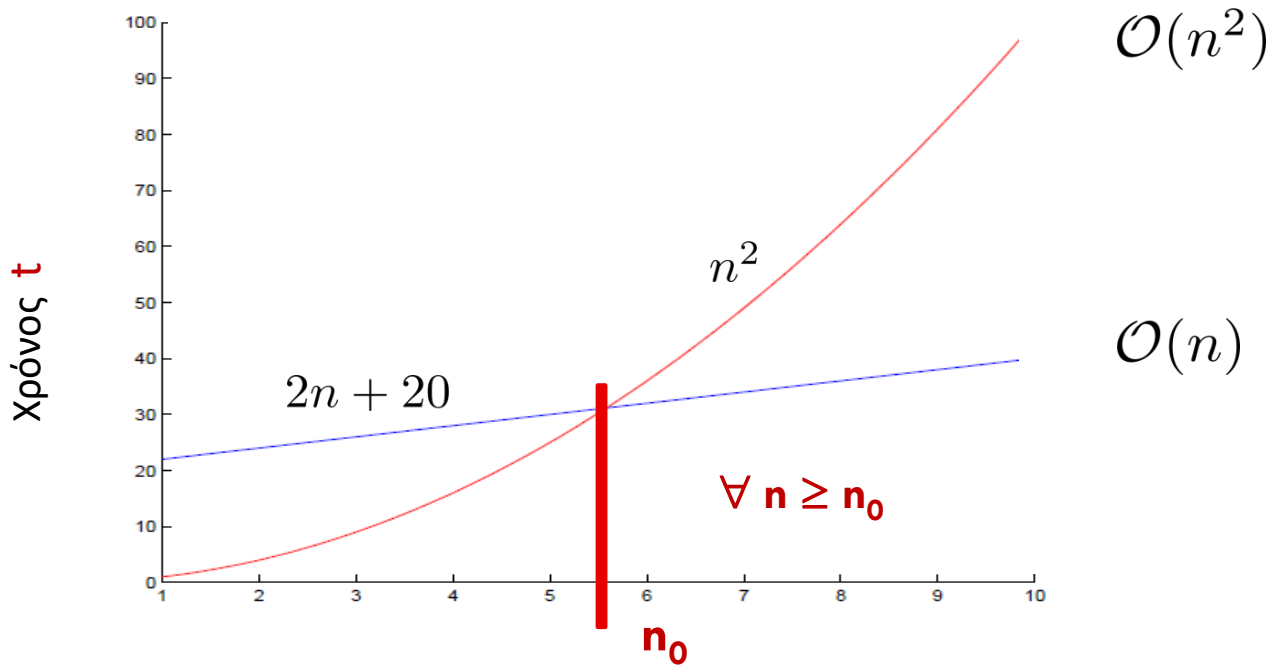
[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

Τώρα Γνωρίζω !!!

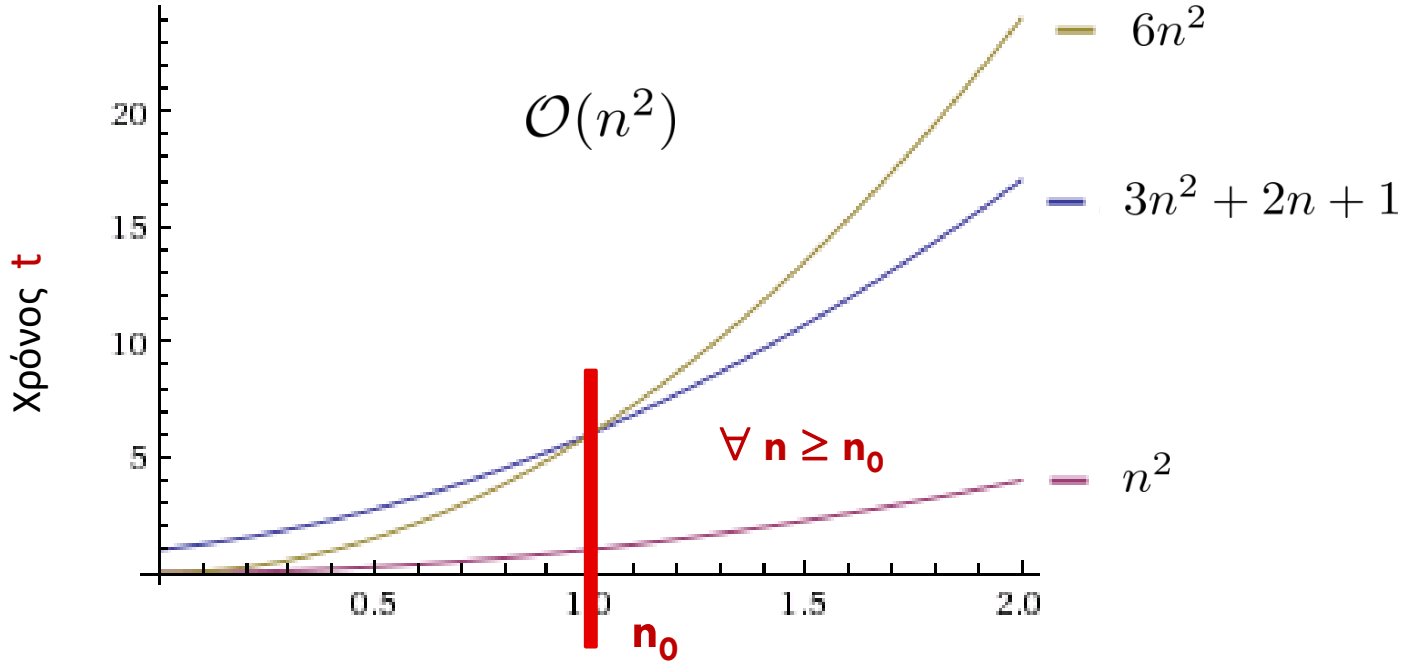
n	Cray-1 Fortran[a] **$3n^3$** νανο-δευτερόλεπτα	TRS-80 Basic[b] **$19.500.000n$** νανο-δευτερόλεπτα
	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$
	Ασυμπτωτική Πολυπλοκότητα Αλγόριθμου Α	Ασυμπτωτική Πολυπλοκότητα Αλγόριθμου Β
2500	**50.0** δευτερόλεπτα	**50.0** δευτερόλεπτα ← $\forall\ n \geq n_0 = 2500$
10.0000	**49.0** λεπτά	**3.2** λεπτά
1.000.000	**95.0** χρόνια	**5.4** ώρες

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

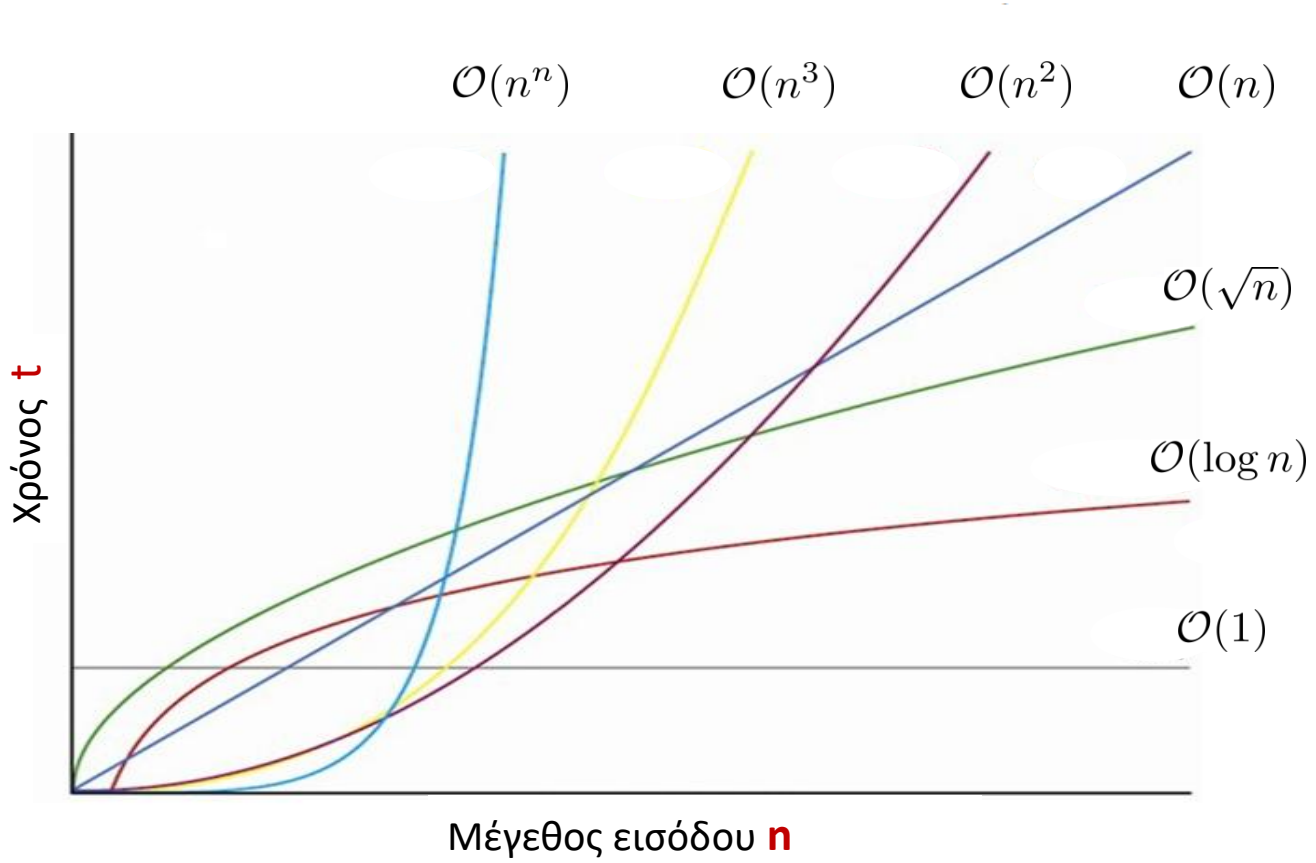
- **Τώρα Γνωρίζω !!!**



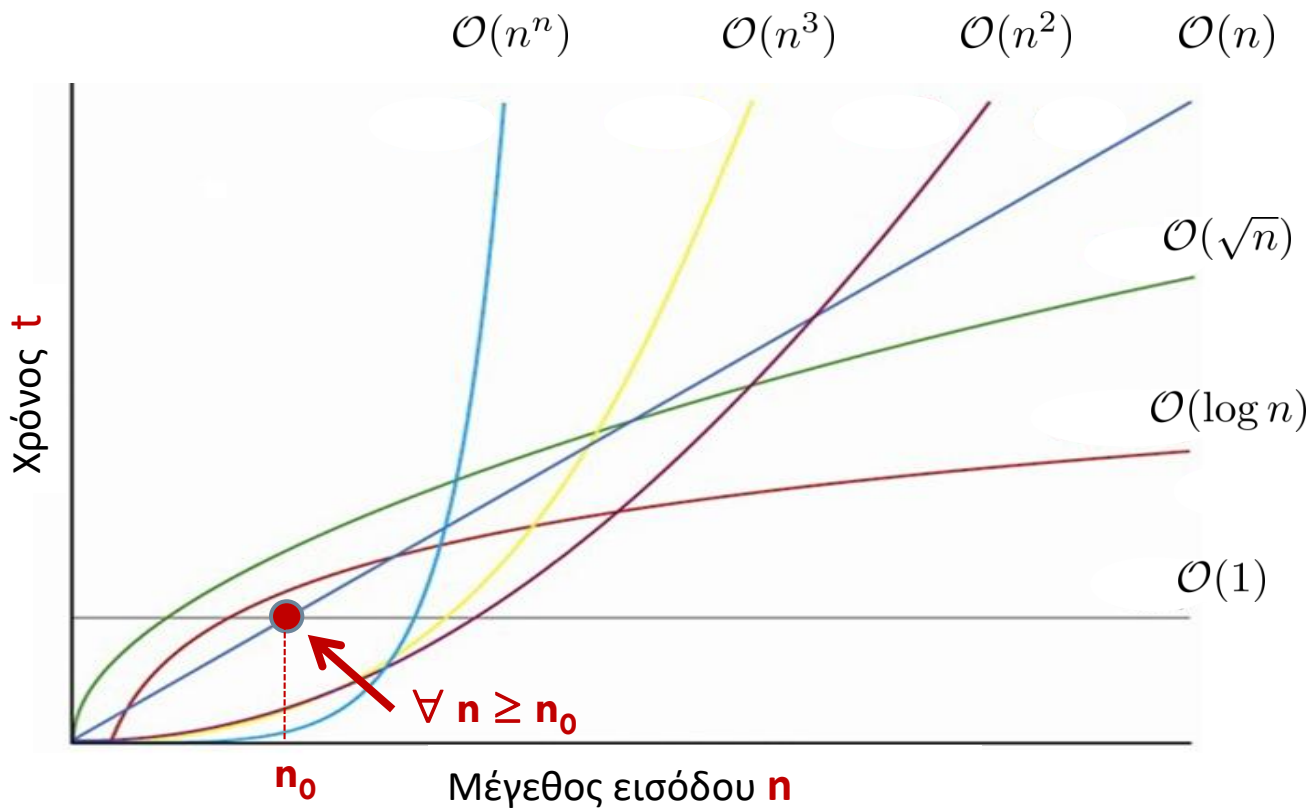
- **Τώρα Γνωρίζω !!!**



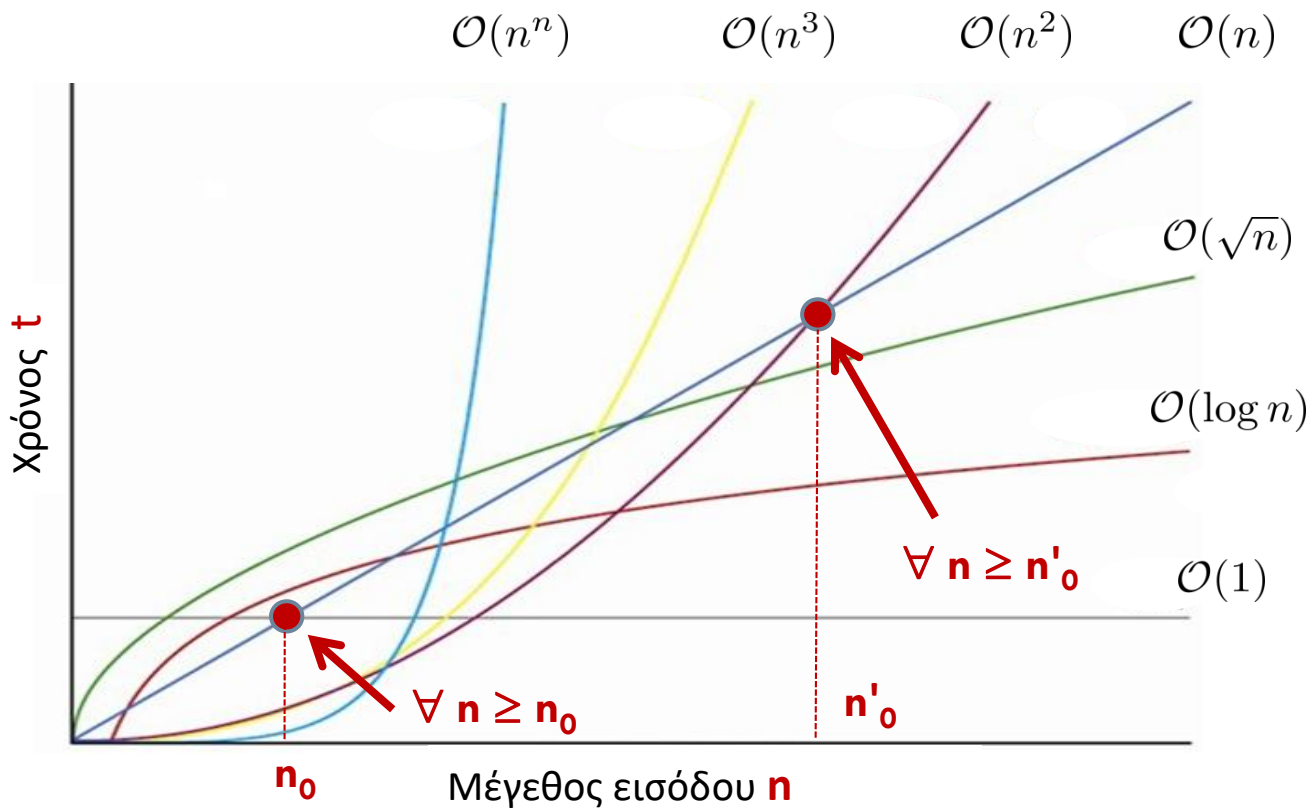
Σύγκριση Συναρτήσεων Πολυπλοκότητας !!!



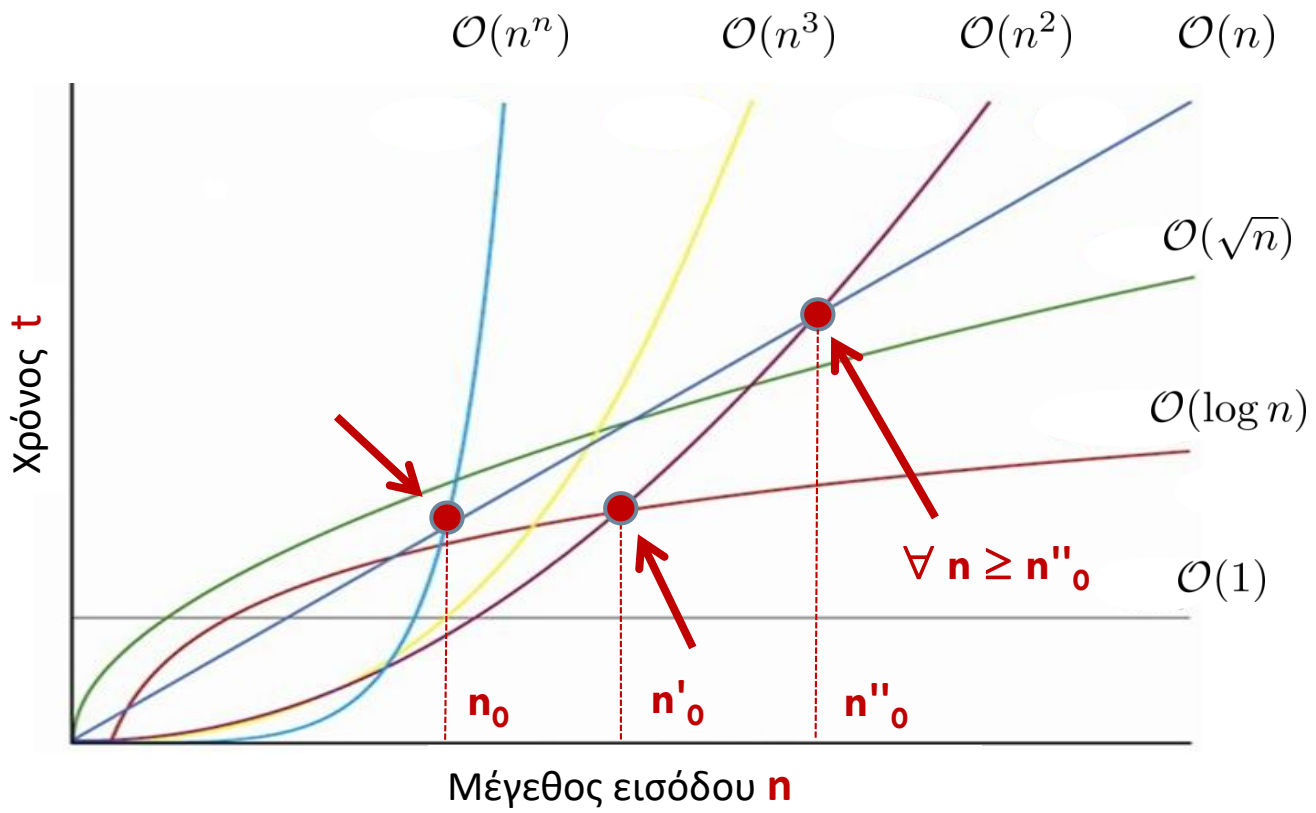
Σύγκριση Συναρτήσεων Πολυπλοκότητας !!!



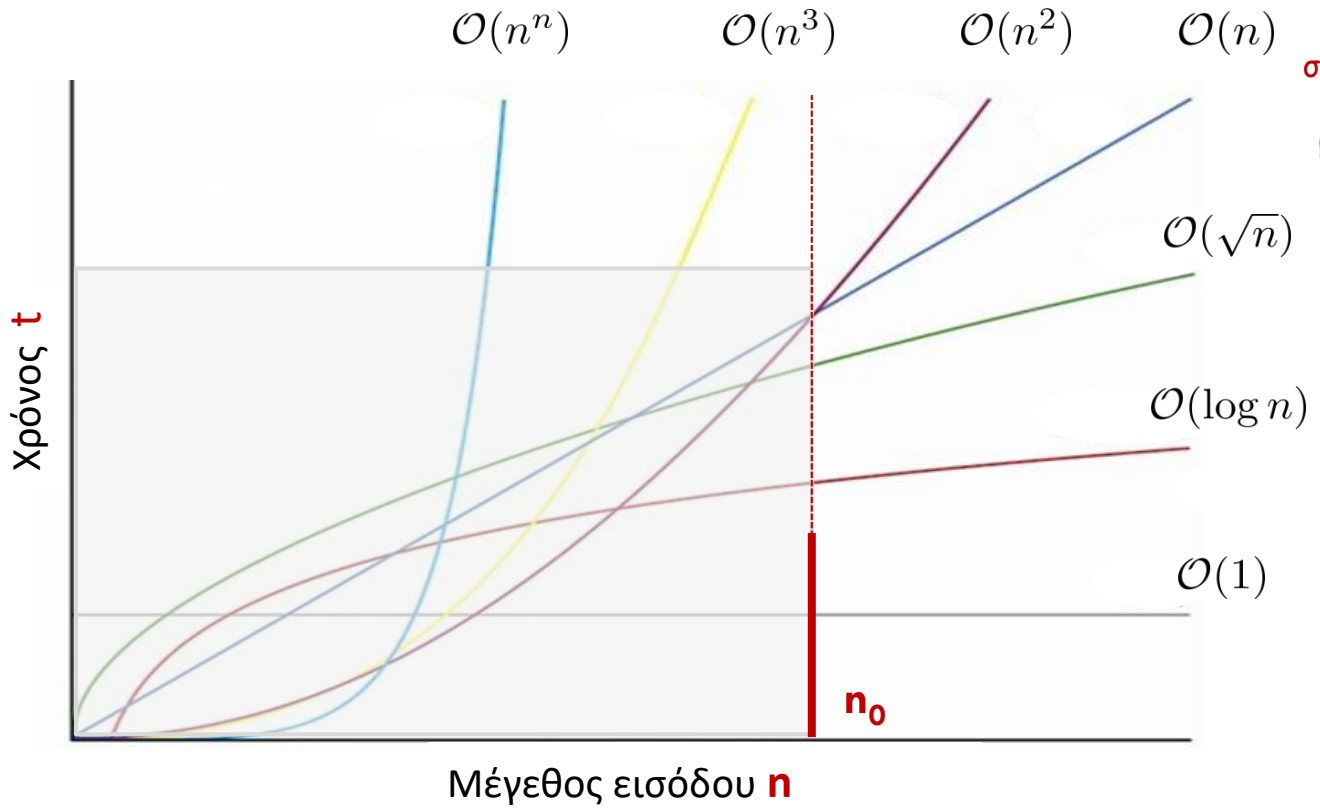
Σύγκριση Συναρτήσεων Πολυπλοκότητας !!!



Σύγκριση Συναρτήσεων Πολυπλοκότητας !!!



Σύγκριση Συναρτήσεων Πολυπλοκότητας !!!



Ενδιαφερόμαστε για την συμπεριφορά των αλγορίθμων μας για μεγάλες τιμές n του μεγέθους της εισόδου τους !!

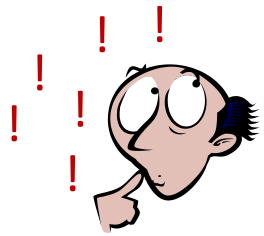
Όλοι οι αλγόριθμοι για μικρές τιμές του n είναι αποτελεσματικοί !!!

Ασυμπτωτική Πολυπλοκότητα

- **Συναρτήσεις Πολυπλοκότητας !!!**

Notation	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(\log(\log(n))$	Double logarithmic (iterative logarithmic)
$o(n)$	Sublinear
$O(n)$	Linear
$O(n \log(n))$	Loglinear, Linearithmic, Quasilinear or Supralinear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^c)$	Polynomial (different class for each $c > 1$)
$O(c^n)$	Exponential (different class for each $c > 1$)
$O(n!)$	Factorial
$O(n^n)$	- (Yuck!)

Απλοί Κανόνες για Απλοποίηση Συναρτήσεων !!!



- Οι πολλαπλασιαστικές σταθερές παραλείπονται:
  π.χ., το $14n^2$ γίνεται n^2, το 19.000.000 n γίνεται n.
- Το n^a επικρατεί έναντι του n^b εάν $a > b$: το n^3 επικρατεί έναντι του n^2.
- Ένας εκθετικός όρος επικρατεί έναντι ενός πολυωνυμικού:
  το 3^n επικρατεί έναντι του n^{100} (επικρατεί ακόμη και έναντι του 2^n).
- Ένας πολυωνυμικός όρος επικρατεί έναντι ενός λογαριθμικού:
  το n επικρατεί έναντι του $(\log n)^3 = \log^3 n$, το n^2 επικρατεί έναντι του $n \log n$.
- Και φυσικά:

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max[f(n), g(n)])$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = c \ (c \in \mathcal{R}^*, c \neq \infty) \implies T(n) \in \Theta(f(n)) \ \ or \ \ T(n) \in \mathcal{O}(f(n))$$

Αλγόριθμος Linear-Search

Algorithm **Linear-Search**

```
begin
    index ← 1
    while S[index] ≠ x && index ≤ n do
          index ← index + 1
    end
    if index > n then index ← 0
    return index
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: **W(n) = n**

Μέση-Περίπτωση: **A(n) = (n+1)/2**

Ασυμπτωτική Πολυπλοκότητα

W(n) = *O*(n)

A(n) = *O*(n)

$$\mathcal{O}(f(n)+g(n)) = \mathcal{O}(max[f(n),g(n)])$$

$$\lim_{n\to\infty}\frac{T(n)}{f(n)} = c\ (c\in\mathcal{R}^*, c\neq\infty) \implies T(n)\in\Theta(f(n))\ \ or\ \ T(n)\in\mathcal{O}(f(n))$$

Αλγόριθμος `Find_Max-I`

Algorithm **Find_Max-I**

```
begin
   max ← S[1]; posmax ← 1
     for i ← 2 to n do
         if S[i]>max then
               max ← S[i]; posmax ← I
          end
     end
   return max, posmax
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: **W(n) = n-1**

Μέση-Περίπτωση: **A(n) = n-1**

Ασυμπτωτική Πολυπλοκότητα

W(n) = *O*(n)

A(n) = *O*(n)

$$\mathcal{O}(f(n)+g(n)) = \mathcal{O}(max[f(n),g(n)])$$

$$\lim_{n\to\infty}\frac{T(n)}{f(n)} = c\ (c\in\mathcal{R}^*, c\neq\infty) \implies T(n)\in\Theta(f(n))\ \ or\ \ T(n)\in\mathcal{O}(f(n))$$

Αλγόριθμος `Find_Max-II`

Algorithm **Find_Max-II**

```
begin
      for i ← 1 to logn do
          k ← 1
          for j ← 1 to n/2^(i-1) step 2 do
                if S[j] < S[j+1]
                      then S[k] ← S[j+1]
                      else S[k] ← S[j]
                k ← k+1;
          end
      end
      return max ← S[1]
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: **W(n) = n-1**

Μέση-Περίπτωση: **A(n) = n-1**

Ασυμπτωτική Πολυπλοκότητα

W(n) = *O*(n)

A(n) = *O*(n)

$$\mathcal{O}(f(n)+g(n)) = \mathcal{O}(max[f(n),g(n)])$$

$$\lim_{n\to\infty}\frac{T(n)}{f(n)} = c\ (c\in\mathcal{R}^*, c\neq\infty) \implies T(n)\in\Theta(f(n))\ \ or\ \ T(n)\in\mathcal{O}(f(n))$$

Αλγόριθμος `Find_Min-Max`

Algorithm **Find_Min-Max**

```
begin
      min ← max ← S[1]
      for i ← 2 to n do
          if S[i] < min then
             min ← S[i]
          else
              if S[i] > max then
                  max ← S[i]
           end
      end
      return min, max
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: $W(n) = 2n-2$

Μέση-Περίπτωση: $A(n) = 3/2\,(n-1)$

Ασυμπτωτική Πολυπλοκότητα

$W(n) = O(n)$

$A(n) = O(n)$

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max[f(n), g(n)])$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = c \ (c \in \mathcal{R}^*, c \neq \infty) \implies T(n) \in \Theta(f(n)) \ \ or \ \ T(n) \in \mathcal{O}(f(n))$$

Αλγόριθμος Bubble-sort

Algorithm **Bubble-sort**

```
begin
      for i ← 1 to n-1 do
         for j ← n downto i+1 do
           if S[j] < S[j-1] then
              Swap(S[j], S[j-1])
           end
      end
      return S
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: **W(n)** = **n(n-1)/2**

Μέση-Περίπτωση: **A(n)** = n(**n-1**)**/2**

Ασυμπτωτική Πολυπλοκότητα

W(n) = $O(n^2)$

A(n) = $O(n^2)$

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max[f(n), g(n)])$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = c \ (c \in \mathcal{R}^*, c \neq \infty) \implies T(n) \in \Theta(f(n)) \ \ or \ \ T(n) \in \mathcal{O}(f(n))$$

Αλγόριθμος Insertion-sort

Algorithm **Insertion-sort**

```
begin
      for i ← 2 to n do
         x ← S[i]; j ← i-1
         while j >= 1 and S[j] > x do
               S[j+1] ← S[j]
               j ← j-1
         end
         S[j+1] ← x
      end
      return S
end
```

Πολυπλοκότητα

Χείριστη-Περίπτωση: $\mathbf{W(n) = n(n-1)/2}$

Μέση-Περίπτωση: $\mathbf{A(n) = n^2/4}$

Ασυμπτωτική Πολυπλοκότητα

$\mathbf{W(n)} = O(n^2)$

$\mathbf{A(n)} = O(n^2)$

$$\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max[f(n), g(n)])$$

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = c \ (c \in \mathcal{R}^*, c \neq \infty) \implies T(n) \in \Theta(f(n)) \ \ or \ \ T(n) \in \mathcal{O}(f(n))$$

Algorithm **Min-Max(S)**

```
S: array[1..n] of integer
begin
    if |S| = 2 then
        return(Min(S), Max(S))
    else
        Διαίρεσε το S σε δύο περίπου
        ίσα υποσύνολα S1 και S2
        (min1, max1) ← Min-Max(S1)
        (min2, max2) ← Min-Max(S2)
        return Min(min1,min2), Max(max1,max2)
    end
end
```

Αναδρομικός Αλγόριθμος Min-Max

Algorithm **Min-Max(S)**

```
S: array[1..n] of integer
begin
   if |S| = 2 then
        return(Min(S), Max(S))
   else
        Διαίρεσε το S σε δύο περίπου
        ίσα υποσύνολα S1 και S2
        (min1, max1) ← Min-Max(S1)
        (min2, max2) ← Min-Max(S2)
        return Min(min1,min2), Max(max1,max2)
   end
end
```

Πολυπλοκότητα

$W(n) = ?$ $A(n) = ?$

Βασική Πράξη

$|S|=2$ και οι βασικές πράξεις των δύο Min-Max με είσοδο $\sim n/2$

$W(n) = 1$ εάν $n = 2$

$W(n) = 2W(n/2)+2$ εάν $n > 2$

Αναδρομικός Αλγόριθμος `Min-Max`

Algorithm **Min-Max(S)**

Βασική Πράξη : |S|=2 & οι βασικές πράξεις του Min-Max είσοδο ~ n/2

Πολυπλοκότητα :

$$W(n) = \begin{cases} 1 & n = 2 \\ 2W(n/2) + 2 & n > 2 \end{cases}$$

Αναδρομική Σχέση

Ανάλυση Χείριστης – Περίπτωσης : W(n) = **?**

Αναδρομικός Αλγόριθμος `Binary-Search`

Algorithm **Binary-Search(A, x, low, high)**

```
A: array[1..n] of integer
begin
    if (high < low) return(not-found)
    mid ← (low + high) / 2
    if (A[mid] > x) then
        return BinarySearch(A, x, low, mid-1)
    else if (A[mid] < x) then
        return BinarySearch(A, x, mid+1, high)
    else
        return mid
end
```

Πολυπλοκότητα

$W(n) = ?$ $A(n) = ?$

Βασική Πράξη

`A[mid] > value` και οι βασικές πράξεις του **`BinarySearch`** με είσοδο $\sim n/2$

$W(n) = 1$ εάν $n = 1$

$W(n) = 1 + W(n/2)$ εάν $n > 1$

Αναδρομικός Αλγόριθμος `Binary-Search`

Algorithm **Binary-Search(A, x, low, high)**

Βασική Πράξη : **`A[mid] > x`** & οι ΒΠ του **`BinarySearch`** με είσοδο ~ n/2

Πολυπλοκότητα :

$$W(n) = \begin{cases} 1 & n = 1 \\ 1 + W(\lfloor n/2 \rfloor) & n > 1 \end{cases}$$

Αναδρομική Σχέση

Ανάλυση Χείριστης – Περίπτωσης : W(n) = **?**

Αναδρομικός Αλγόριθμος `Alpha`

Algorithm **Alpha**

Έστω ότι η πολυπλοκότητα του αλγόριθμου Alpha δίδεται από την σχέση:

Πολυπλοκότητα :

$$W(n) = \begin{cases} 1 & n = 2 \\ 2W(n/2) + c\,n & n > 2 \end{cases}$$

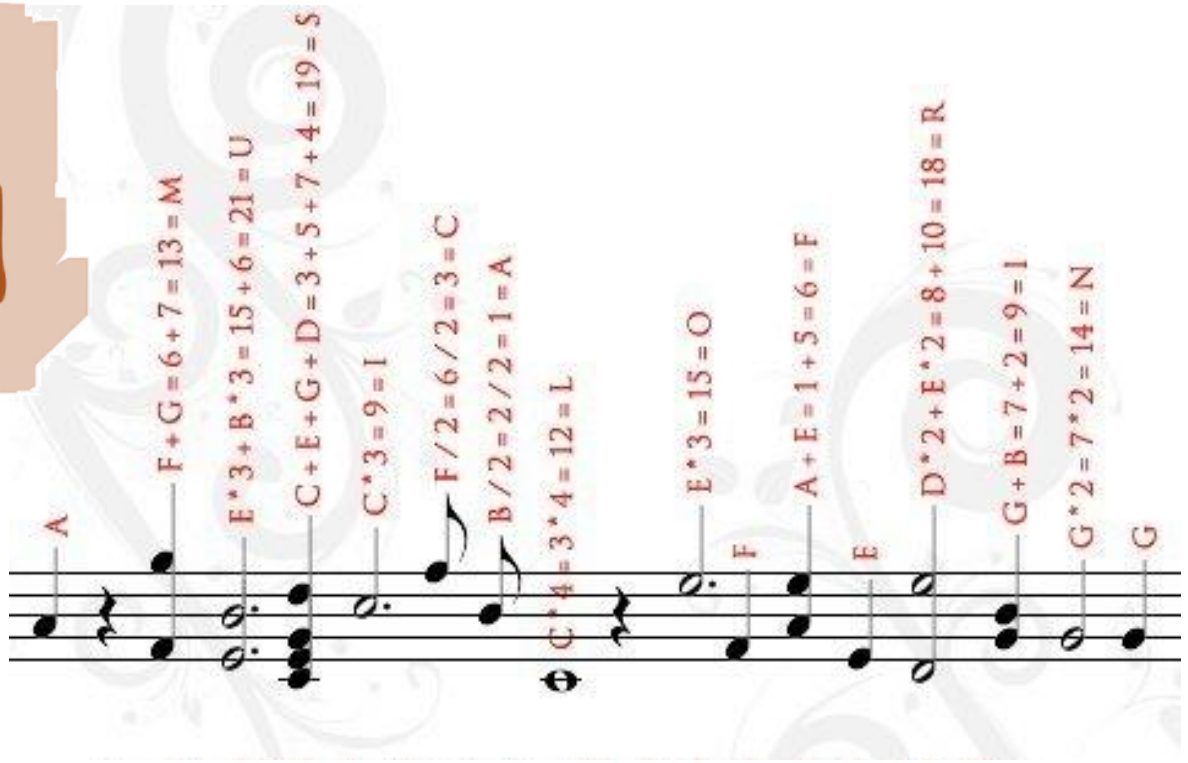
Αναδρομική Σχέση

Ανάλυση Χείριστης – Περίπτωσης : W(n) = **?**

Επίλυση Αναδρομικών Σχέσεων

- Με Εικασία και Απόδειξή της
- Με χρήση της Χαρακτηριστικής Εξίσωσης
- Με Επαναληπτική Αντικατάσταση
- Με χρήση της Κύριας Μεθόδου (Master Method or Theorem)

Master Method



A powerful tool for solving recurrences.

Κύρια Μέθοδος (Master Method)

- Η Κύρια μέθοδος είναι ένα σημαντικό εργαλείο για την επίλυση μιας μεγάλης κατηγορίας αναδρομικών σχέσεων.

- Έστω $T(n)$: $Z^+ \rightarrow R^*$ μια αναδρομική σχέση:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

όπου

- $a \geq 1$ και $b > 1$ σταθερές,
- n/b είναι είτε $\lfloor n/b \rfloor$ ή $\lceil n/b \rceil$, και
- $f(n)$ μια θετική συνάρτηση ορισμένη πάνω στο σύνολο των θετικών ακεραίων, δηλ. $f(n)$: $Z^+ \rightarrow R^*$.

Κύρια Μέθοδος

- Εάν ο χρόνος εκτέλεσης $T(n)$ ενός αλγόριθμου Α για την επίλυση ενός προβλήματος Π μεγέθους Θ(n) δίδεται από την αναδρομική σχέση

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

τότε:

$T(n)$ είναι ο χρόνος για την επίλυση a υποπροβλημάτων μεγέθους n/b συν $f(n)$ η οποία είναι το άθροισμα:

- ✓ του χρόνου για τη διαμέριση του αρχικού Π σε a υπο-προβλήματα, και
- ✓ του χρόνου για τον συνδυασμό των λύσεων των υπο-προβλημάτων για να πάρουμε την τελική λύση του αρχικού προβλήματος Π.

Κύρια Μέθοδος

- Έστω $f(n) = O(n^c)$, τότε:

$$T(n) = a \cdot T(\frac{n}{b}) + O(n^c)$$

και η λύση της είναι:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{εάν} \quad c < \log_b a \quad (1) \\ O(n^c \log n) & \text{εάν} \quad c = \log_b a \quad (2) \\ O(n^c) & \text{εάν} \quad c > \log_b a \quad (3) \end{cases}$$

Επίλυση Αναδρομικών Σχέσεων - Παραδείγματα

Κύρια Μέθοδος

- **Παράδειγμα 1:** $T(n) = T(\frac{n}{2}) + 1$

$$a = 1 \quad b = 2 \quad \Longrightarrow \quad \log_2 1 = 0$$

$$c = 0 \quad \{1 = O(1) = O(n^0)\}$$

Ισχύει : $c = \log_b a$ (2)

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{εάν} \quad c < \log_b a \quad (1) \\ O(n^c \log n) & \text{εάν} \quad c = \log_b a \quad (2) \\ O(n^c) & \text{εάν} \quad c > \log_b a \quad (3) \end{cases}$$

Λύση : $T(n) = O(n^c \log n) = O(\log n)$ ✓

Κύρια Μέθοδος

- **Παράδειγμα 2:** $T(n) = 2\,T(\frac{n}{2}) + n$

$$a = 2 \quad b = 2 \quad \Longrightarrow \quad \log_2 2 = 1$$

$$c = 1$$

Ισχύει : $c = \log_b a$ (2)

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{εάν} \quad c < \log_b a \quad (1) \\ O(n^c \log n) & \text{εάν} \quad c = \log_b a \quad (2) \\ O(n^c) & \text{εάν} \quad c > \log_b a \quad (3) \end{cases}$$

Λύση : $T(n) = O(n^c \log n) = O(n \log n)$ ✓

Κύρια Μέθοδος

- **Παράδειγμα 3:** $T(n) = T(\frac{n}{4}) + n^{1/2}$

$$a = 1 \quad b = 4 \quad \Longrightarrow \quad \log_4 1 = 0$$

$$c = 1/2$$

Ισχύει : $c > \log_b a$ (3)

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{εάν} \quad c < \log_b a \quad (1) \\ O(n^c \log n) & \text{εάν} \quad c = \log_b a \quad (2) \\ O(n^c) & \text{εάν} \quad c > \log_b a \quad (3) \end{cases}$$

Λύση : $T(n) = O(n^c) = O(n^{1/2})$ ✓

Κύρια Μέθοδος

- **Παράδειγμα 4:** $T(n) = 9\ T(\frac{n}{3}) + n$

$$a = 9 \quad b = 3 \quad \Longrightarrow \quad \log_3 9 = 2$$

$$c = 1$$

Ισχύει : $c < \log_b a$ (1)

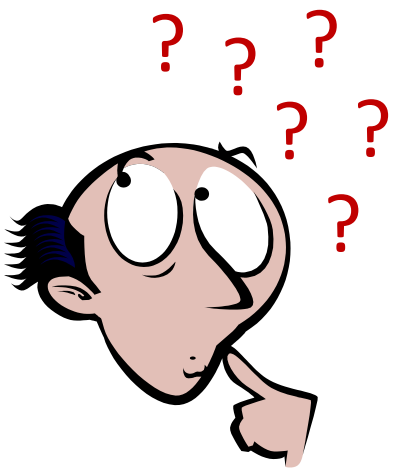
$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{εάν} \quad c < \log_b a \quad (1) \\ O(n^c \log n) & \text{εάν} \quad c = \log_b a \quad (2) \\ O(n^c) & \text{εάν} \quad c > \log_b a \quad (3) \end{cases}$$

Λύση : $T(n) = O(n^{\log_b a}) = O(n^2)$ ✓

Επίλυση Αναδρομικών Σχέσεων - Παραδείγματα

Κύρια Μέθοδος

- **Παράδειγμα 5:** $T(n) = 3T(\frac{n}{4}) + n \cdot \log n$



$$T(n) = aT(n/b) + O(n^c)$$

Αλγοριθμικές Τεχνικές Σχεδίασης !!!

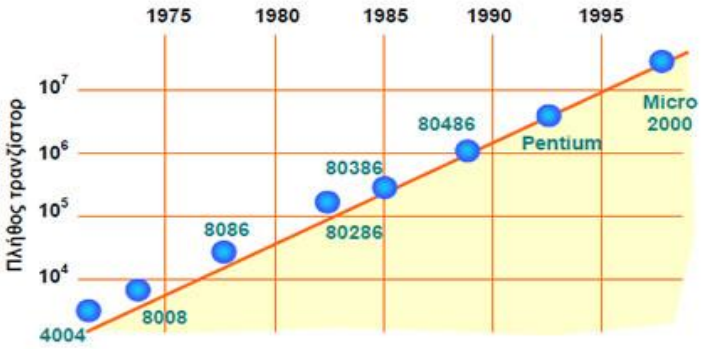
- Θεμελιώδη Εργαλεία

  - ✓ Αναδρομή (recursion) ←
  - ✓ Διαίρει και κυρίευε (divide and conquer)
  - ✓ Απληστία (greedy algorithms)
  - ✓ Δυναμικός προγραμματισμός (dynamic programming)
  - ✓ Γραμμικός Προγραμματισμός (linear programming)
  - ✓ Αναγωγές (reduction)

Εκθετικός Αλγόριθμος vs Τεχνολογία

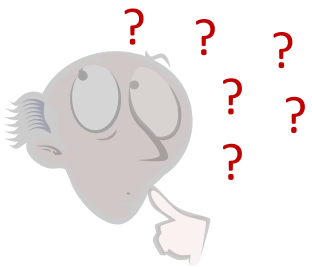
Ιστορία των Sissa και Moore

$$2^{64} - 1$$



Εκθετικός Αλγόριθμος vs Πολυωνυμικός Αλγόριθμος

Πρόβλημα Π

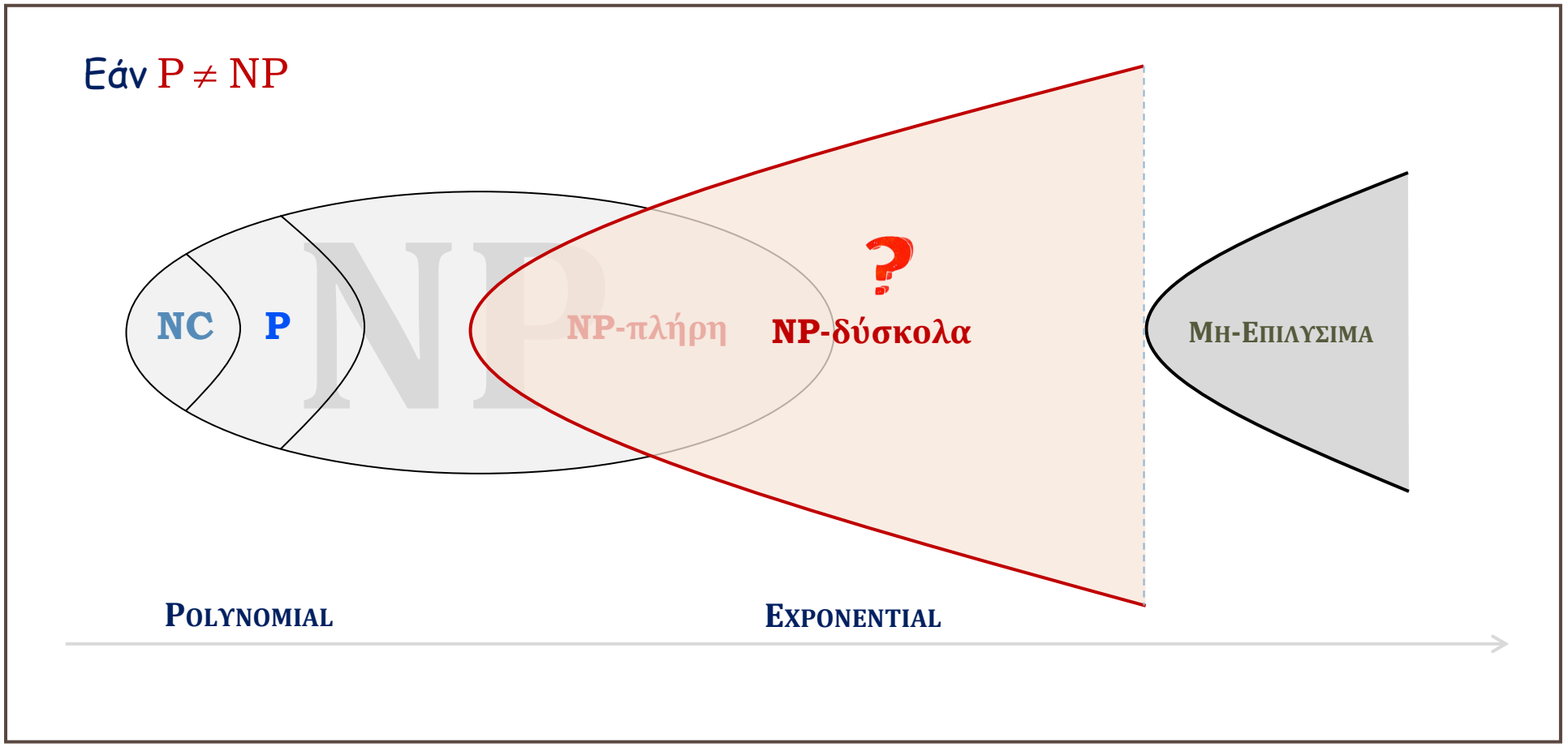


Μέγεθος Εισόδου n

Πολυπλοκότητα Χρόνου

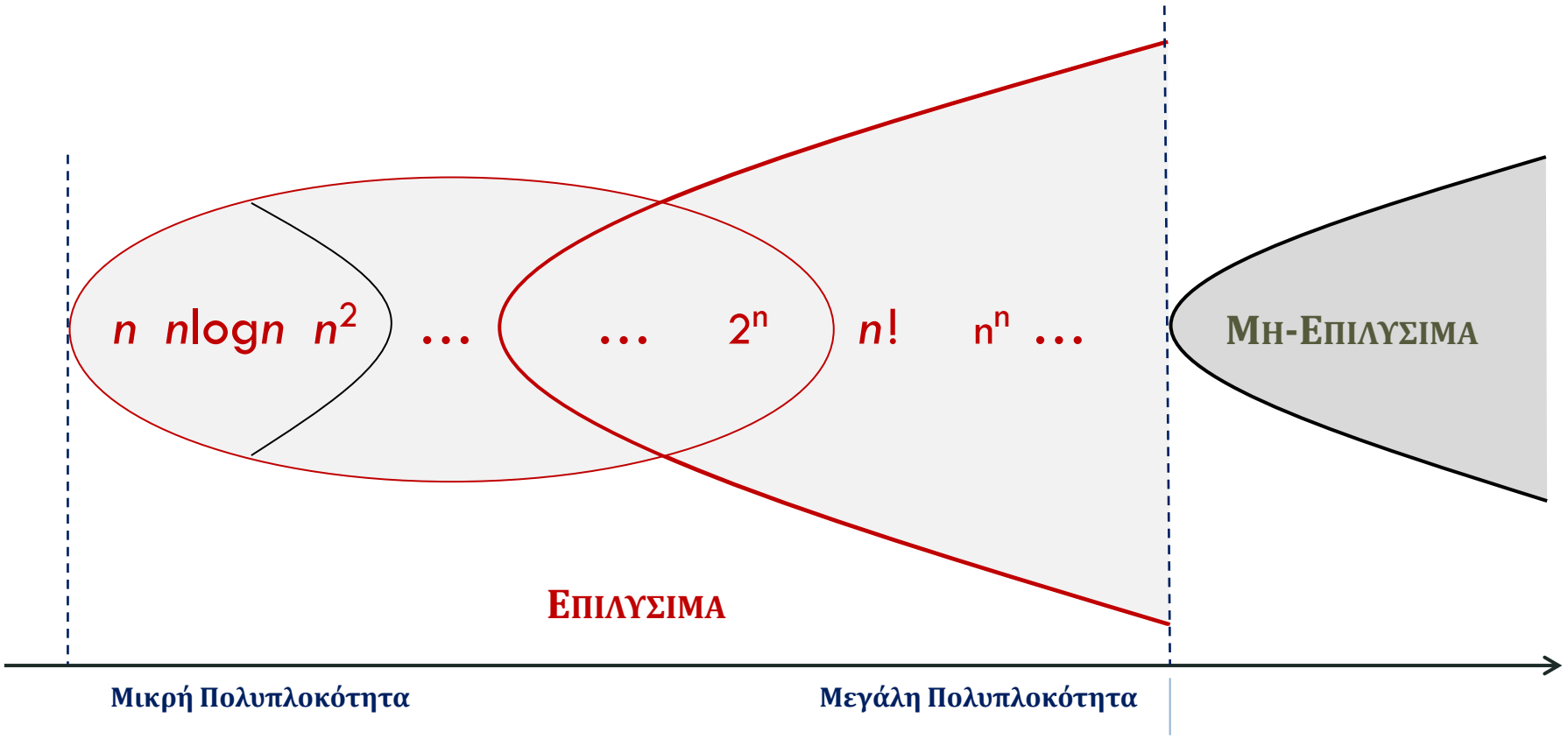
	2	8	32	64
$\log n$	1 δεύτερο	3 δεύτερα	5 δεύτερα	6 δεύτερα
n	2 δεύτερα	8 δεύτερα	32 δεύτερα	1.07 λεπτά
$n \log n$	2 δεύτερα	24 δεύτερα	2.67 λεπτά	6.4 λεπτά
n^2	4 δεύτερα	1.07 λεπτά	17.07 λεπτά	1.14 ώρες
...	...	...	...	...
2^n	4 δεύτερα	4.27 λεπτά	1.36 αιώνες	5.86×10^9 αιώνες
$n!$	2 δεύτερα	4.2 ώρες	8.34×10^{25} αιώνες	4.02×10^{79} αιώνες

Ο Χάρτης των Κλάσεων Μέχρι Σήμερα



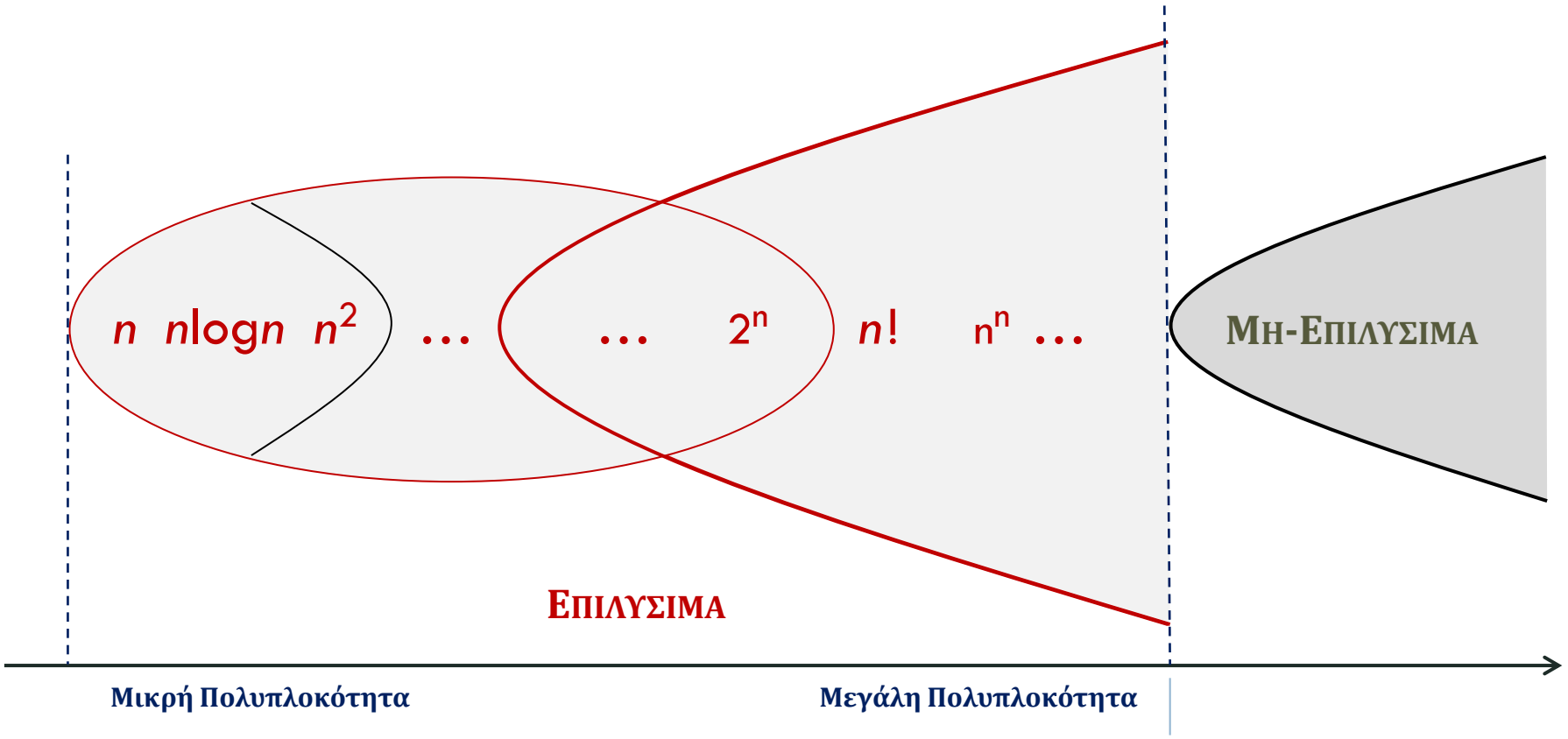
Κακός και Καλός Σχεδιασμός Αλγορίθμου

Εάν **P ≠ NP**



Κακός και Καλός Σχεδιασμός Αλγορίθμου

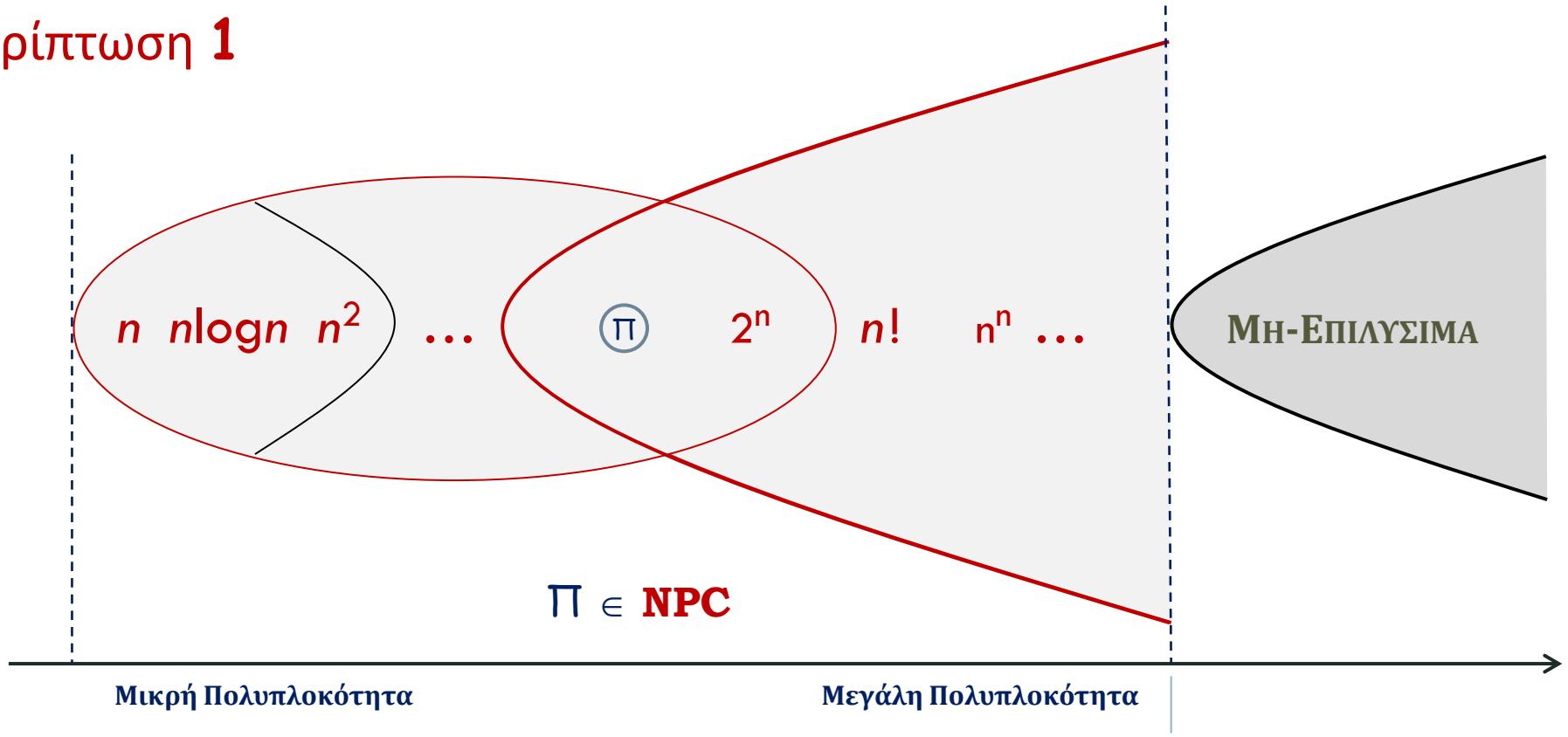
Εάν **P ≠ NP** Πρόβλημα **Π** ⇒ Αλγόριθμος ***A***



Κακός και Καλός Σχεδιασμός Αλγορίθμου

Εάν **P ≠ NP** Πρόβλημα **Π** ⇒ Αλγόριθμος ***A***

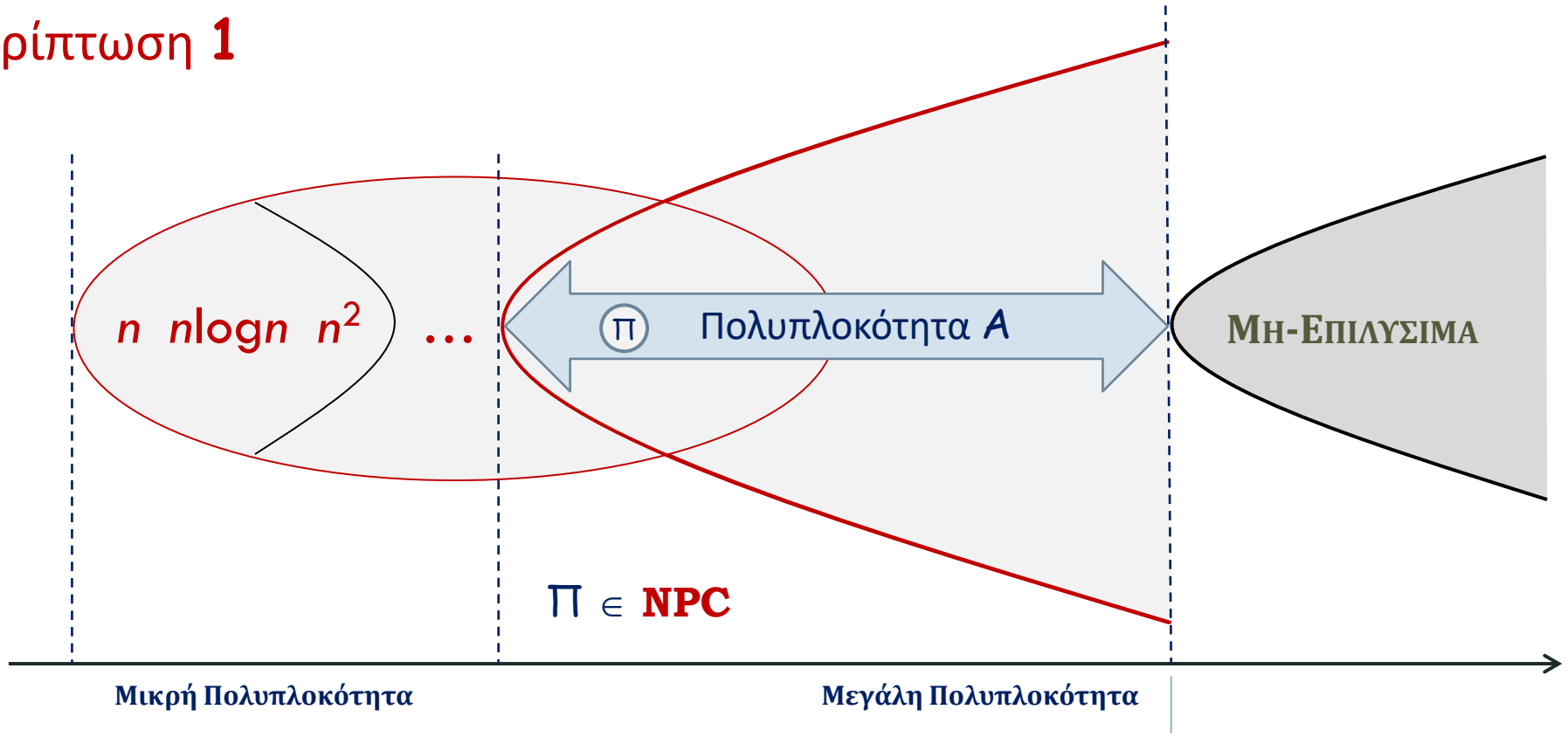
Περίπτωση **1**



Κακός και Καλός Σχεδιασμός Αλγορίθμου

Εάν **P ≠ NP** Πρόβλημα **Π** ⇒ Αλγόριθμος ***A***

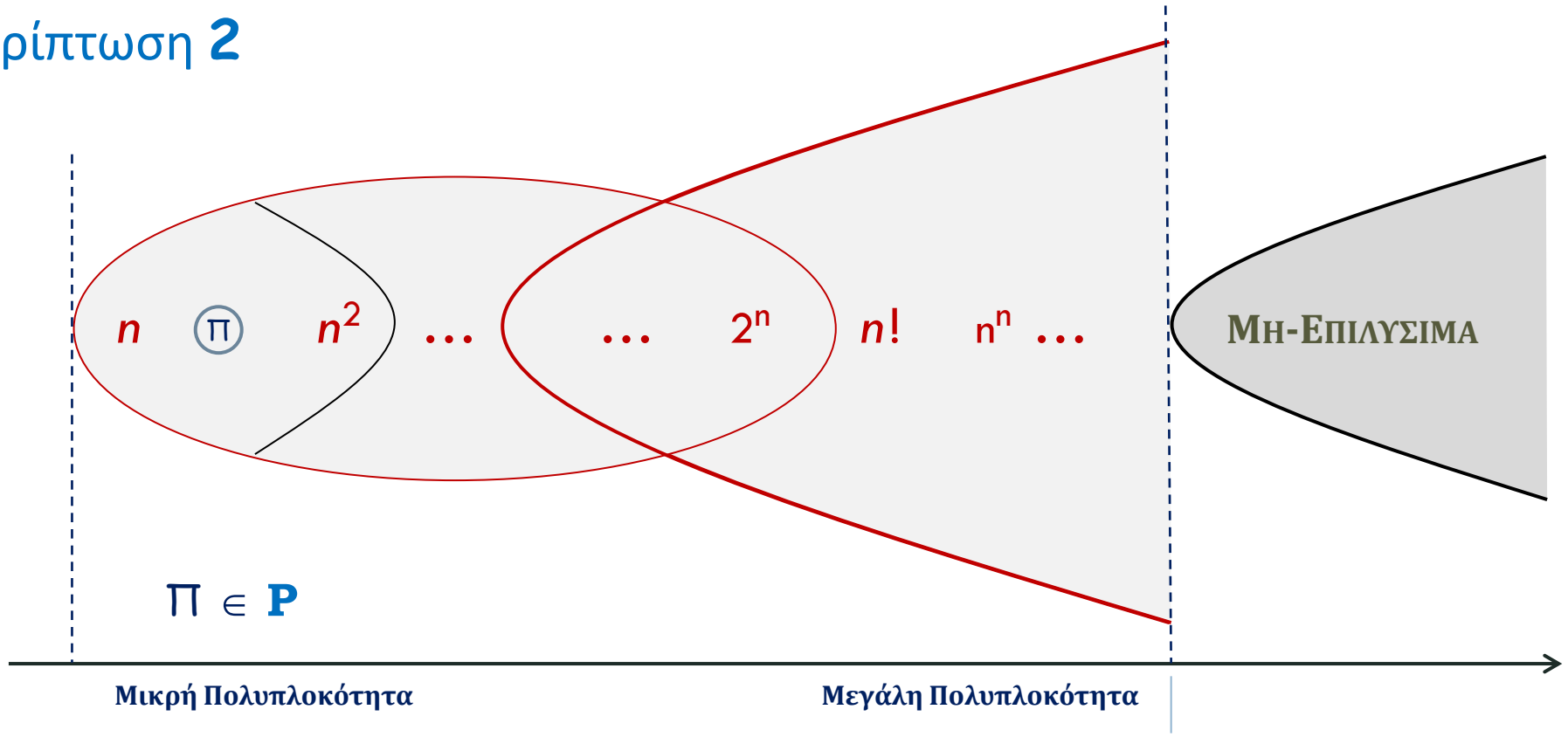
Περίπτωση **1**



Κακός και Καλός Σχεδιασμός Αλγορίθμου

Εάν **P ≠ NP** Πρόβλημα **Π** ⇒ Αλγόριθμος ***A***

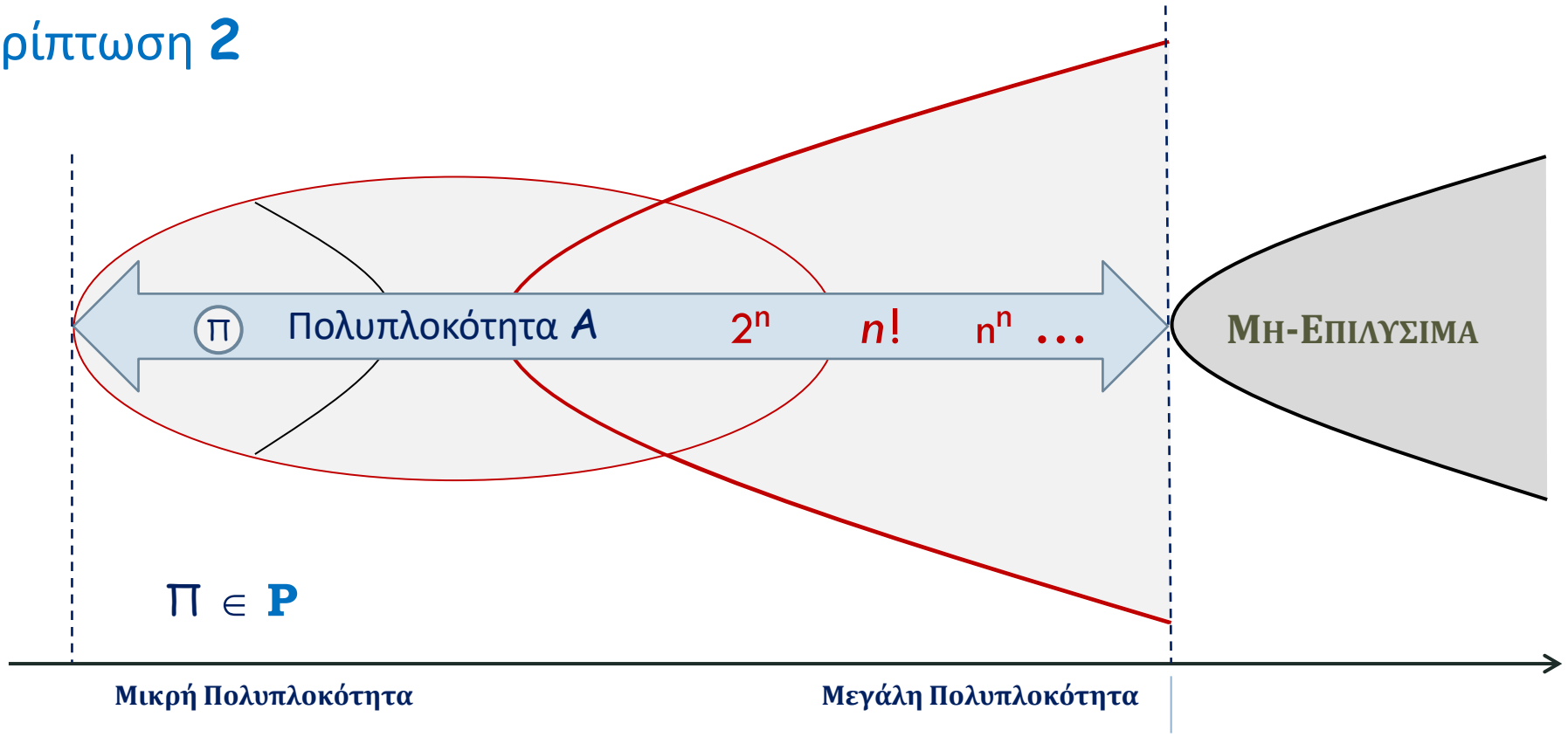
Περίπτωση **2**



Κακός και Καλός Σχεδιασμός Αλγορίθμου

Εάν **P ≠ NP** Πρόβλημα **Π** ⇒ Αλγόριθμος ***A***

Περίπτωση **2**



Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

$$F_n = \begin{cases} 0 & \text{εάν} \quad n = 0 \\ 1 & \text{εάν} \quad n = 1 \\ F_{n-1} + F_{n-2} & \text{εάν} \quad n > 1 \end{cases}$$

Συμβατικά Ορίζουμε: $F_0 = 0$

Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

$$F_n = \begin{cases} 0 & \text{εάν } n = 0 \\ 1 & \text{εάν } n = 1 \\ F_{n-1} + F_{n-2} & \text{εάν } n > 1 \end{cases}$$

Οι αριθμοί Fibonacci αυξάνουν *σχεδόν* το ίδιο γρήγορα με τις δυνάμεις του 2

Για παράδειγμα: $F_{30} > 1.000.000$

F_{100} αποτελείται ήδη από 21 ψηφία !!!

Γενικά, ισχύει: $F_n \approx 2^{0.694n}$

Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

$$F_n = \begin{cases} 0 & \text{εάν } n = 0 \\ 1 & \text{εάν } n = 1 \\ F_{n-1} + F_{n-2} & \text{εάν } n > 1 \end{cases}$$

Αλγόριθμος Fib1

Πολύ απλός αλγόριθμος !!!

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1) + Fib1(n-2)
```

Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

Για κάθε αλγόριθμο που σχεδιάζουμε θέτουμε πάντα 3 θεμελιώδη ερωτήματα !!!

1. Είναι σωστός;
2. Πόσο χρόνο απαιτεί (ως συνάρτηση του μεγέθους n της εισόδου);
3. Μπορούμε να τον βελτιώσουμε;

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

1. **Είναι σωστός;**

Ακαδημαϊκή ερώτηση!!!

Ο αλγόριθμος **Fib1** είναι ακριβώς ο ορισμός του Fibinacci για το F_n !!!

Καμία σοβαρή δική μας σκέψη !!!

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

2. Πόσο χρόνο απαιτεί;

Έστω T(n) το πλήθος των *υπολογιστικών βημάτων* (βασικών πράξεων) που απαιτούνται για τον υπολογισμό της Fib1(n).

$$T(n) \leq 2, \qquad n \leq 1$$

$$T(n) = T(n-1) + T(n-2) + 2, \qquad n > 1$$

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

2. Πόσο χρόνο απαιτεί;

$$T(n) = T(n-1) + T(n-2) + 3, \qquad n > 1$$

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

$$\Rightarrow \quad T(n) \geq F_n$$

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

2. **Πόσο χρόνο απαιτεί;**

$$T(n) \geq F_n$$ Πολύ άσχημα νέα !!!

Ο χρόνος εκτέλεσης του αλγόριθμου αυξάνει το ίδιο γρήγορα με τους αριθμούς Fibonacci!!!

$$F_n \approx 2^{0.694n} \quad \Rightarrow \quad T(n)$$ Εκθετικός ως προς το n

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

2. Πόσο χρόνο απαιτεί;

Τι σημαίνει αυτό?... Πόσο κακός είναι ο **Fib1** ?

$T(200) \geq F_{200} \geq 2^{138}$ υπολογιστικά βήματα

Earth Simulator της NEC με ταχύτητα 40 τρισεκατομμύρια βήματα/sec

Η συνάρτηση Fib1(200) απαιτεί 2^{92} sec !!!

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

2. Πόσο χρόνο απαιτεί;

Η συνάρτηση Fib1(200) απαιτεί 2^{92} sec στον Earth Simulator της NEC

Εάν ξεκινούσε σήμερα, αυτός ο υπολογισμός θα συνεχιζόταν και μετά την μετατροπή του ήλιου μας σε κόκκινο γίγαντα!!!!

Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

Η ταχύτητα των Η/Υ διπλασιάζεται περίπου κάθε 18 μήνες (Νόμος Moore)

Ίσως τότε ο αλγόριθμος Fib1 να εκτελείται πολύ πιο γρήγορα στους Η/Υ του επόμενου έτους !!!

Ο χρόνος εκτέλεσης του Fib1(n) είναι ανάλογος του

$$2^{0.694n} \approx (1.6)^n$$

Σήμερα μπορούμε να υπολογίσουμε το Fib1(100), το επόμενο έτος το Fib1(101), το επόμενο το Fib1(102), … **Έναν μόνο όρο κάθε έτος !!!**

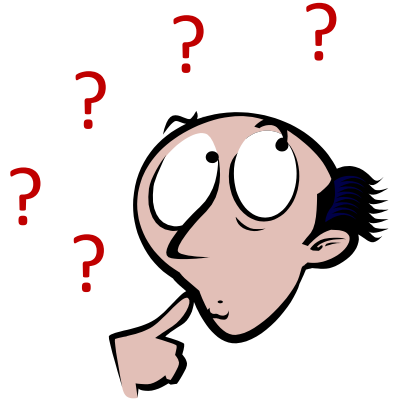
- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

Γιατί ο αλγόριθμος που σχεδιάσαμε είναι τόσο πολύ αργός?

Πρέπει να τον μελετήσουμε και να απαντήσουμε την ερώτηση !!!

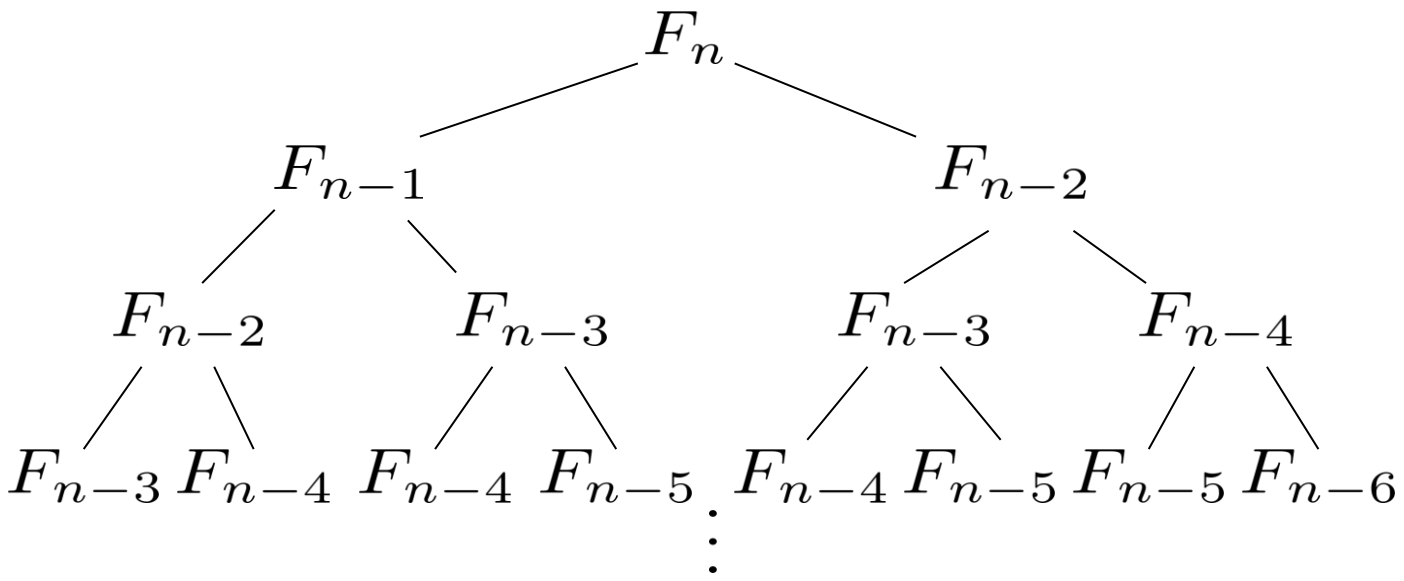


Κακός και Καλός Σχεδιασμός Αλγορίθμου

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

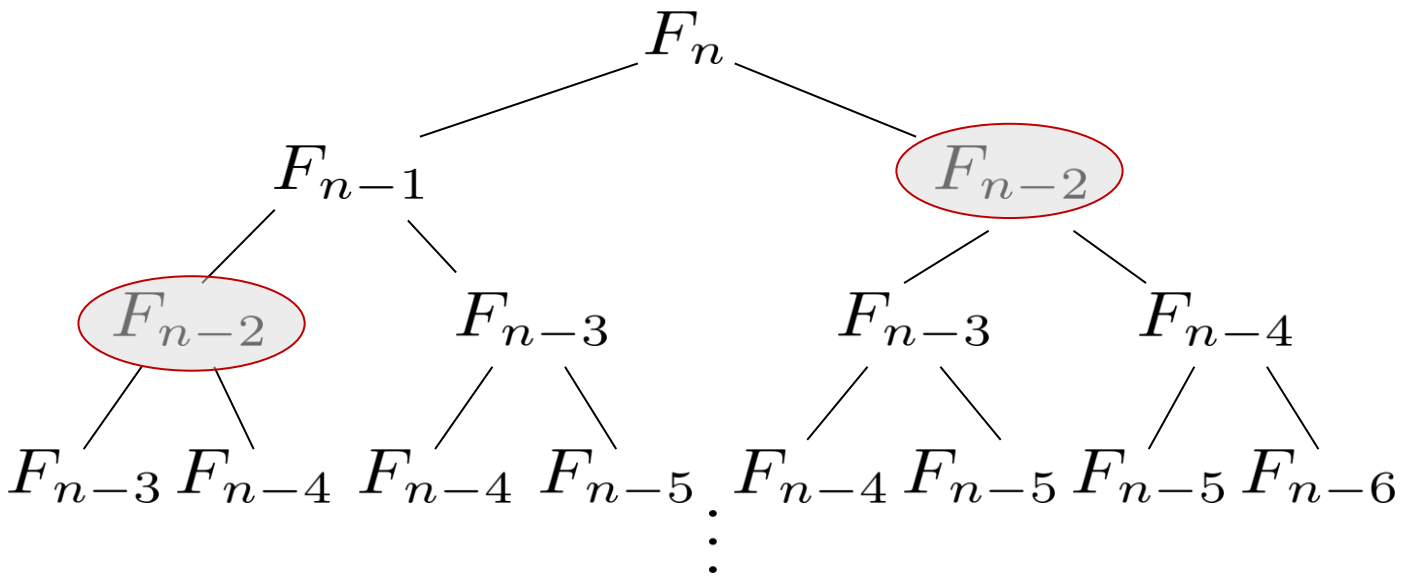


Κακός και Καλός Σχεδιασμός Αλγορίθμου

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

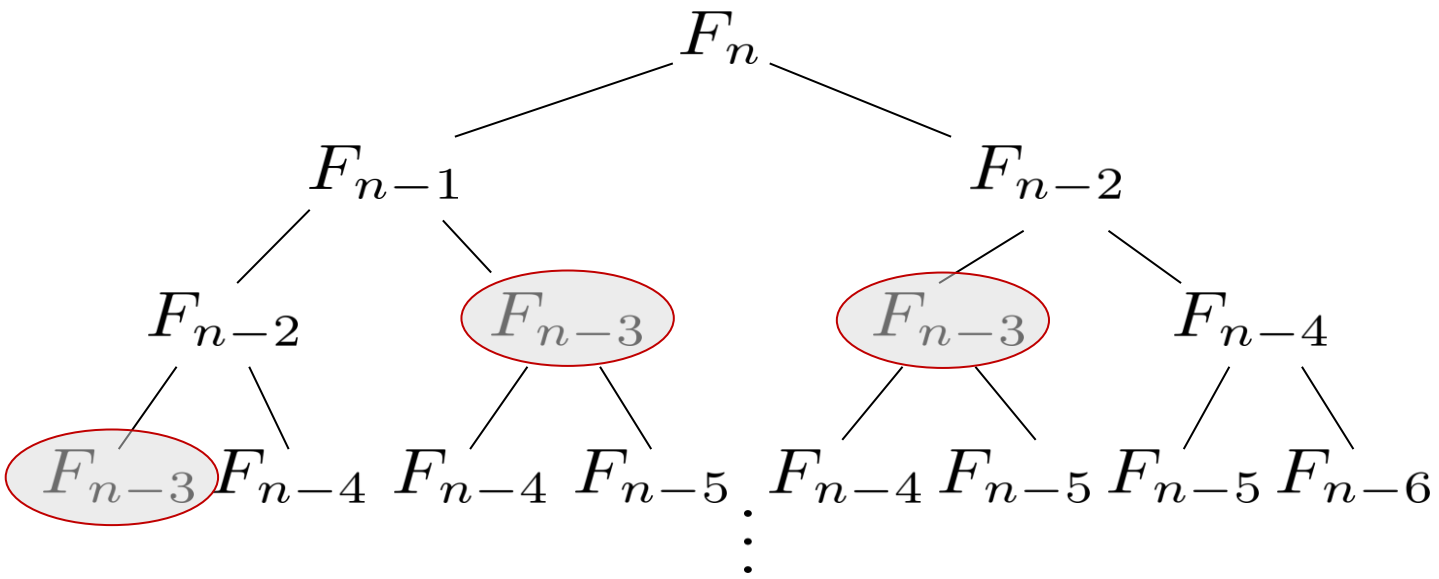


Κακός και Καλός Σχεδιασμός Αλγορίθμου

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

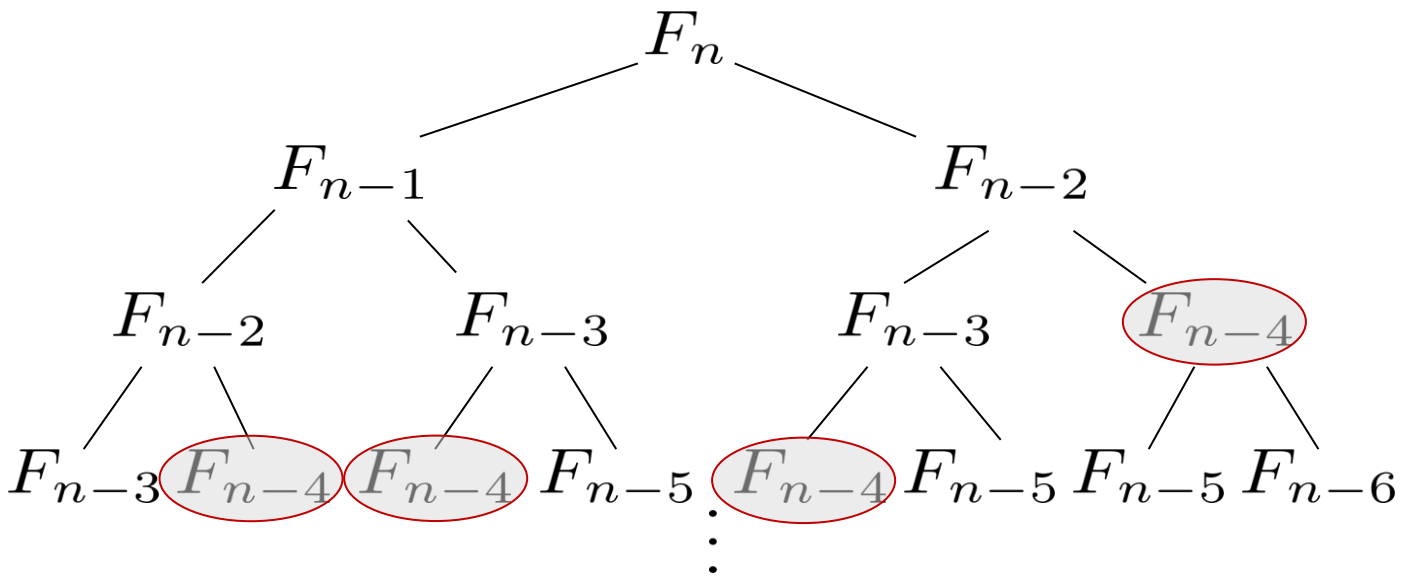


Κακός και Καλός Σχεδιασμός Αλγορίθμου

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

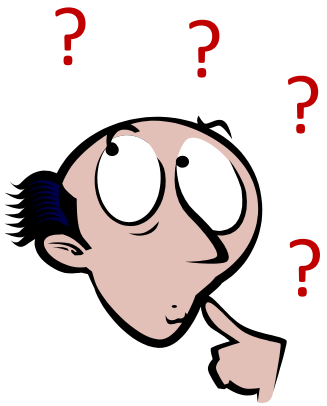


- **Ακολουθία Fibonacci**

Αλγόριθμος Fib1

```
Fib1(n)
    if n = 0 then return 0
    if n = 1 then return 1
    return Fib1(n-1)+Fib1(n-2)
```

3. **Μπορούμε να τον βελτιώσουμε;**



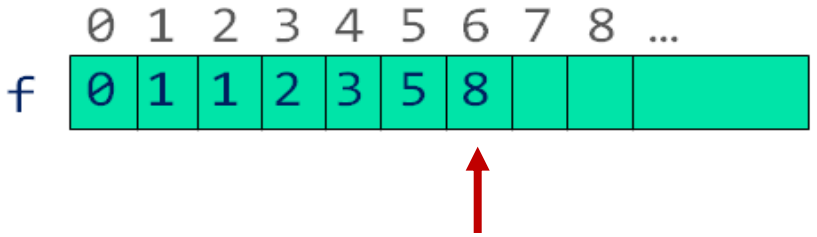
Ιδέα !!!

Να αποθηκεύουμε τα ενδιάμεσα αποτελέσματα !!!

Ακολουθία Fibonacci

Αλγόριθμος Fib2

```
Fib2(n)

    if n = 0 then return 0

    Δημιούργησε πίνακα f[0..n]

    Θέσε f[0] ← 0 και f[1] ← 1

    for i = 2 to n do
        f[i] = f[i-1] + f[i-2]

    return f[n]
```




```
f[6] = f[5]+f[4]
     = 8
```

Κακός και Καλός Σχεδιασμός Αλγορίθμου

Ακολουθία Fibonacci

Αλγόριθμος Fib2

```
Fib2(n)

if n=0 then return 0
Δημιούργησε πίνακα f[0..n]
Θέσε f[0]=0 και f[1]=1
for i=2 to n do
        f[i]=f[i-1]+f[i-2]

return f[n]
```

1. **Είναι σωστός;**

   Η ορθότητα προκύπτει
   άμεσα από τον ορισμό του F_n

2. **Πόσο χρόνο απαιτεί;**

   Ο εσωτερικός βρόχος έχει 1 μόνο υπολογιστικό βήμα και εκτελείται n-1 φορές !!!

$$T(n) = O(n)$$

Τώρα με τον Fib2(n) μπορούμε να υπολογίσουμε όχι μόνο τον όρο F_{200}
αλλά και τον F_{200000} !!!

Κακός και Καλός Σχεδιασμός Αλγορίθμου

- **Ακολουθία Fibonacci**

Αλγόριθμος Fib2

3. **Μπορούμε να τον βελτιώσουμε;**

```
Fib2(n)

if n=0 then return 0
Δημιούργησε πίνακα f[0..n]
Θέσε f[0]=0 και f[1]=1
for i=2 to n do
        f[i]=f[i-1]+f[i-2]

return f[n]
```

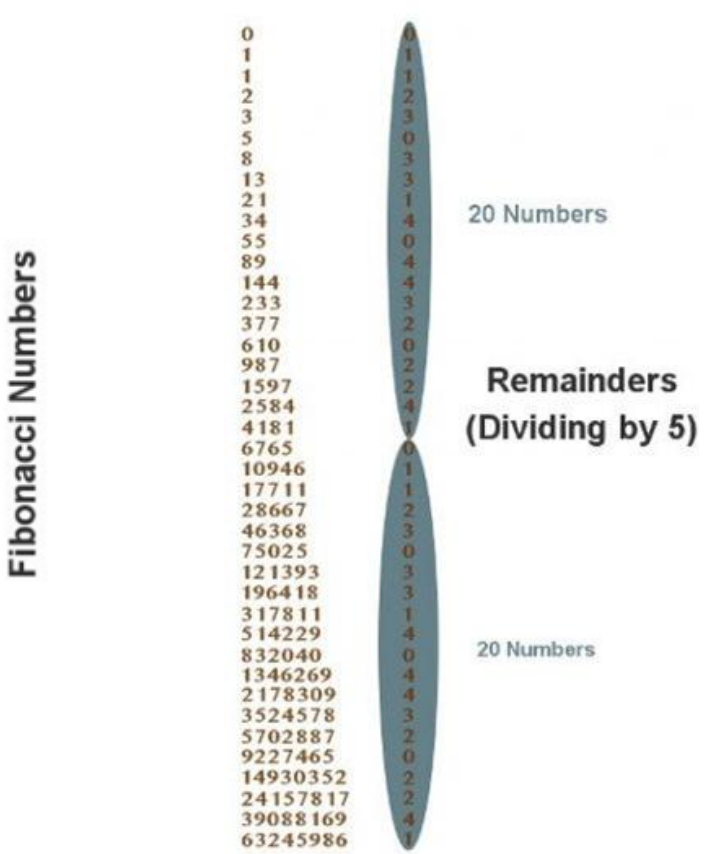
ΝΑΙ !!!

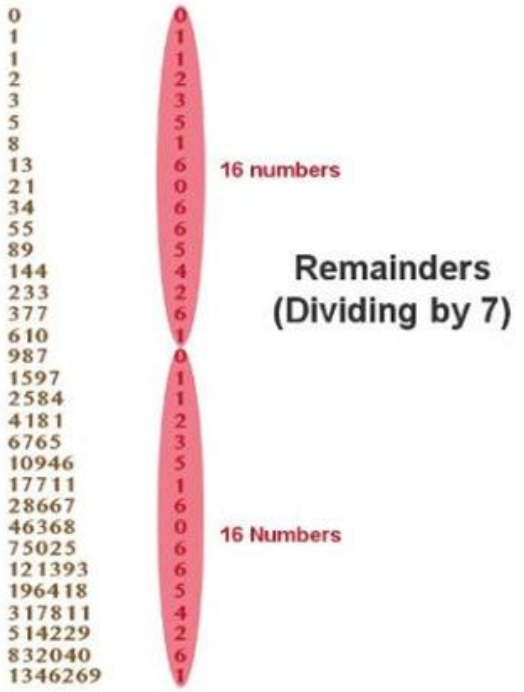
Ιδέα !!!

Πολλαπλασιασμός Πινάκων!!!!

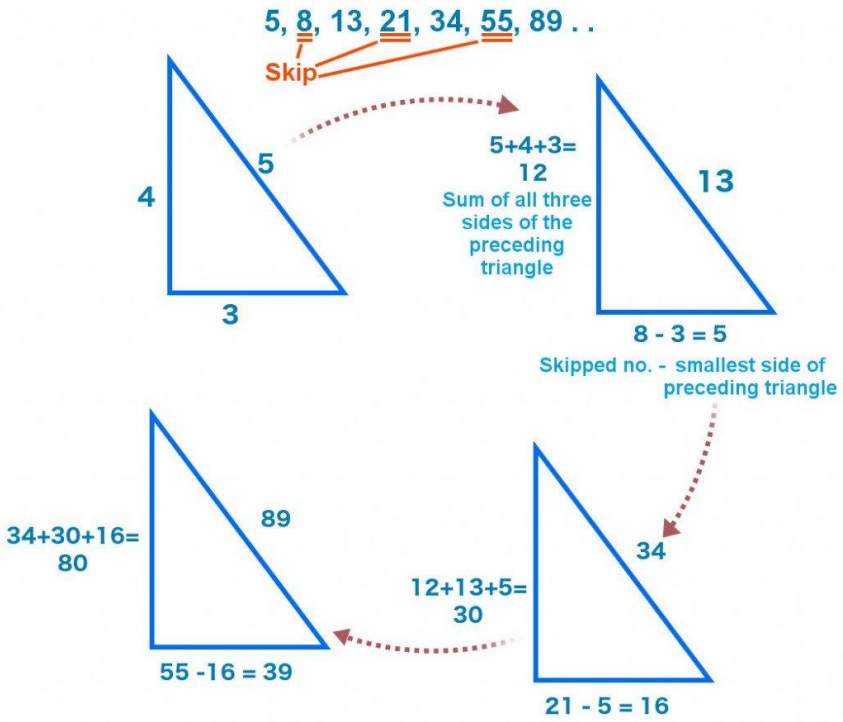
… Αλγόριθμος Fib3

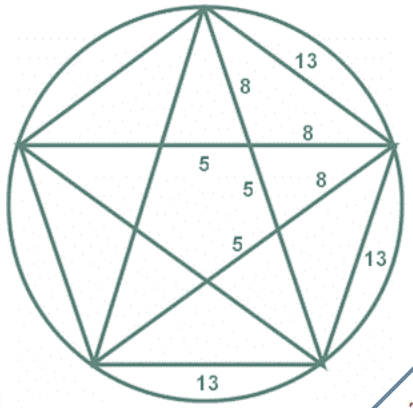
Ακολουθία Fibonacci - Ιδιότητες

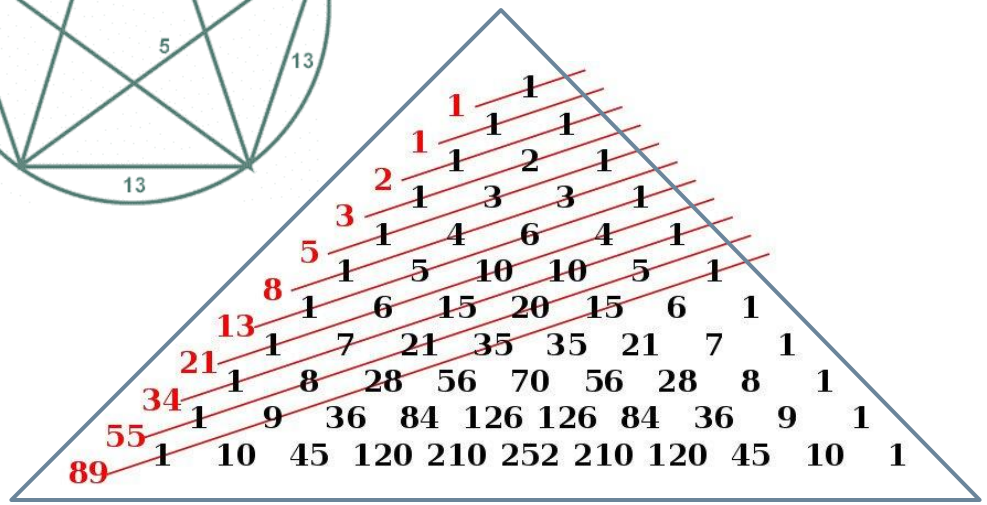




Ακολουθία Fibonacci - Ιδιότητες





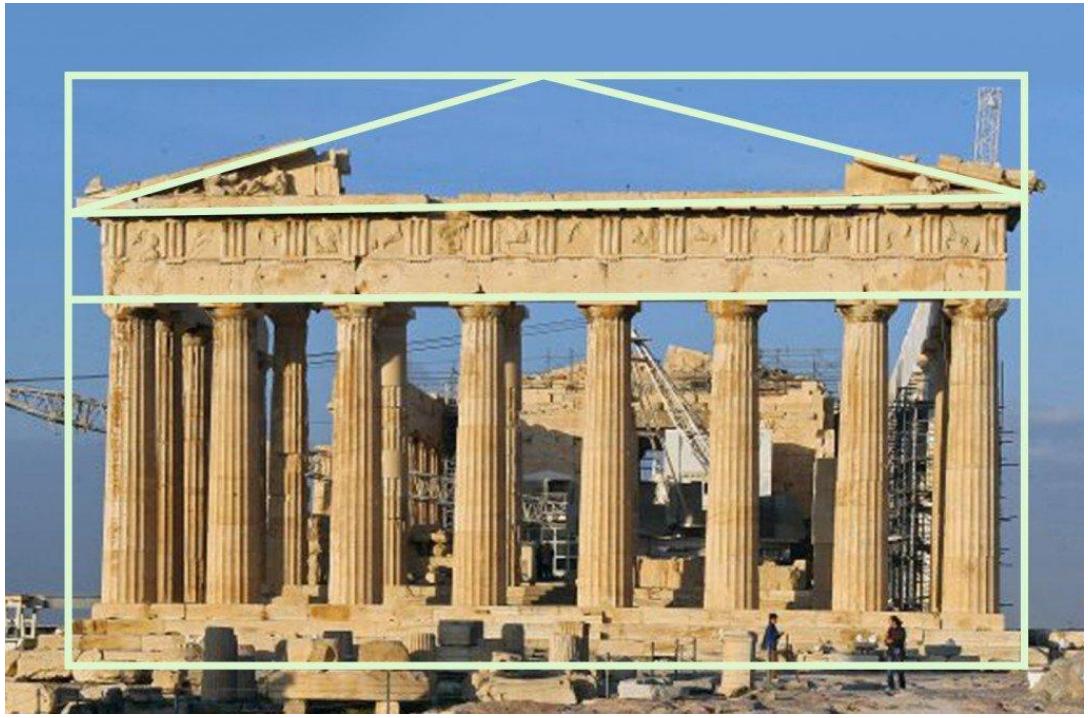


Pascal's triangle

Ιδιότητες Ακολουθίας Fibonacci

Ακολουθία Fibonacci - Ιδιότητες

Golden Ratio:
long_part/*short_part* = *whole_line*/*long_part* = 1.618…



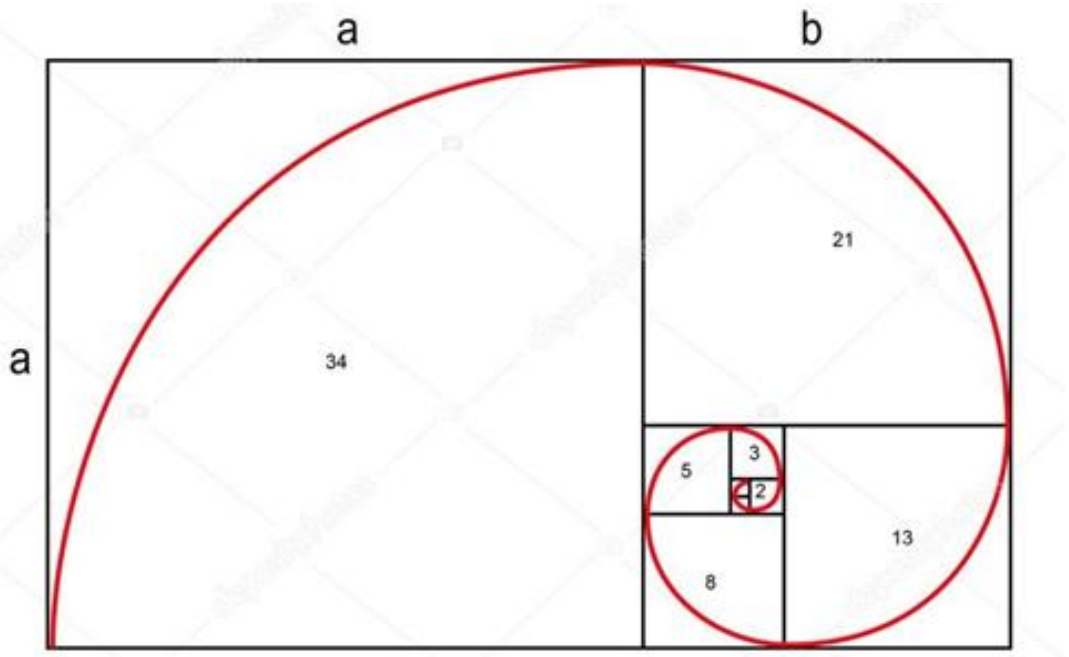
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

```
…
5/3     =  1.666
8/5     =  1.600
13/8    =  1.625
…
55/34   =  1.617
89/55   =  1.618
144/89  =  1.617
…
```

Ιδιότητες Ακολουθίας Fibonacci

Ακολουθία Fibonacci - Ιδιότητες

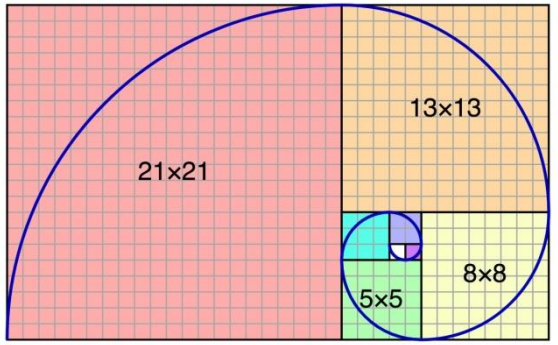
Golden Spiral:



$$\Phi = \frac{a+b}{a} = \frac{a}{b} = 1{,}618$$

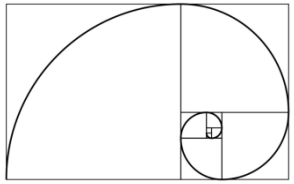
$$F_n = F_{n-1} + F_{n-2}$$

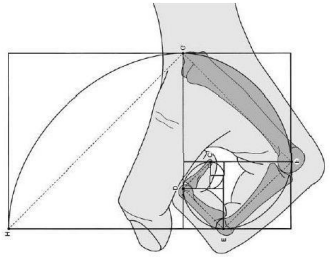
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

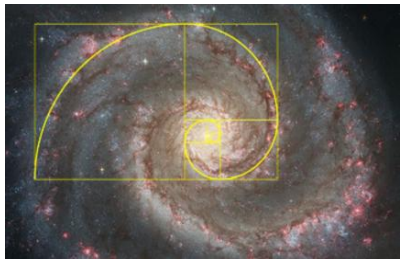


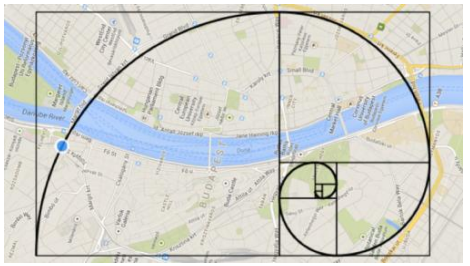
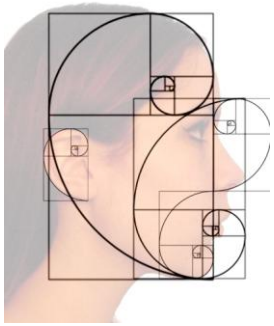
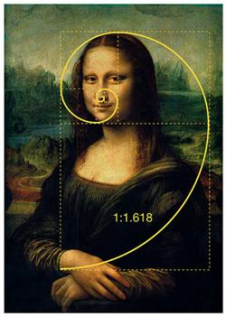
Ακολουθία Fibonacci - Ιδιότητες

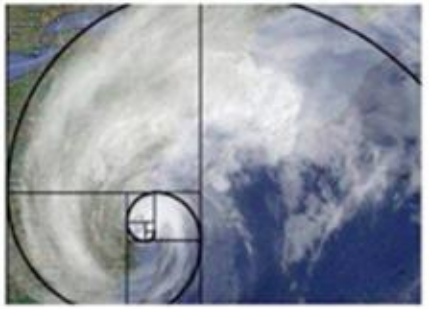
Golden Spiral:





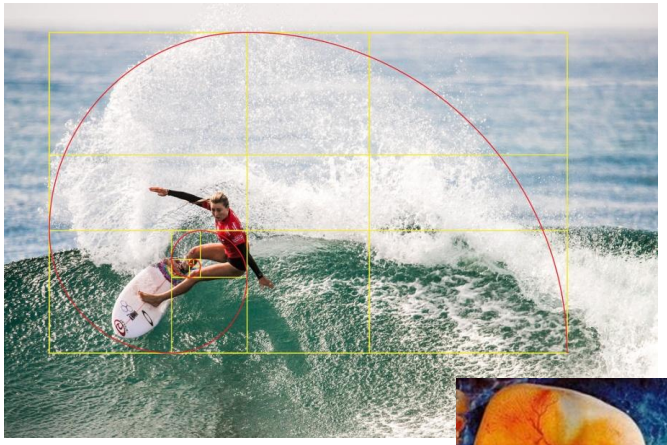



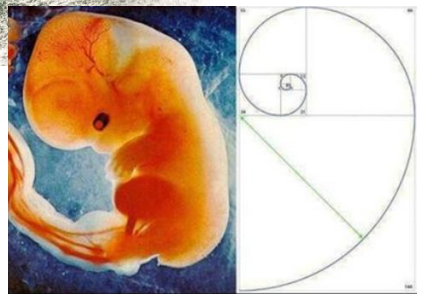





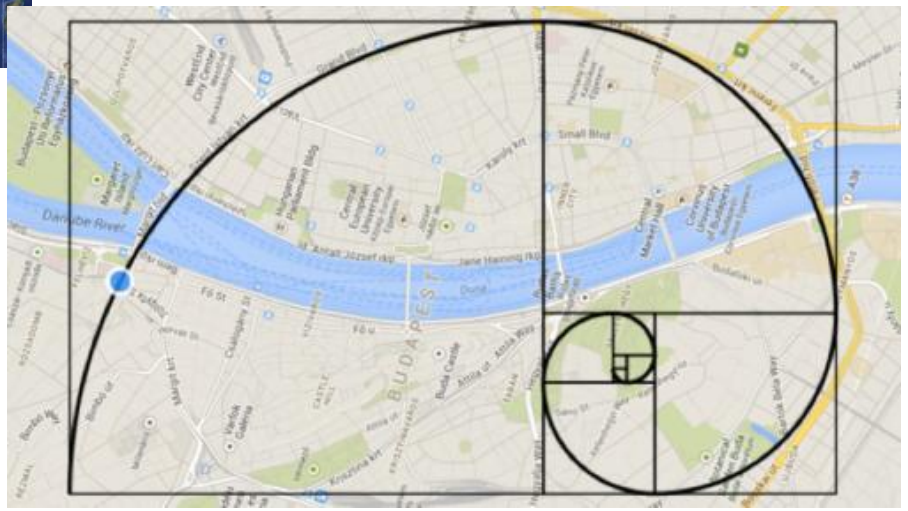




Ακολουθία Fibonacci - Ιδιότητες

Golden Spiral:

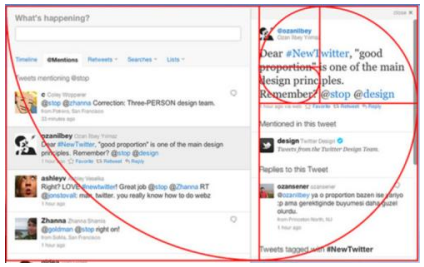


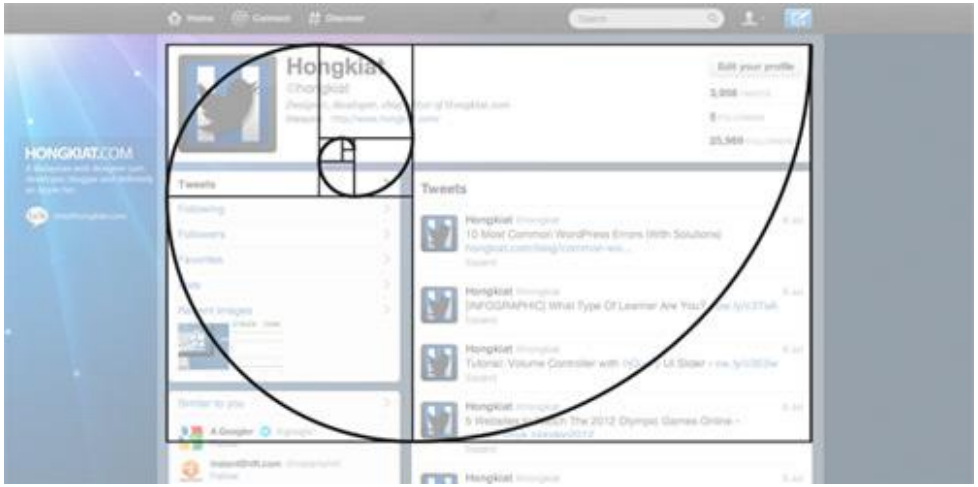




Ακολουθία Fibonacci - Ιδιότητες

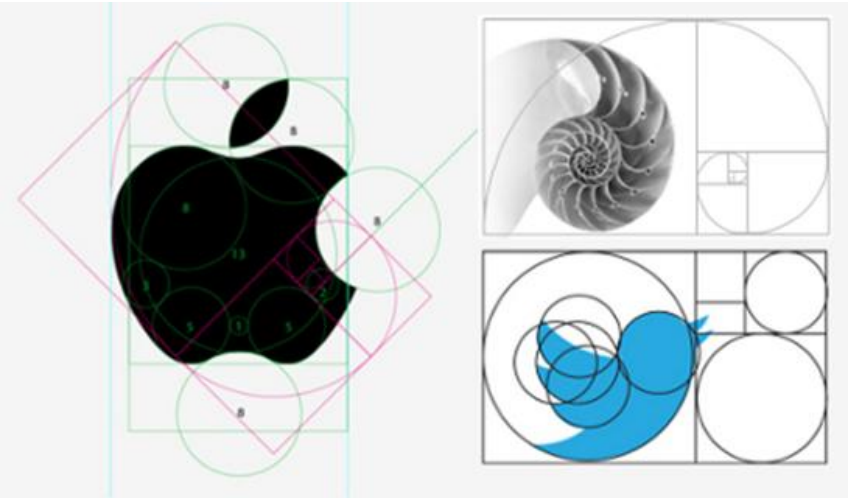
Golden Spiral:

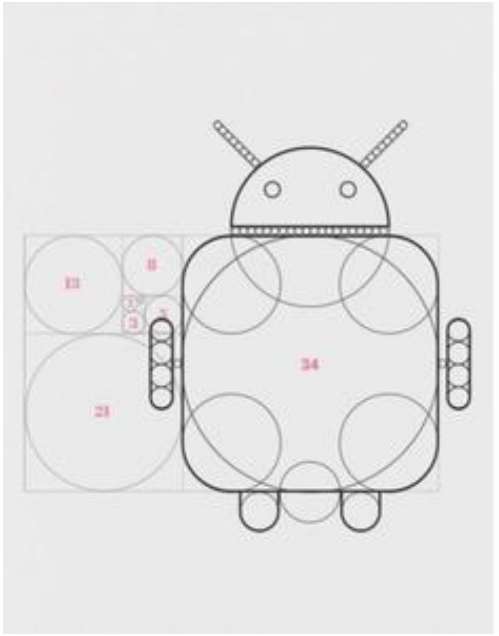




Ακολουθία Fibonacci - Ιδιότητες

Golden Spiral:





Ευχαριστώ για τη Συμμετοχή σας !!!

Σταύρος Δ. Νικολόπουλος

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

Webpage: www.cs.uoi.gr/~stavros