Διάλεξη 14-15: - Πολυπλοκότητα Αλγορίθμων / Επανάληψη Χρήσιμων Μαθηματικών Ορισμών

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Αλγόριθμοι, Κριτήρια Αξιολόγησης Αλγόριθμων, Γιατί αναλύουμε τους Αλγόριθμους
- Εμπειρική – Θεωρητική Ανάλυση Αλγορίθμων, Αρχή Σταθερότητας
- Εργαλεία εκτίμησης πολυπλοκότητας: οι τάξεις O(n), Ω(n), Θ(n)
- Γραφική Απεικόνιση O(n), Ω(n), Θ(n)
- Παραδείγματα Ανάλυσης Πολυπλοκότητας
- Χρήσιμοι μαθηματικοί ορισμοί
- Μαθηματική Επαγωγή

Διδάσκων: Παναγιώτης Ανδρέου

Αλγόριθμοι

Αλγόριθμος: είναι μια πεπερασμένη ακολουθία εντολών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, οι οποίες αν ακολουθηθούν επιτυγχάνεται κάποιο επιθυμητό αποτέλεσμα.

- **Προγράμματα:** Υλοποίηση αλγορίθμου σε μια γλώσσα προγραμματισμού

Ένα σωστό πρόγραμμα δεν είναι αρκετό

- **Τι είναι σημαντικό σε ένα Αλγόριθμο;**
  - **Ορθότητα:** όταν τα δεδομένα εισόδου ικανοποιούν τις αναγκαίες συνθήκες, τότε το πρόγραμμα τερματίζει με το αναμενόμενο αποτέλεσμα.
  - **Χρόνος Εκτέλεσης:** Ο χρόνος που χρειάζεται το πρόγραμμα για να δημιουργήσει το αναμενόμενο αποτέλεσμα.
  - **Απαίτηση στην Μνήμη, Δίκτυο, Μαγνητικό Δίσκο (ανάλογα με την εφαρμογή)**: Τους υπολογιστικούς πόρους τους οποίους χρειάζεται η εφαρμογή, πέρα από τον επεξεργαστή, για να δημιουργηθεί το αναμενόμενο αποτέλεσμα.

Ανάλυση Αλγορίθμων

- Θέλουμε να μπορούμε να υπολογίζουμε τον χρόνο εκτέλεσης ενός αλγορίθμου / και να τον βελτιώνουμε, **προτού** τον τρέξουμε σε κάποια μηχανή.

Η σημερινή διάλεξη θα απαντήσει τις ερωτήσεις:

- Τι είναι η θεωρητική ανάλυση ενός αλγόριθμου;

- Ποια είναι τα κριτήρια ανάλυσης αλγορίθμων;

- Πως παριστάνουμε και υπολογίζουμε την απόδοση ενός αλγορίθμου με μαθηματικά εργαλεία;

Ανάλυση Αλγορίθμων – Γιατί?

Γιατί πρέπει να αναλύσουμε την πολυπλοκότητα ενός αλγορίθμου;

- Όταν ένα πρόγραμμα πρόκειται να επεξεργαστεί ένα μεγάλο όγκο δεδομένων, πρέπει να βεβαιωθούμε ότι θα **τερματίσει σε αποδεκτά χρονικά πλαίσια.**
  - Αν ένας αλγόριθμος εύρεσης του πιο φθηνού εισιτηρίου σε ένα ηλεκτρονικό σύστημα αεροπορικών κρατήσεων θέλει 2 μήνες για να εκτελεστεί, τότε το σύστημα δεν είναι χρήσιμο.
- Η θεωρητική ανάλυση, επιτρέπει σε ένα προγραμματιστή να **υπολογίσει με την χρήση μαθηματικών** εργαλείων τον αναμενόμενο χρόνο **προτού κωδικοποιήσει** το πρόγραμμα.
  - Εάν ο αναμενόμενος χρόνος είναι πολύ μεγάλος, τότε θα σχεδιάσει κάποιο βελτιωμένο αλγόριθμο.

Εμπειρική – Θεωρητική Ανάλυση Αλγορίθμων

Ένας αλγόριθμος μπορεί να μελετηθεί:

- **Εμπειρικά,** μετρώντας το χρόνο και χώρο εκτέλεσής του σε συγκεκριμένο υπολογιστή. Μπορούμε να υλοποιήσουμε τον αλγόριθμο και να μετρήσουμε τον χρόνο με την βιβλιοθήκη <time.h>
- **Θεωρητικά,** μπορούμε να υπολογίσουμε το χρόνο και το χώρο που απαιτεί ο αλγόριθμος σαν συνάρτηση του **μεγέθους** των εξεταζομένων στιγμιότυπων.

  π.χ. ***Πρόβλημα:*** *ταξινόμηση λίστας.*

  Στιγμιότυπο: *λίστα με n στοιχεία.*

  Μέγεθος: *n*
- Τυπικά, **μέγεθος ενός στιγμιότυπου** αντιστοιχεί στο μέγεθος της μνήμης που απαιτείται για αποθήκευση του στιγμιότυπου στον υπολογιστή.
- **Θεωρητική Ανάλυση Vs. Εμπειρική Ανάλυση**

  Η θεωρητική ανάλυση:

  1. δεν εξαρτάται από το υλικό του Η/Υ (μνήμη, cache, κλπ)
  2. δεν εξαρτάται από τη γλώσσα προγραμματισμού ή το μεταφραστή
  3. δεν εξαρτάται από τις ικανότητες του προγραμματιστή.
  4. δεν απαιτεί την υλοποίηση του αλγόριθμου προτού την δοκιμή!
  5. είναι ΓΕΝΙΚΗ

Θεωρητική Ανάλυση Αλγορίθμων

- Κάνουμε χρήση μιας **υψηλού επιπέδου περιγραφής του αλγόριθμου** και όχι της υλοποίησης του – δηλαδή χρήση **Ψευδοκώδικα**.
- Θα χαρακτηρίσουμε τον **χρόνο εκτέλεσης του αλγόριθμου σαν συνάρτηση του μεγέθους των δεδομένων, N**
- **Η ανάλυση μας θα λάβει υπόψη όλες τις δυνατές εισόδους.**

  π.χ. βρείτε τον αριθμό 5 σε ένα πίνακα ακέραιων.

  - Στην καλύτερη περίπτωση: το βρίσκουμε στην πρώτη θέση

  - Στην χειρότερη περίπτωση: δεν τον βρίσκουμε καθόλου
- Επομένως, ένα αλγόριθμος μπορεί να εκτελείται **γρηγορότερα** σε συγκεκριμένα σύνολα δεδομένων από ότι σε κάποια άλλα.
- Η εύρεση της **μέσης περίπτωσης** είναι συνήθως δύσκολη, για αυτό οι αλγόριθμοι μετριούνται με βάση την **χειρότερη περίπτωση.**
- Σε συγκεκριμένα πεδία (air traffic control, συστήματα πραγματικού χρόνου, κτλ), η γνώση της χειρότερης περίπτωσης είναι πολύ σημαντική

Παράδειγμα Ανάλυσης 1

Υποθέστε ότι θέλετε να βρείτε το μεγαλύτερο στοιχείο σε μια λίστα θετικών ακεραίων.

```
int largest( int X[], int n){
    int current=0, i=0;
    while ( i < n ){
        if ( X[i] > current){
            current = X[i]; // σημείωσε τον μεγαλύτερο
        }
        i = i+1;
    }
    return current;
}
```

***Μέγεθος δεδομένων εισόδου*: n**

Στόχος: υπολογισμός του αριθμού βασικών πράξεων

Χειρότερη Περίπτωση: εξέταση όλων των στοιχείων $t(n) = n$

Βέλτιστη Περίπτωση: εξέταση όλων των στοιχείων $t(n) = n$

Παράδειγμα Ανάλυσης 2

Αν ήταν ταξινομημένη η λίστα (σε αύξουσα σειρά) τότε το μεγαλύτερο στοιχείο μπορεί να βρεθεί σε σταθερό χρόνο (δηλαδή πάντα το τελευταίο στοιχείο).

```
int largest2( int X[], int n){
      if (n<0)
            return -1; // error;
      return X[n-1];
}
```

Μέγεθος δεδομένων εισόδου: n

Χειρότερη Περίπτωση: εξέταση τελευταίου στοιχείου $t(n) = 1$

Βέλτιστη Περίπτωση: εξέταση τελευταίου στοιχείου $t(n) = 1$

Μονάδα Σύγκρισης της Αποδοτικότητας

Η αρχή της σταθερότητας

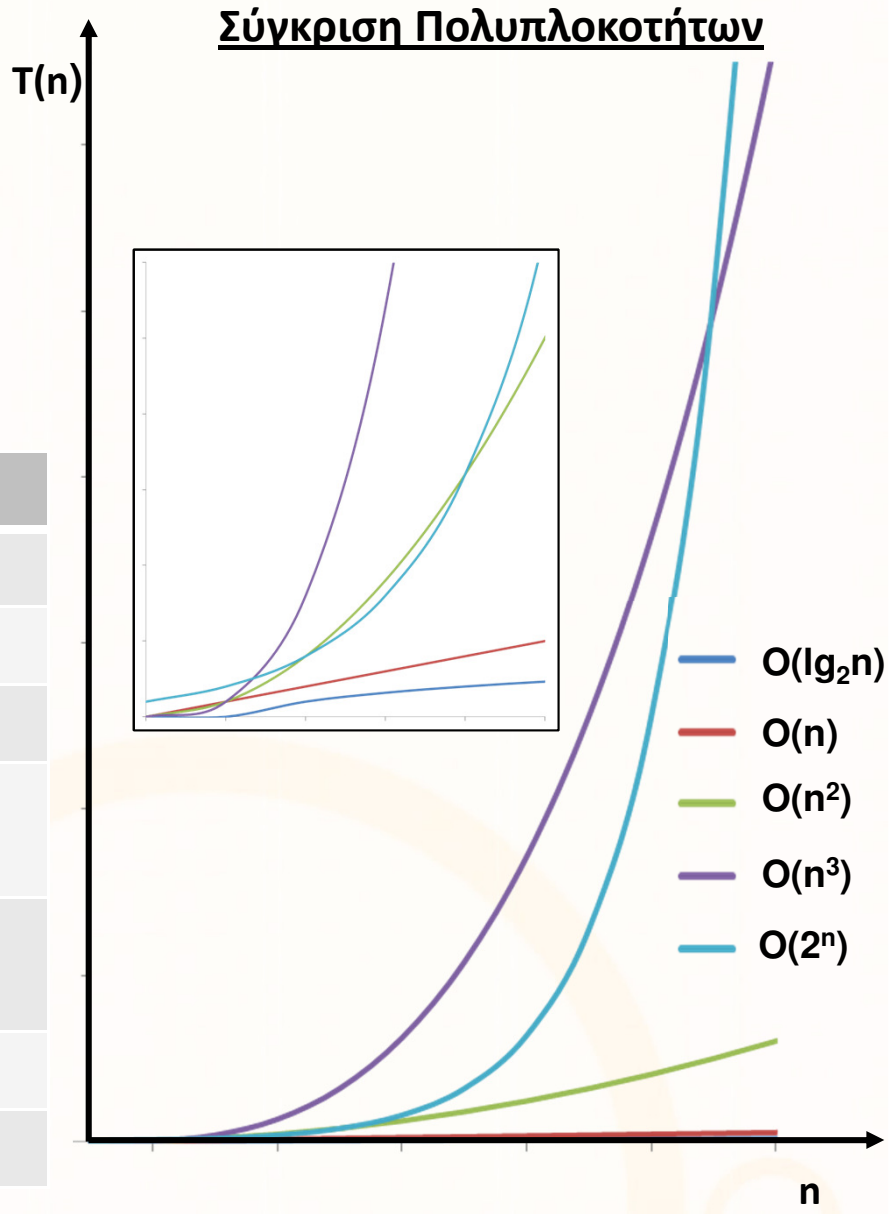
Δύο διαφορετικές υλοποιήσεις του ίδιου αλγορίθμου (σε διαφορετικές μηχανές ή σε διαφορετικές γλώσσες ή από διαφορετικούς προγραμματιστές) **δεν διαφέρουν** στο χρόνο εκτέλεσής τους **περισσότερο** από κάποιο **σταθερό πολλαπλάσιο**. δηλ. αν $\mathbf{E_1}$ είναι ο χρόνος εκτέλεσης της μίας υλοποίησης και $\mathbf{E_2}$ της άλλης, τότε ισχύει $\mathbf{E_1 = c \cdot E_2}$ για κάποια σταθερά **c**.

- Για παράδειγμα έχουμε ένα πρόγραμμα το οποίο ταξινομεί ένα πίνακα μεγέθους N, στον υπολογιστή PC του σπιτιού μας σε χρόνο E1 και σε ένα IBM Blue Gene σε χρόνο E2.
- Παρατηρούμε ότι ισχύει : E1 = 1,000,000 * E2
- Η σταθερά c=1,000,000 δεν εξαρτάται από το N αλλά από το περιβάλλον εκτέλεσης (μεταγλωττιστής, υλικό, λειτουργικό σύστημα, κτλ).
- Επομένως εάν αυξηθεί το N, τότε και πάλι θα εκτελείται 1,000,000 πιο αργά στο PC μας

Σύγκριση Πολυπλοκότητας Αλγορίθμων

- Για την αξιολόγηση των διαφόρων αλγορίθμων χρησιμοποιούνται διάφορα μεγέθη.
- n: μέγεθος του προβλήματος
- T(n): Ο χρόνος εκτέλεσης

t(n)	Περιγραφή
1 (ένα)	Σταθερός (constant)
n	Γραμμικός (linear)
n^k	Πολυωνυμικός (polynomial)
n^2	Τετραγωνικός – δευτεροβάθμιος (quadratic), π.χ., Shortest Path Dijkstra
n^3	Κυβικός - τριτοβάθμιος (cubic), π.χ., All-Pair-Shortest Path Floyd-Warshall
c^n , n^n	Εκθετικός (exponential)
$\log_k n$	Λογαριθμικός (logarithmic)



Ρυθμός Αύξησης μιας Συνάρτησης

- Ο ρυθμός αύξησης μιας συνάρτησης είναι σημαντικός όταν ο αριθμός των δεδομένων εισόδου N, είναι σημαντικά μεγάλος.

  Για n=4 2^n=16 και n^2=16

  Για n=100 2^n=**$1.26*10^{30}$** και n^2=**10,000**

- Εάν ένας αλγόριθμος εκτελείται σε 2^n και μια άλλη έκδοση n^2 πράξεις τότε για μικρές τιμές θα έχουν παρόμοιο χρόνο εκτέλεσης. Για μεγάλες τιμές αυτό βέβαια δεν ισχύει!

Ρυθμός Αύξησης μιας Συνάρτησης (συν.)

O(n)	10	20	30	50	100	500	1000
$Log_{10}n$	0.000006 second	0.000007 second	0.000008 second	0.000009 second	0.000012 second	0.000015 second	0.000018 second
n	0.00006 second	0.00012 Second	0.00018 second	0.0003 second	0.0006 second	0.003 second	0.006 second
n^2	0.0006 second	0.0024 second	0.0054 second	0.015 second	0.06 second	1.6 seconds	6.0 seconds
n^3	0.006 second	0.048 second	0.162 second	0.75 second	6.0 seconds	12.5 minutes	1.66 ΩΡΕΣ
n^4	0.06 second	0.96 second	4.86 second	37.5 second	10.0 minutes	4.34 ΗΜΕΡΕΣ	2.3 ΜΗΝΕΣ
n^5	0.6 second	19.2 seconds	2.43 minutes	31.25 minutes	16.66 ΩΡΕΣ	6.02 ΧΡΟΝΙΑ	ΑΙΩΝΕΣ
2^n	0.005 second	5.0 seconds	1.5 ΩΡΕΣ	ΑΙΩΝΕΣ	ΑΙΩΝΕΣ	ΧΙΛ/ΔΕΣ ΑΙΩΝΕΣ	ΧΙΛ/ΔΕΣ ΑΙΩΝΕΣ
3^n	0.3 second	4.8 ΩΡΕΣ	30.1 ΧΡΟΝΙΑ	ΧΙΛ/ΔΕΣ ΑΙΩΝΕΣ	ΧΙΛ/ΔΕΣ ΑΙΩΝΕΣ	ΗΛΙΚΙΑ ΓΗΣ	ΗΛΙΚΙΑ ΓΑΛΑΞΙΑ
n!	0.36 minutes	12.5 ΩΡΕΣ	ΑΙΩΝΕΣ	ΧΙΛ/ΔΕΣ ΑΙΩΝΕΣ	ΗΛΙΚΙΑ ΗΛΙΟΥ	ΗΛΙΚΙΑ ΓΑΛΑΞΙΑ	∞
Μέγεθος Προβ/τος		**Προβλήματα Μικρού Μεγέθους**		**Προβλήματα Μεσαίου Μεγέθους**		**Προβλήματα Μεγάλου Μεγέθους**	

Ρυθμός Αύξησης μιας Συνάρτησης (συν.)

- **Σημειώστε ότι η τιμή μιας συνάρτηση, καθορίζεται κυρίως από τον μεγαλύτερο όρο για μεγάλες τιμές του n.**

- Π.χ. Ένας αλγόριθμος μας θέλει f(n) χρόνο για να ολοκληρώσει:
  $f(n) = 10*n^3 + n^2 + 40*n + 80$

- Για n=1000 ➔ f(n) = 10,001,040,080

- Χρησιμοποιώντας μόνο τον πιο μεγάλο όρο $\mathbf{10*n^3}$ =10,000,000,000 θα μπορούσαμε να υπολογίσουμε τον χρόνο εκτέλεσης του προγράμματος με 99.99% ακρίβεια!

- Επομένως τα $n^2 + 40*n + 80$ είναι αχρείαστα στην ανάλυση μας

Μοντέλο Υπολογισμού της ανάλυσης μας

- Υπολογιστής που εκτελεί οδηγίες διαδοχικά (όχι παράλληλες μηχανές).
- **Βασική πράξη** θεωρούμε ότι είναι **οποιαδήποτε πράξη** της οποίας ο χρόνος εκτέλεσης είναι φραγμένος από κάποια σταθερά (δηλ. $\leq c$, για κάποια σταθερά c). *Π.χ., μαθηματικές πράξεις (πρόσθεση, αφαίρεση…), σύγκριση, καταχώρηση μεταβλητής, επιστροφή αποτελέσματος*.
- Συνεπώς ο **χρόνος εκτέλεσης** μιας βασικής πράξης μπορεί να προσδιοριστεί **ανεξάρτητα** από **το στιγμιότυπο** (πχ 10+3 ή 10000+3 θέλουν τον ίδιο ακριβώς χρόνο εκτέλεσης).
- Επειδή ορίζουμε το χρόνο εκτέλεσης ενός αλγορίθμου με την έννοια του σταθερού πολλαπλασίου **c**, **για την ανάλυση θα χρειαστούμε μόνο τον αριθμό των βασικών πράξεων** που εκτελούνται από ένα αλγόριθμο και **όχι τον ακριβή χρόνο** που απαιτούν η κάθε μια.
- ***Άρα για τον υπολογισμό του χρόνου εκτέλεσης ενός αλγόριθμου απλά μετρούμε τον αριθμό των βασικών πράξεων που εκτελεί.***

Ασυμπτωτική Προσέγγιση: Big-O

- **Μας επιτρέπει να υπολογίσουμε τον ρυθμό αύξησης μιας συνάρτησης χωρίς να ανησυχούμε για:**

  a) σταθερά πολλαπλάσια (c) ή

  b) για μικρότερους όρους, οι οποίοι έτσι και αλλιώς δεν επηρεάζουν τον χρόνο εκτέλεσης ενός προγράμματος.

- Επομένως δεν έχουμε να ανησυχούμε για το υλικό, μεταγλωττιστή, κτλ. με τα οποία θα εκτελεστεί ο αλγόριθμος μας.

- Η προσέγγιση Big-O, μας επιτρέπει να ορίσουμε μια **σχετική σειρά** στις συναρτήσεις συγκρίνοντας τους επικρατέστερους όρους.

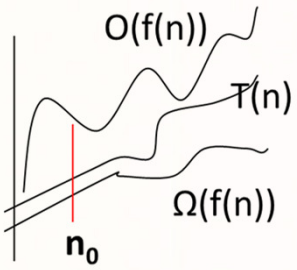
  π.χ. ένας αλγόριθμος θέλει **$f(n)=100n^2 + 17n + 4$** πράξεις ενώ ένας άλλος θέλει **$g(n)=n^3$** πράξεις τότε ξέρουμε ότι ο πρώτος εκτελείται πιο γρήγορα για μεγάλες τιμές του n

Εργαλεία Εκτίμησης Πολυπλοκότητας

Ορισμός: Θεωρούμε συνάρτηση T(n). Ορίζουμε

f(n)μπορεί να είναι οποιαδήποτε συνάρτηση => n, n^2, lgn

1. **T(n) ∈ O(f (n)),** αν υπάρχουν σταθερές c>0 και $n_0 \geq 1$ ώστε **T(n) ≤ c·f(n),** για κάθε $n \geq n_0$.



2. **T(n) ∈ Ω(f(n)),** αν υπάρχουν σταθερές c>0 και $n_0 \geq 1$ ώστε **T(n)≥c·f(n),** για κάθε $n \geq n_0$.

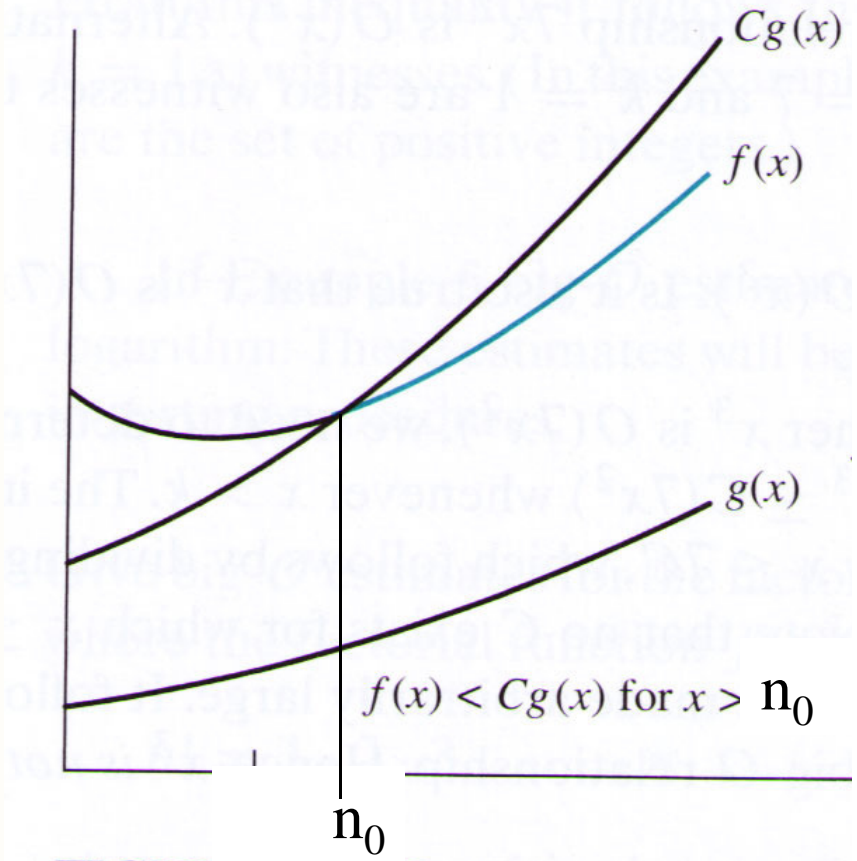
3. **T(n) ∈ Θ(f(n)),** αν T(n)∈O(f(n)) και T(n)∈Ω(f(n)).

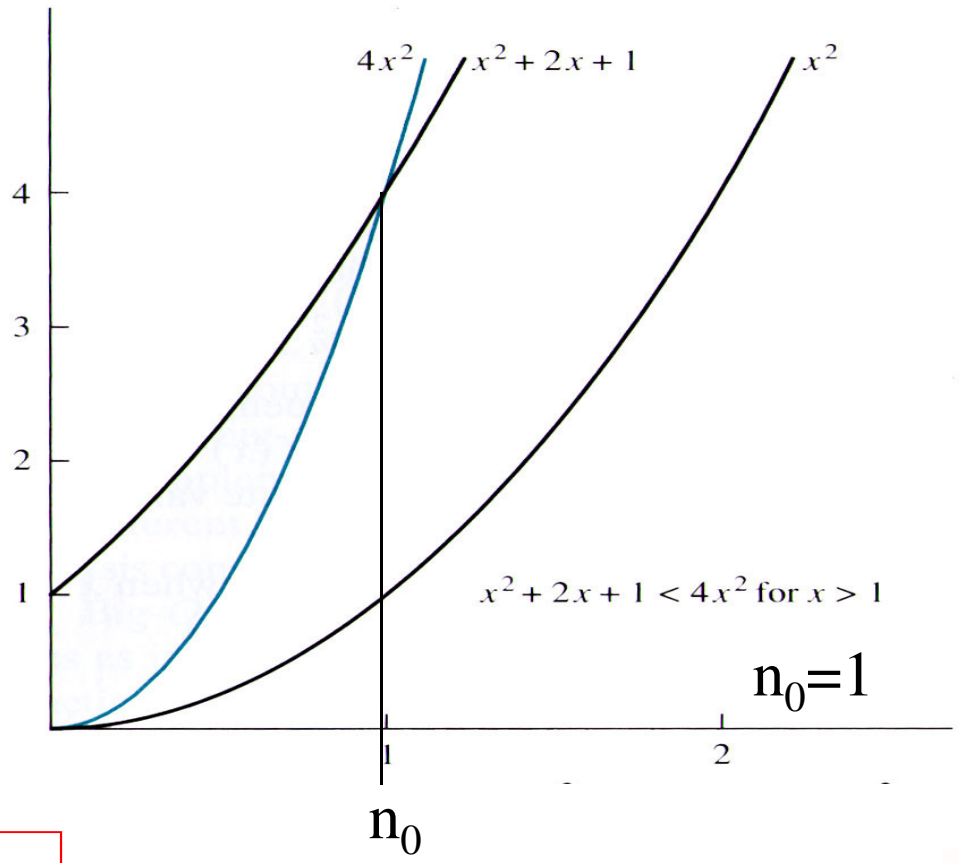
Αν T(n)∈O(f (n)), τότε λέμε πως η συνάρτηση T είναι της **τάξεως f(n).**

Αν T(n) ∈ Ω(f (n)), τότε λέμε πως η T είναι της **τάξεως ωμέγα της f (n).**

Αν T(n)∈ Θ(f (n)), τότε λέμε πως η T είναι της **τάξεως θήτα της f (n).**
(λέγεται και ακριβής τάξη)

Γραφική Απεικόνιση Big-O





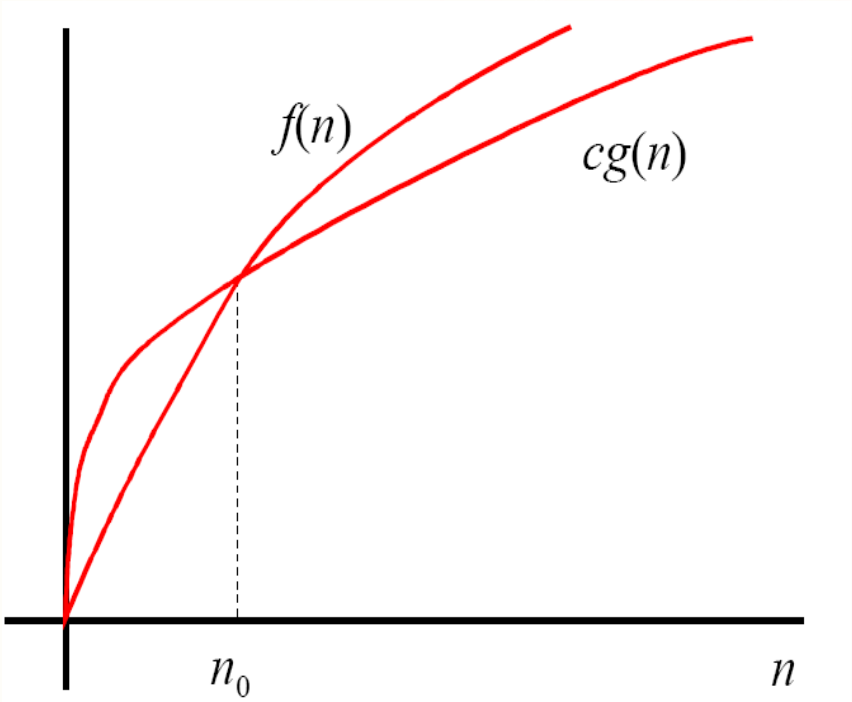
Η συνάρτηση f(x) είναι O(g(x))

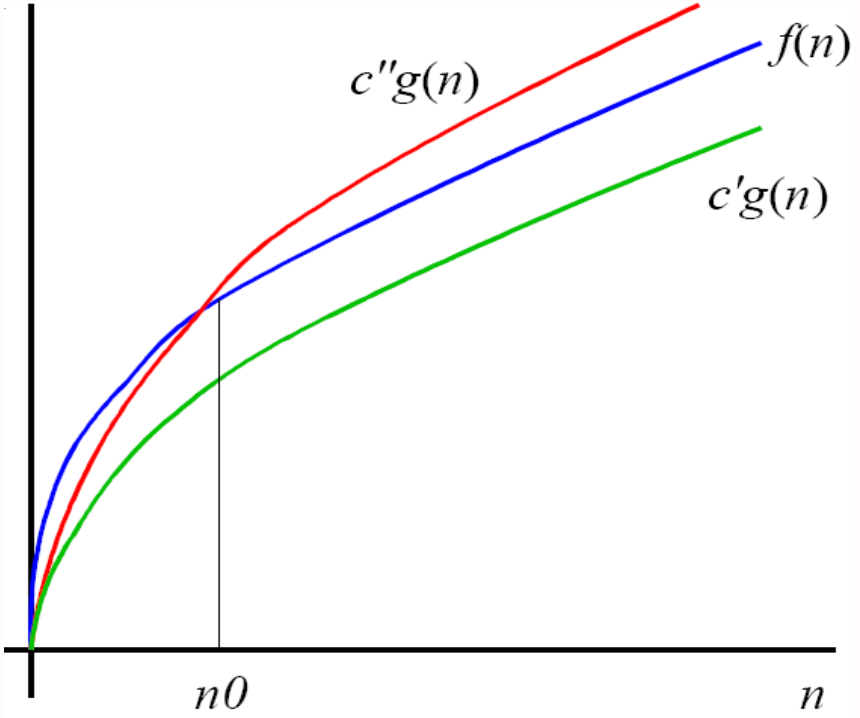
Δηλαδή υπάρχουν σταθερές c>0 και n0 ≥1 ώστε:

f(n) ≤ c·g(n), για κάθε n ≥ n0 .

Η συνάρτηση x^2+2x+1 είναι $O(x^2)$

Γραφική Απεικόνιση Big-Ω και Big-Θ





Η συνάρτηση f(n) ε Ω(g(n))

Δηλαδή υπάρχουν σταθερές c>0 και $n_0 \geq 1$ ώστε:

f(n)≥c·g(n), για κάθε $n \geq n_0$.

Η συνάρτηση f(n) ε Θ(g(n))

Δηλαδή υπάρχουν σταθερές c'>0,c''>0 και $n_0 \geq 1$ ώστε:

c'.g(n) ≤ f(n) ≤ c''.g(n), για κάθε $n \geq n_0$.

Παραδείγματα Big-O

Παράδειγμα 1

7n-3 είναι O(n)

Τεκμηρίωση

- **Πρέπει να δείξουμε ότι υπάρχει $c>0, n_o \geq 1$ έτσι ώστε: $7n-3 \leq c*n$,**
- Για c=7, $n_o \geq 1$ η παραπάνω ανισότητα ικανοποιείται, επομένως 7n-3 είναι **O(n)**

π.χ.,

- $7*1-3 \leq 7*1$ $(4 \leq 7)$
- $7*2-3 \leq 7*2$ $(11 \leq 14)$
- $7*3-3 \leq 7*3$ $(18 \leq 21)$
- κοκ

Παραδείγματα Big-O

Παράδειγμα 2

$20n^3+10n\log n+5$ είναι $O(n^3)$

Τεκμηρίωση

- **Πρέπει να δείξουμε ότι υπάρχει $c>0, n_o \geq 1$ έτσι ώστε**

$20n^3+10n\log n+5 \leq c^* n^3$

$20n^3+10n\log n+5 \leq 20n^3+10n^3+5n^3$

$20n^3+10n^3+5n^3 \leq 35n^3 \qquad (n>1)$

Για $c=35$, $n_o=1$ η παραπάνω ανισότητα ικανοποιείται,
επομένως **$20n^3+10n\log n+5$** είναι **$O(n^3)$**

Ιδιότητες Big-O

Στις περισσότερες περιπτώσεις μας ενδιαφέρει μόνο το άνω φράγμα (χείριστη περίπτωση εκτέλεσης ενός αλγορίθμου), δηλαδή το O.

Η συνάρτηση O έχει τις ακόλουθες ιδιότητες:

1. $O(X+Y) = O(X) + O(Y) \in \max(O(X),O(Y))$
2. $O(X*Y) = O(X)*O(Y) \in O(X * Y)$
3. $O(c * X) = O(X)$ *(c σταθερά > 0)*

Ισχύει η ακόλουθη σειρά
$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$
$< n^n$ **"<" είναι ασυμπτωτικά μικρότερο**

Παράδειγμα

Έστω ότι ένας αλγόριθμος έχει την ακόλουθη συνάρτηση αύξησης

$f(n) = 16n^2\log n + 3n + 7$

Τότε απλοποιούμε την συνάρτηση αύξησης ως εξής

$= 16n^2\log n$ (διαγράφουμε τις μικρότερες τιμές του n – Κανόνας 1)

$= n^2\log n$ (διαγράφουμε τις σταθερές – Κανόνας 3)

δηλαδή **$f(n) \in O(n^2\log n)$**

Παραδείγματα Ιδιοτήτων Big-O

Παραδείγματα:

$15n + 32 \in O(n)$ $\quad$ $200*(n+n^7) \in O(n^7)$

$1324 \in O(1)$ $\quad$ $(n+n^7)*m^2 \in O(n^7 m^2)$

$5n^2 \in O(n^2)$ $\quad$ $5n\lg n + 4 \in O(n\lg n)$

- Προφανώς, κατά την ανάλυση αλγορίθμων στόχος μας είναι αυτά τα όρια να είναι όσο το δυνατό πιο ακριβή.
- Εάν 7n-4 είναι O(n), τότε προφανώς 7n-4 είναι $O(n^2)$, $O(n^3)$, $O(n^4)$, ….
- Ωστόσο, οι χαρακτηρισμοί big-O πρέπει να είναι όσο το δυνατό πιο μικρής τάξης. (δηλαδή το O(n) είναι το πιο στενό άνω όριο).

Ασυμπτωτική Ανάλυση

Ασυμπτωτικά Μικρότερο (Asymptotically Smaller)

"<" σημαίνει: "είναι ασυμπτωτικά μικρότερο.

- Έστω ότι q(n) και p(n), δυο μη-αρνητικές συναρτήσεις.
- Το q(n) < p(n), εάν ισχύει: $\lim_{n->\infty} \frac{q(n)}{p(n)} = 0$

- Το p(n) ασυμπτωτικά επικρατεί (dominates) το q(n).

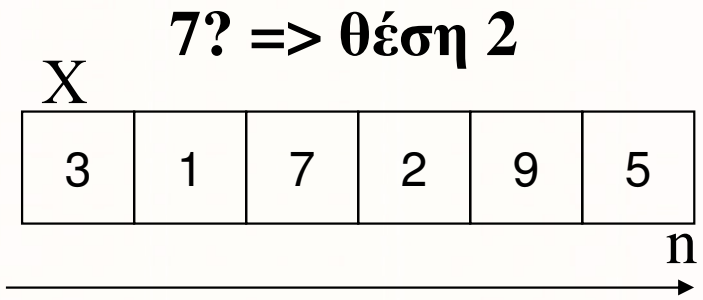
- Παράδειγμα:
- Αποδείξτε ότι ισχύει $10n+7 < 3n^2+2n+6$

$$\lim_{n->\infty} \frac{10n+7}{3n^2+2n+6} = \frac{\frac{10}{n}+\frac{7}{n^2}}{3+\frac{2}{n}+\frac{6}{n^2}} = \frac{0+0}{3+0+0} = \frac{0}{3} = 0$$

Παράδειγμα Ανάλυσης

Υποθέστε ότι θέλετε να βρείτε την θέση κάποιο στοιχείου K σε μια λίστα X[] (μήκους n).

```
int index( int X[],int n,int k){
1       int i=0;
2       while (i < n) {
3        if (X[i]==k) // item found
4              return i;
5         i+=1; }
6       return error;
}
```



Μέγεθος δεδομένων εισόδου: n

Χειρότερη Περίπτωση:

εξέταση όλων των στοιχείων (π.χ. ψάχνουμε το 5) $t(n) = n$ => O(n)

Βέλτιστη Περίπτωση:

το στοιχείο βρίσκετε στην θέση 1 (π.χ. ψάχνουμε το 3) $t(n) = 1$ => Ω(1)

Άρα εκτός από το **μέγεθος εισόδου** και το **ίδιο το στιγμιότυπο εισόδου** παίζει κάποιο σημαντικό ρόλο στην ανάλυση κάποιου αλγορίθμου (η συνάρτησης)

Ανάλυση Χειρίστης Περίπτωσης

- Στις περισσότερες περιπτώσεις, μας ενδιαφέρει η **ανάλυση χείριστης περίπτωσης**, επειδή μας δίδει ένα **άνω φράγμα** στον **χρόνο εκτέλεσης** ενός αλγόριθμου.
- Αν $\mathbf{D_n}$ είναι το **σύνολο όλων των εισόδων** (**στιγμιότυπων***) μεγέθους **n**, και **t(I)** ο αριθμός **βασικών πράξεων** που εκτελούνται από τον αλγόριθμο για κάθε $\mathbf{I \in D_n}$ τότε ορίζουμε την ***:***

***Πολυπλοκότητα* Χειρίστης Περίπτωσης:**

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

- Δηλαδή, ο ορισμός δίνει ένα **άνω φράγμα (upper bound)** της **πολυπλοκότητας** του **αλγορίθμου**.

- * D_n ={ {3,1,7,2,9,5}, {1,3,7,2,9,5}, …., (6!=720 διατάξεις) }

Ανάλυση Μέσης Περίπτωσης

- Υποθέτουμε πως μπορούμε να αντιστοιχίσουμε μια πιθανότητα **p(I)** σε κάθε είσοδο **$I \in D_n$** .

  Ορίζουμε την ***πολυπλοκότητα <u>Μέσης</u> Περίπτωσης*** ως

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

t(I): αριθ. βασικών πράξεων

p(I): Πιθανότητα εμφάνισης στιγμιοτύπου

Παράδειγμα: Ψάχνουμε το 3 σε μια λίστα n (6 αριθμών)

Το 3 μπορεί να βρίσκεται σε μια από τις πιο κάτω θέσεις:

I_1 ={3,x,x,x,x,x} I_2 ={x,3,x,x,x,x} I_3 ={x,x,3,x,x,x}

I_4 ={x,x,x,3,x,x} I_5 ={x,x,x,x,3,x} I_6 ={x,x,x,x,x,3}

Υποθέτουμε ότι και οι **n=6** καταστάσεις έχουν πιθανότητα εμφάνισης **p(I)=1/n**

Πολυπλοκότητα Μέσης Περίπτωσης ➔

1/n * (1 πράξη) + 1/n * (2 πράξεις) + 1/n * (3 πράξεις)

1/n * (4 πράξεις) + 1/n * (5 πράξεις) + 1/n * (6 πράξεις)

$$A(n) = \frac{1}{n} \sum_{i \in D_n} i = \frac{1}{n} \left(\frac{n.(n+1)}{2} \right) = \frac{n+1}{2}$$

Παράδειγμα 1: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int k=0;
for (i=0; i<n; i++)
      k+=i; //k=k+i
```

Ανάλυση

- k+=i; // Η βασική πράξη η οποία χρειάζεται O(1) χρόνο
- Ο αλγόριθμος εκτελεί την βασική πράξη n φορές

$$\sum_{i \in n} 1 = n$$

- Ο αλγόριθμος έχει πολυπλοκότητα
- της τάξης O(n)
- Στην βέλτιστη περίπτωση χρειάζεται Ω(n) => O(n) & Ω(n) => Θ(n)

Επανάληψη Χρήσιμων Μαθηματικών Ορισμών

Ορισμός 1: $\log_x a = b$ **iff** $x^b = a$

Χρήσιμοι νόμοι λογάριθμων:

$\log ab = \log a + \log b$ $\quad\quad$ $\log a^b = b.\log a$

$\log a \div b = \log a - \log b$ $\quad\quad$ $\log_a b = (\log_c b) \div (\log_c a)$

$b^{\log_b x} = x$ $\quad\quad$ $\log a = 1$

$\log^2 n = (\log n)^2 = \log n * \log n$ $\quad\quad$ $\log n^2 = \log n + \log n = 2x\log n$

$\log_{10}$ (common) , $\log_e$ (ln - natural), **$\log_2$ (lg - binary)**

π.χ., $\log_2 2=1$ $\quad$ $\log_2 1=0$ $\quad\quad$ $\log_2 0$ δεν ορίζεται

Ορισμός 2: $\lfloor x \rfloor = \max\{a \mid a \le x, \text{int}(a)\}$ (floor)

π.χ., $\lceil x \rceil = \min\{a \mid a \ge x, \text{int}(a)\}$ (ceiling)

$$\left\lceil \frac{63}{11} \right\rceil = \lceil 5.72 \rceil = 6, \quad \lfloor 5.1634 \rfloor = 5$$

Ακολουθίες και Αθροίσματα

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

$$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

Άθροισμα Γεωμετρικής Προόδου

$$\sum_{i=0}^{\log_2 n} 2^i = n + n/2 + n/4 + \ldots + 2 + 1 = 2n - 1$$

Το οποίο προκύπτει αλλιώς ως: $\sum_{i=0}^{\log_2 n} 2^i = \frac{2^{\log_2 n+1}-1}{2-1} = 2^{\log_2 n} * 2^1 - 1 = 2n - 1$

Μαθηματική Επαγωγή

Στόχος

Να αποδειχθεί ότι η (μαθηματική) πρόταση Π(n) ισχύει για κάθε $n \geq 0$.

Μέθοδος

1. Επαληθεύουμε πως η Π ισχύει για n=0,
2. Υποθέτουμε πως η Π ισχύει για n=k και
3. Αποδεικνύουμε πως η Π ισχύει για n=k+1.

Παραλλαγές

- Αντί του 0, σε ορισμένες περιπτώσεις ενδιαφερόμαστε για $n \geq a$, όπου το a είναι κάποιος ακέραιος.
- Στο δεύτερο βήμα: Υποθέτουμε πως η Π ισχύει για $n \leq k$ και αποδεικνύουμε πως η Π ισχύει για n=k+1.

Λογική πίσω από την Μαθηματική Επαγωγή

Έστω ότι κάποιος ισχυρίζεται ότι ο ακόλουθος τύπος ισχύει πάντα

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

- Καλούμαστε να **αποδείξουμε** (δηλαδή να δείξουμε ότι για κάθε τιμή του X ισχύει) η να το **διαψεύσουμε** (ότι δηλ., υπάρχει κάποια τιμή του n που κάνει αναληθή τον ισχυρισμό.
- **Δοκιμή για n=1 :** (1*2)/2 = 1 το οποίο είναι το ίδιο με το 1!
- Δοκιμή για n=2 : (2*3)/2 = 3 το οποίο είναι το ίδιο με το 1+2!
- Δοκιμή για n=3 : (3*4)/2 = 6 το οποίο είναι το ίδιο με το 1+2+3!
- Δοκιμή για n=4 : (4*5)/2 = 10 το οποίο είναι το ίδιο με το 1+2+3+4!
- ….
- **Ας υποθέσουμε ότι ισχύει και για n=k**
- **Μπορούμε να δείξουμε ότι ισχύει και για n=k+1** με βάσει την προηγούμενη υπόθεση; Εάν ναι τότε το έχουμε αποδείξει. Εάν όχι τότε δεν το έχουμε διαψεύσει, απλά δεν έχουμε βρεί την απάντηση!

Παράδειγμα 1 – Τυπική Απόδειξη

Να αποδείξετε ότι $\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$

Απόδειξη:

Θα αποδείξουμε με τη μέθοδο της μαθηματικής επαγωγής την πρόταση

$\Pi(n) \equiv \sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$

1. Επαληθεύουμε την Π(0) ως εξής: $\sum_{i=0}^{0} i = 0 = \frac{0 \cdot (0+1)}{2}$

2. Υποθέτουμε ότι ισχύει η Π(k), δηλαδή ότι $\sum_{i=0}^{k} i = \frac{k \cdot (k+1)}{2}$

3. Και θα αποδείξουμε ότι ισχύει η Π(k+1):

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + (k+1) = \frac{k \cdot (k+1)}{2} + (k+1)$$

$$= \frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

Επαγωγική υπόθεση

Παράδειγμα 2 – Τυπική Απόδειξη

Να αποδείξετε ότι $\sum_{i=0}^{n} i^3 = \frac{n^2 \cdot (n+1)^2}{4}$

Απόδειξη: Θα αποδείξουμε με τη μέθοδο της μαθηματικής επαγωγής την πρόταση Π(n) $\equiv \sum_{i=0}^{n} i^3 = \frac{n^2 \cdot (n+1)^2}{4}$

1. Επαληθεύουμε την Π(0) ως εξής: $\sum_{i=0}^{0} i^3 = 0 <=> \frac{0 \cdot (0+1)^2}{4} = 0$
2. Υποθέτουμε ότι ισχύει η Π(k), δηλαδή ότι $\sum_{i=0}^{k} i^3 = \frac{k^2 \cdot (k+1)^2}{4}$
3. Και θα αποδείξουμε ότι ισχύει η Π(k+1): $\sum_{i=0}^{k+1} i^3 = \frac{(k+1)^2 \cdot ((k+1)+1)^2}{4}$

$$\sum_{i=0}^{k+1} i^3 = \sum_{i=0}^{k} i^3 + (k+1)^3 = \frac{k^2 \cdot (k+1)^2}{4} + (k+1)^3$$

$$= \frac{k^2 \cdot (k+1)^2 + 4 \cdot (k+1)^3}{4} = \frac{(k+1)^2 \cdot (k^2 + 4 \cdot (k+1))}{4}$$

$$= \frac{(k+1)^2 \cdot (k^2 + 4 \cdot (k+1))}{4} = \frac{(k+1)^2 \cdot (k^2 + 4k + 4)}{4}$$

$$= \frac{(k+1)^2 \cdot (k+2)^2}{4} = \frac{(k+1)^2 \cdot ((k+1)+1)^2}{4}$$

Παράδειγμα 3 – Τυπική Απόδειξη

- Να αποδείξετε ότι $3^n < n!$ για n>=7 .

Απόδειξη:

Θα αποδείξουμε με τη μέθοδο της μαθηματικής επαγωγής την πρόταση

Π(n) ≡ $3^n < n!$ για n>=7 .

1. Επαληθεύουμε την Π(7) ως εξής: $3^7 = 2187 < 7! = 5040$
2. Υποθέτουμε ότι ισχύει η Π(k), δηλαδή ότι $3^k < k!$.
3. Και θα αποδείξουμε ότι ισχύει η Π(k+1): $3^{k+1} < (k+1)!$

$$
\begin{aligned}
3^{k+1} &= 3^k . 3 \\
&< 3.k! && \text{[Επαγωγική υπόθεση]} \\
&< (k+1).k! && \text{[Για k>7 ισχύει πάντα (k+1>3)]} \\
&< (k+1)! && \textbf{[Το οποίο είναι το ζητούμενο]}
\end{aligned}
$$