Δομές Δεδομένων και Αλγόριθμοι

Ασυμπτωτική Πολυπλοκότητα

Χρήστος Γκόγκος

Πανεπιστήμιο Ιωαννίνων, Τμήμα Πληροφορικής και Τηλεπικοινωνιών (2019-2020)

Τρεις εκδόσεις για έναν απλό αλγόριθμο αναζήτησης

Σειριακή αναζήτηση: Με είσοδο έναν πίνακα a με n μη διατεταγμένες τιμές ζητείται να απαντηθεί το εάν μια τιμή key υπάρχει στον πίνακα ή όχι.

Απλή σειριακή αναζήτηση

- Για κάθε αναζήτηση της τιμής key στον a θα πραγματοποιηθούν $n+1$ συγκρίσεις της μεταβλητής i με το n, λόγω της εντολής for, και n συγκρίσεις του $a[i]$ με το key.

```
#include <iostream>
using namespace std;

int linear_search(int *a, int n, int key)
{
    int position = -1;
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            position = i;
    return position;
}

int main()
{
    int a[] = {5, 7, 2, 1, 6, 3, 4, 0, 9};
    int n = sizeof(a) / sizeof(int);
    cout << "Found at: " << linear_search(a, n, 6) << endl;
    cout << "Found at: " << linear_search(a, n, 8) << endl;
}
```

Καλύτερη σειριακή αναζήτηση

- Εφόσον εντοπιστεί το key δεν απαιτείται η εξέταση των υπόλοιπων στοιχείων. Μέσω της εντολής return η συνάρτηση τερματίζει νωρίτερα την εκτέλεσή της.
- Αν το key δεν υπάρχει στον a τότε θα πραγματοποιηθούν $n+1$ συγκρίσεις της μεταβλητής i με το n, λόγω της εντολής for, και n συγκρίσεις του $a[i]$ με το key.

```
int better_linear_search(int *a, int n, int key)
{
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

Ακόμα καλύτερη σειριακή αναζήτηση

- Προκειμένου να αποφευχθούν οι συγκρίσεις του i με το n λόγω της εντολής for, αντικαθιστούμε το τελευταίο στοιχείο του πίνακα με την τιμή key.
- Αν βρεθεί το στοιχείο πριν τη τελευταία θέση η μεταβλητή i θα έχει τιμή μικρότερη από το $n-1$.
- Με αποθήκευση της τελευταίας τιμής του πίνακα πριν αντικατασταθεί από το key και επαναφορά της τιμής της όταν τελειώσει η επανάληψη καλύπτεται και η περίπτωση που το key βρίσκεται στη τελευταία θέση του πίνακα.

```
int sentinel_linear_search(int *a, int n, int key)
{
    int last = a[n - 1];
    a[n - 1] = key;
    int i = 0;
    while (a[i] != key)
        i++;
    a[n - 1] = last;
    if (i < n - 1 || a[n - 1] == key)
        return i;
    else
        return -1;
}
```

Η σημασία των αποδοτικών αλγορίθμων

- Η εξέλιξη στην ταχύτητα των υπολογιστών στις δεκαετίες που έχουν περάσει από τη δημιουργία των πρώτων υπολογιστών μέχρι σήμερα είναι εκπληκτική.
- Σήμερα (2019), ένας υπολογιστής μπορεί να εκτελέσει πάνω από 1 δισεκατομμύριο λειτουργίες όπως προσθέσεις, πολλαπλασιασμούς, μεταφορές τιμών κ.α. το δευτερόλεπτο.
- Ωστόσο, συχνά προκύπτουν προβλήματα στα οποία, καθώς το μέγεθος της εισόδου τους μεγαλώνει, γίνονται δυσανάλογα δυσκολότερα στην επίλυσή τους.

Πως μπορεί να μετρηθεί η αποδοτικότητα αλγορίθμων;

- Η προφανής ερώτηση "**Πόσο χρόνο θα χρειαστεί για να εκτελεστεί ένας αλγόριθμος;**" είναι δύσκολο να απαντηθεί (λόγω εξάρτησης από την είσοδο, την αρχιτεκτονική του υπολογιστή στον οποίο θα εκτελεστεί, τη γλώσσα προγραμματισμού, την επιδεξιότητα του προγραμματιστή κ.α.).
- Επιλέγουμε να αντικαταστήσουμε την ερώτηση με την "**Πως εξελίσσεται ο χρόνος εκτέλεσης καθώς το μέγεθος της εισόδου μεγαλώνει;**".

Παράδειγμα θεωρητικού προσδιορισμού χρόνου εκτέλεσης (σειριακή αναζήτηση) 1/2

```
int linear_search(int *a, int n, int key)
{
    int position = -1;  // 1
    for (int i = 0; i < n; i++)  // 2
        if (a[i] == key) // 3
            position = i; // 4
    return position; // 5
}
```

- Το βήμα 1 εκτελείται 1 φορά
- Στο βήμα 2:
  - A. γίνεται αρχικοποίηση της μεταβλητής i
  - B. γίνονται n+1 συγκρίσεις του i με το n
  - C. γίνεται n φορές μοναδιαία αύξηση του i
- Το βήμα 3 εκτελείται n φορές
- Το βήμα 4 μπορεί να εκτελεστεί από 0 μέχρι n φορές
- Το βήμα 5 εκτελείται 1 φορά
  κάτω όριο: $t_1 + t_{2A} + (n+1) * t_{2B} + n * t_{2C} + n * t_3 + 0 * t_4 + t_5$
  άνω όριο: $t_1 + t_{2A} + (n+1) * t_{2B} + n * t_{2C} + n * t_3 + n * t_4 + t_5$

Παράδειγμα θεωρητικού προσδιορισμού χρόνου εκτέλεσης (σειριακή αναζήτηση) 2/2

- κάτω όριο:
  - $t_1 + t_{2A} + (n+1) * t_{2B} + n * t_{2C} + n * t_3 + 0 * t_4 + t_5 \Rightarrow$
  - $t_1 + t_{2A} + t_{2B} + t_5 + n * t_{2B} + n * t_{2C} + n * t_3 \Rightarrow$
  - $(t_{2B} + t_{2C} + t_3) * n + (t_1 + t_{2A} + t_{2B} + t_5) \Rightarrow$
  - $c * n + d$
- άνω όριο:
  - $t_1 + t_{2A} + (n+1) * t_{2B} + n * t_{2C} + n * t_3 + n * t_4 + t_5 \Rightarrow$
  - $t_1 + t_{2A} + t_{2B} + t_5 + n * t_{2B} + n * t_{2C} + n * t_3 + n * t_4 \Rightarrow$
  - $(t_{2B} + t_{2C} + t_3 + t_4) * n + (t_1 + t_{2A} + t_{2B} + t_5) \Rightarrow$
  - $c' * n + d$

Πολυπλοκότητα αλγορίθμων

- Η πολυπλοκότητα ενός αλγορίθμου εκφράζεται ως συνάρτηση της **διάστασης του προβλήματος** που αντιμετωπίζεται. Με τον όρο διάσταση του προβλήματος αναφερόμαστε στο πλήθος των ατομικών δεδομένων που υποβάλλονται για επεξεργασία.
- Για παράδειγμα:
  - Στο πρόβλημα της σειριακής αναζήτησης που αναφέρθηκε, διάσταση του προβλήματος είναι το μέγεθος n του πίνακα.
  - Στο πρόβλημα εντοπισμού των πρώτων παραγόντων ενός μεγάλου ακέραιου αριθμού, διάσταση του προβλήματος είναι το πλήθος d των ψηφίων του αριθμού.
  - Στο πρόβλημα της άθροισης τιμών, διάσταση του προβλήματος είναι το πλήθος n των τιμών που πρόκειται να αθροιστούν.
  - Στο πρόβλημα του εντοπισμού της συντομότερης διαδρομής από μια αφετηρία προς έναν προορισμό σε έναν χάρτη, διάσταση του προβλήματος είναι ο συνδυασμός του πλήθους των κόμβων $|V|$ και του πλήθους των ακμών $|E|$ του γραφήματος που αναπαριστά αφαιρετικά το χάρτη.

Χειρότερη, καλύτερη και μέση περίπτωση

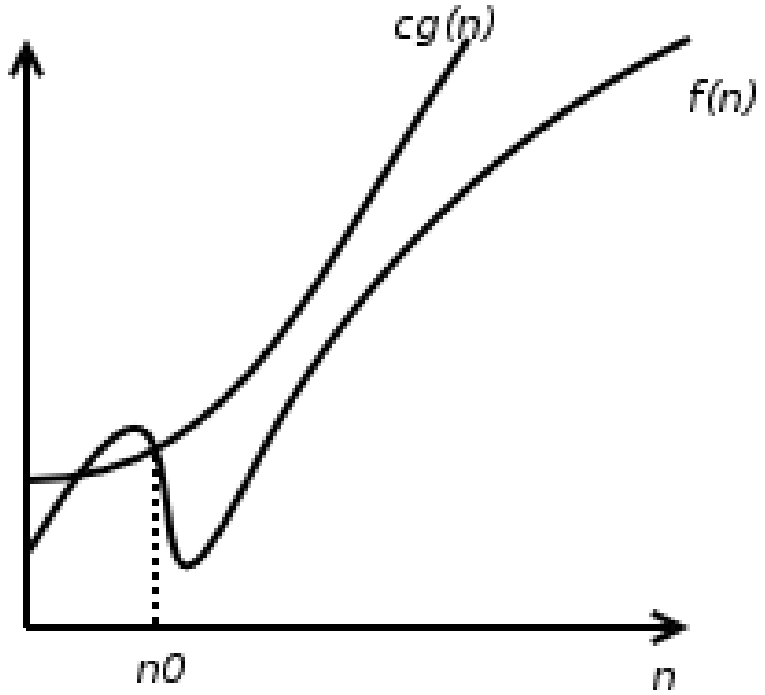
- **Χρόνος εκτέλεσης χειρότερης περίπτωσης**: Ο μεγαλύτερος χρόνος εκτέλεσης που μπορεί να παρατηρηθεί για οποιαδήποτε είσοδο διάστασης n (αποτελεί άνω όριο του χρόνου εκτέλεσης για οποιαδήποτε είσοδο).
- **Χρόνος εκτέλεσης καλύτερης περίπτωσης**: Ο συντομότερος χρόνος εκτέλεσης που μπορεί να παρατηρηθεί για οποιαδήποτε είσοδο διάστασης n (αποτελεί κάτω όριο του χρόνου εκτέλεσης για οποιαδήποτε είσοδο).
- **Χρόνος εκτέλεσης μέσης περίπτωσης**: Η μέση αναμενόμενη επίδοση λαμβάνοντας υπόψη όλες τις πιθανές εισόδους διάστασης n (γενικά είναι καλύτερος από το χρόνο χειρότερης περίπτωσης, αλλά μερικές φορές είναι περίπου το ίδιο "κακός χρόνος" με το χρόνο εκτέλεσης χειρότερης περίπτωσης).

Τάξη ανάπτυξης ή ρυθμός ανάπτυξης (order of growth) και ασυμπτωτική ανάλυση

- Για είσοδο διάστασης n, αυτό που έχει σημασία είναι η τάξη ανάπτυξης του χρόνου εκτέλεσης ασυμπτωτικά (καθώς το n γίνεται πολύ μεγάλο).
- Πραγματοποιείται:
  - Παράβλεψη των όρων χαμηλότερης τάξης καθώς είναι σχετικά ασήμαντοι για πολύ μεγάλο n. Για παράδειγμα $n^2 + n + \sqrt{n} + \log(n) \Rightarrow n^2$.
  - Παράβλεψη του συντελεστή του κυρίαρχου όρου, καθώς η συμμετοχή του στο ρυθμό αύξησης δεν είναι εξίσου σημαντική με τον κυρίαρχο όρο. Για παράδειγμα $3n^2 \Rightarrow n^2$.
- Η μελέτη του χρόνου εκτέλεσης (ή των απαιτήσεων μνήμης) καθώς αυξάνεται η διάσταση του προβλήματος προσεγγίζοντας πολύ μεγάλες τιμές ονομάζεται **ασυμπτωτική ανάλυση**.
- Υπάρχουν 3 βασικοί συμβολισμοί οι $\Theta, \mathrm{O}, \Omega$ καθώς και οι συμβολισμοί o, ω.

Ο συμβολισμός του μεγάλου Ο

Ο συμβολισμός του μεγάλου Ο αφορά το χαρακτηρισμό της χειρότερης περίπτωσης. Τοποθετεί ένα άνω όριο στο χρόνο εκτέλεσης ενός αλγορίθμου.

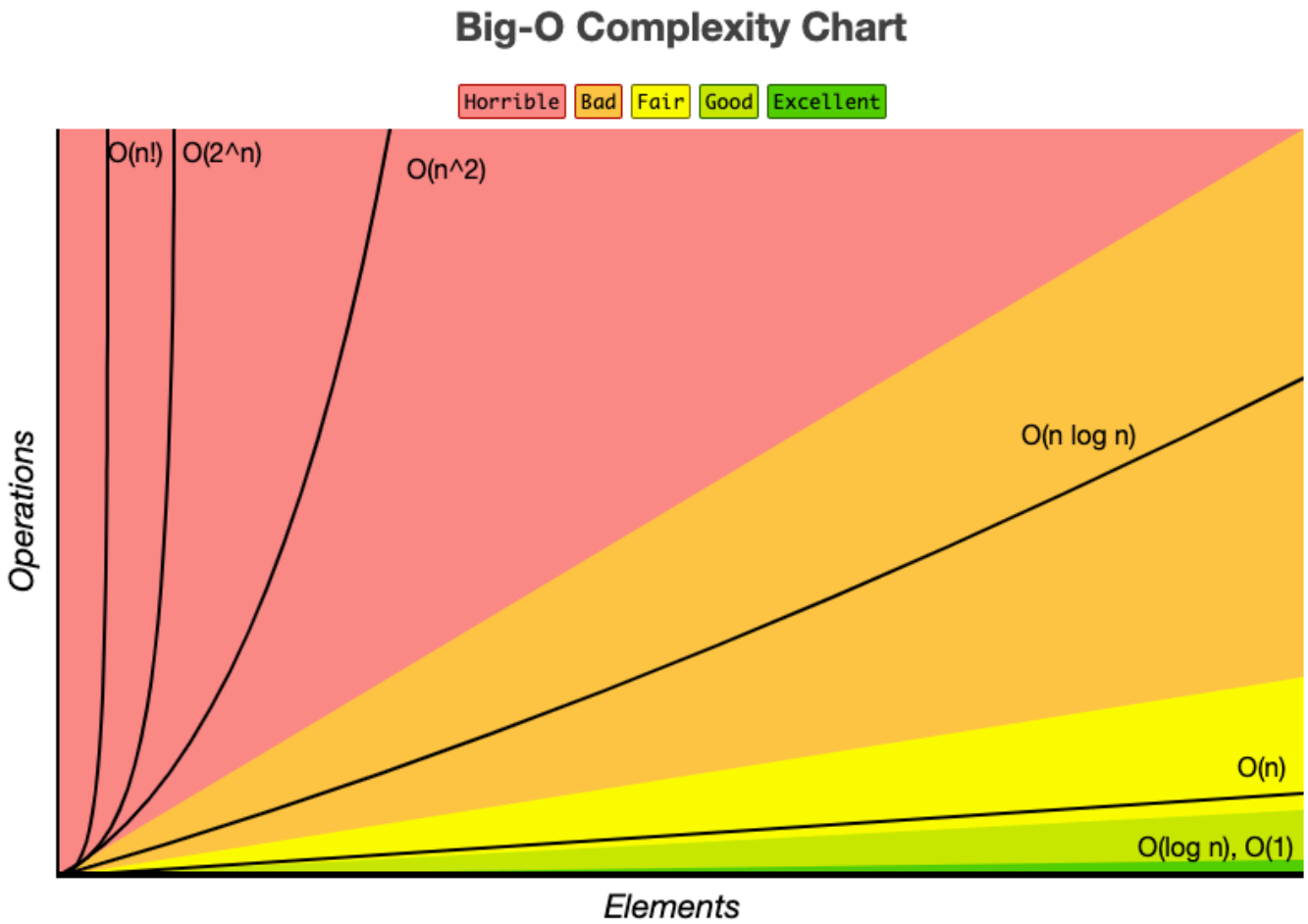


$f(n) = \mathrm{O}(g(n))$ σημαίνει ότι $\exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$

Δηλαδή, η συνάρτηση $f(n)$ είναι $\mathrm{O}(g(n))$ αν υπάρχουν θετικές σταθερές c, n_0 τέτοιες ώστε $0 \leq f(n) \leq cg(n)$ για κάθε $n \geq n_0$ (ή αλλιώς ένα πολλαπλάσιο του $g(n)$ είναι ασυμπτωτικό πάνω όριο για την $f(n)$).

Μεγάλο O (μικρότερο προς μεγαλύτερο)

$$\mathrm{O}(1) \ll \mathrm{O}(\log(n)) \ll \mathrm{O}(n) \ll \mathrm{O}(n\log(n)) \ll \mathrm{O}(n^2) \ll \mathrm{O}(n^3) \ll \mathrm{O}(2^n) \ll \mathrm{O}(n!)$$



<https://www.bigocheatsheet.com/>

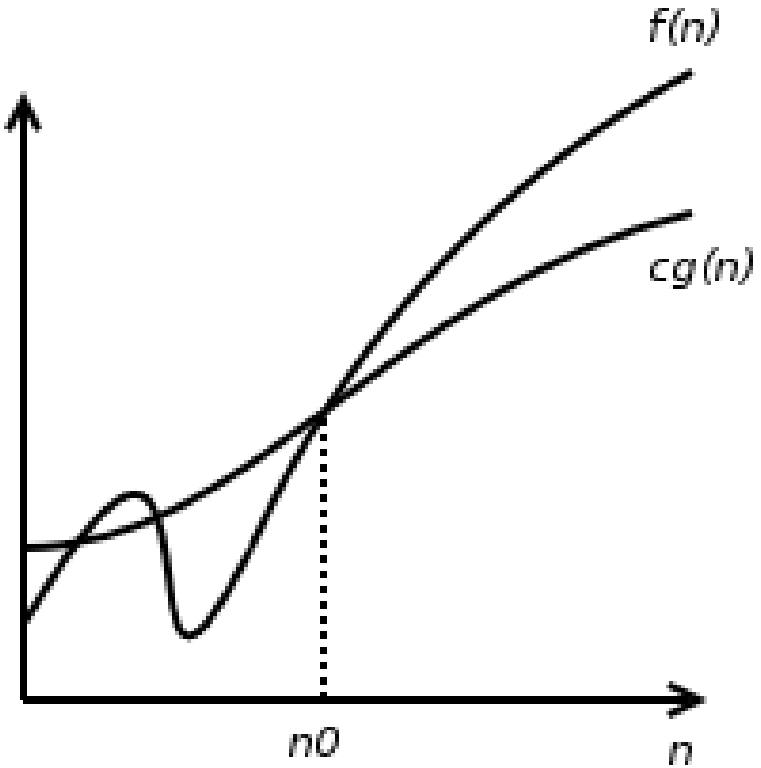
Παραδείγματα μεγάλου O

Ο συμβολισμός του μεγάλου O είναι ένας μαθηματικά τυπικός τρόπος σύμφωνα με τον οποίο προκειμένου να χαρακτηριστεί μια συνάρτηση δεν λαμβάνονται υπόψη σταθεροί παράγοντες και όροι χαμηλότερης τάξης παρά μόνο εξετάζεται το "σχήμα" της συνάρτησης.

- Ασκήσεις
  - Δείξτε ότι η συνάρτηση $f(n) = 2n + 10$ είναι $\mathrm{O}(n)$.
  - Δείξτε ότι η συνάρτηση $f(n) = 2n + 10$ είναι $\mathrm{O}(n^2)$.
  - Δείξτε ότι η συνάρτηση $f(n) = n^2$ δεν είναι $\mathrm{O}(n)$.
  - Δείξτε ότι η συνάρτηση $f(n) = 2^{n+5}$ είναι $\mathrm{O}(2^n)$.
  - Δείξτε ότι η συνάρτηση $f(n) = 2^{5n}$ δεν είναι $\mathrm{O}(2^n)$.

Ο συμβολισμός Ω

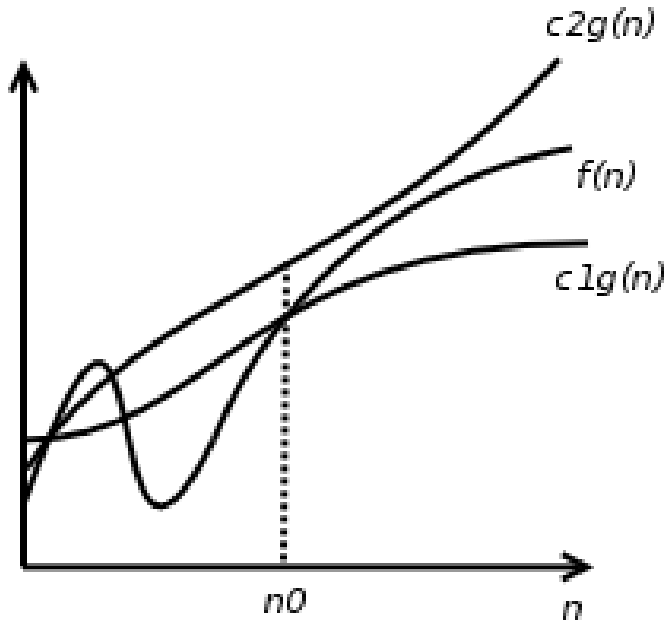
Ο συμβολισμός του μεγάλου Ω αφορά το χαρακτηρισμό της καλύτερης περίπτωσης. Τοποθετεί ένα κάτω όριο στο χρόνο εκτέλεσης ενός αλγορίθμου.



$f(n) = \Omega(g(n))$ σημαίνει ότι $\exists c, n_0 \geq 0 : 0 \leq cg(n) \leq f(n) \forall n \geq n_0$

Ο συμβολισμός Θ

Ο συμβολισμός του μεγάλου Θ αφορά τον ταυτόχρονο προσδιορισμό του χρόνου εκτέλεσης με κάτω και άνω όρια. Σημαίνει ότι καθώς $n \to \infty$ ο χρόνος εκτέλεσης $f(n)$ είναι το πολύ $c_2 g(n)$ και τουλάχιστον $c_1 g(n)$ για κάποιες σταθερές c_1 και c_2.



$f(n) = \Theta(g(n))$ σημαίνει ότι
$\exists c_1, c_2, n_0 > 0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$

Ισχύει ότι $f(n) = \Theta(g(n))$ αν και μόνο αν $f(n) = \mathrm{O}(g(n))$ και $f(n) = \Omega(g(n))$.

Παραδείγματα με μεγάλο Θ

- Ασκήσεις
  - Δείξτε ότι η συνάρτηση $f(n) = 6n^3$ δεν είναι $\Theta(n^2)$.

Ο συμβολισμός του μικρού o

Ο συμβολισμός του μικρού o αφορά το χαρακτηρισμό της χειρότερης περίπτωσης.

$f(n) = o(g(n))$ σημαίνει ότι $\forall c > 0, \exists n_0 > 0 : 0 \leq f(n) < cg(n) \quad \forall n \geq n_0$

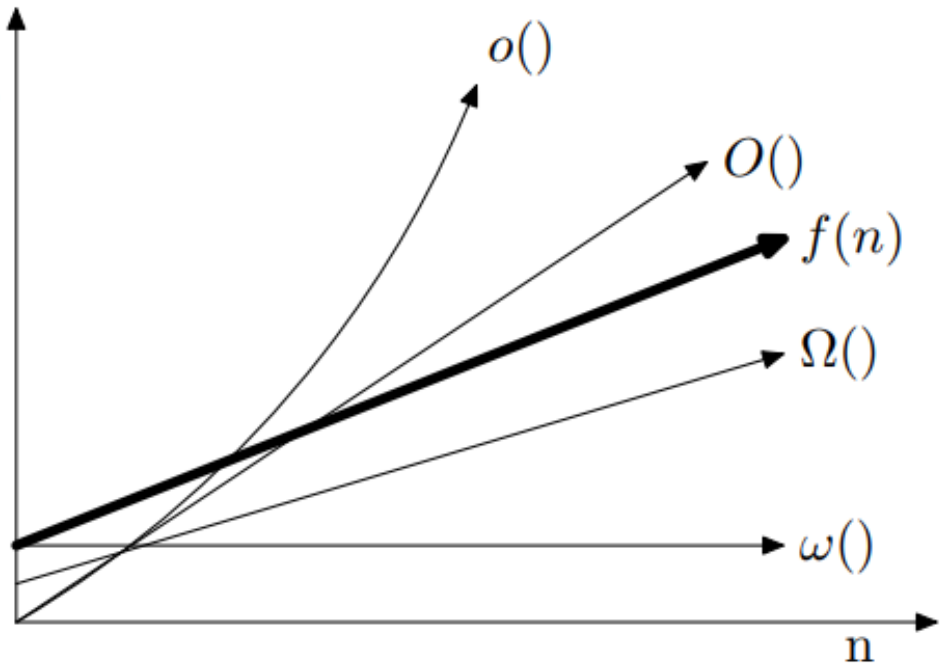
Δηλαδή, η συνάρτηση $f(n)$ είναι ασυμπτωτικά ασήμαντη σε σχέση με ένα πολλαπλάσιο της $g(n)$.

Ο συμβολισμός του μικρού ω

Ο συμβολισμός του μικρού ω αφορά το χαρακτηρισμό της καλύτερης περίπτωσης.

$f(n) = \omega(g(n))$ σημαίνει ότι $\forall c > 0, \exists n_0 > 0 : 0 \leq cg(n) < f(n) \quad \forall n \geq n_0$

Σχέση μεταξύ συμβολισμών O, Ω, o, ω



- Το μεγάλο O περιγράφει ένα σφικτό άνω όριο (αν και μπορεί να είναι και χαλαρό).
- Το μικρό o περιγράφει ένα άνω όριο που δεν μπορεί να είναι σφικτό.
- Το μεγάλο Ω περιγράφει ένα σφικτό κάτω όριο (αν και μπορεί να είναι και χαλαρό).
- Το μικρό ω περιγράφει ένα κάτω όριο που δεν μπορεί να είναι σφικτό.

Ιδιότητες ασυμπτωτικών συμβολισμών (1/2)

- Αν $\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \neq 0$ τότε $f(n) = \Theta(g(n))$
- Αν $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ τότε $f(n) = o(g(n))$
- Αν $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ τότε $f(n) = \omega(g(n))$

Ιδιότητες ασυμπτωτικών συμβολισμών (2/2)

- Μεταβατικότητα (transitivity)
  - Αν $f(n) = \Theta(g(n))$ και $g(n) = \Theta(h(n))$ τότε $f(n) = \Theta(h(n))$
  - Αν $f(n) = \mathrm{O}(g(n))$ και $g(n) = \mathrm{O}(h(n))$ τότε $f(n) = \mathrm{O}(h(n))$
  - Αν $f(n) = \Omega(g(n))$ και $g(n) = \Omega(h(n))$ τότε $f(n) = \Omega(h(n))$
  - Αν $f(n) = o(g(n))$ και $g(n) = o(h(n))$ τότε $f(n) = o(h(n))$
  - Αν $f(n) = \omega(g(n))$ και $g(n) = \omega(h(n))$ τότε $f(n) = \omega(h(n))$
- Συμμετρία (symmetry)
  - $f(n) = \Theta(g(n))$ αν και μόνο αν $g(n) = \Theta(f(n))$
- Ανάστροφη συμμετρία (transpose symmetry)
  - $f(n) = \mathrm{O}(g(n))$ αν και μόνο αν $g(n) = \Omega(f(n))$
  - $f(n) = o(g(n))$ αν και μόνο αν $g(n) = \omega(f(n))$
- Πράξεις
  - $\mathrm{O}(g_1(n)) + \mathrm{O}(g_2(n)) = \mathrm{O}(max(g_1(n), g_2(n)))$
  - $\mathrm{O}(g_1(n)) * \mathrm{O}(g_2(n)) = \mathrm{O}(g_1(n) * g_2(n))$

Πολυωνυμικοί αλγόριθμοι

Πολυωνυμικοί αλγόριθμοι είναι οι αλγόριθμοι που έχουν πολυωνυμικό χρόνο εκτέλεσης δηλαδή έχουν πολυπλοκότητα $\mathrm{O}(n^k)$ *όπου* $k > 0$ *είναι μια σταθερά.*

Οι πολυωνυμικοί αλγόριθμοι ορίζουν μια κλάση προβλημάτων που ονομάζεται P δηλαδή προβλήματα γα τα οποία η επίλυσή τους μπορεί να δοθεί από αλγορίθμους με πολυωνυμικό χρόνο εκτέλεσης.

Σταθερός χρόνος

Στο σταθερό χρόνο, ο χρόνος εκτέλεσης είναι $\mathrm{O}(1)$*. Δηλαδή, ο χρόνος εκτέλεσης είναι φραγμένος από πάνω από μια σταθερά που δεν εξαρτάται από τη διάσταση του προβλήματος* n*.*

- Παραδείγματα:
  - Διακλάδωση υπό συνθήκη
  - Αριθμητική ή λογική πράξη
  - Αρχικοποίηση μιας μεταβλητής
  - Πρόσβαση στο στοιχείο i ενός πίνακα
  - Σύγκριση/ανταλλαγή δύο στοιχείων σε έναν πίνακα

```
bool is_even(int x){ return (x%2==0); }
```

Γραμμικός χρόνος

Στο γραμμικό χρόνο, ο χρόνος εκτέλεσης είναι $O(n)$.

Παράδειγμα $O(n)$ αλγορίθμου. Συγχώνευση δύο διατεταγμένων ακολουθιών (std::vector).

```
vector<int> a{1, 4, 5, 7, 8};
vector<int> b{2, 6, 9, 10};
vector<int> c; int i = 0, j = 0;
while (i < a.size() && j < b.size())
    if (a[i] <= b[j])
    {
        c.push_back(a[i]);i++;
    }
    else
    {
        c.push_back(b[j]);j++;
    }
if (i == a.size())
    c.insert(end(c), begin(b) + j, end(b));
else
    c.insert(end(c), begin(a) + i, end(b));
```

Λογαριθμικός χρόνος

Στο λογαριθμικό χρόνο ο χρόνος εκτέλεσης είναι $O(logn)$.

Παράδειγμα $O(logn)$ χρόνου: Δίνεται ένας διατεταγμένος πίνακας με n διακριτούς ακεραίους αριθμούς και ένας ακέραιος x και ζητείται να βρεθεί ο δείκτης του x στον πίνακα.

```
int binary_search(int *a, int lo, int hi, int x)
{
    while (lo <= hi)
    {
        int mid = (lo + hi) / 2;
        if (x < a[mid])
            hi = mid - 1;
        else if (x > a[mid])
            lo = mid + 1;
        else
            return mid;
    }
    return -1;
}
```

Στο τέλος κάθε επανάληψης η διάσταση του προβλήματος μειώνεται στο μισό σε σχέση με την αρχή της επανάληψης.

Γραμμολογαριθμικός χρόνος

Στο γραμμολογαριθμικό χρόνο ο χρόνος εκτέλεσης είναι $O(nlogn)$.

Παράδειγματα $O(nlogn)$ χρόνου είναι οι αποδοτικοί αλγόριθμοι ταξινόμησης με **σύγκριση** όπως η ταξινόμηση με συγχώνευση (mergesort), η γρήγορη ταξινόμηση (quicksort), η ταξινόμηση σωρού (heapsort) κ.α.

Τετραγωνικός χρόνος

Στον τετραγωνικό χρόνο ο χρόνος εκτέλεσης είναι $O(n^2)$.

Παράδειγμα $O(n^2)$ χρόνου: Δίνεται μια λίστα n σημείων του καρτεσιανού επιπέδου $(x_1, y_1), ..., (x_n, y_n)$ και ζητείται να βρεθεί το ζεύγος σημείων που είναι πλησιέστερα το ένα στο άλλο.

```
int x[10] = {3, 5, 6, 2, 1, 9, 7, 3, 3, 4};
int y[10] = {2, 5, 7, 1, 8, 9, 2, 1, 0, 3};
int min = INT_MAX;
for (int i = 0; i < 10; i++)
    for (int j = i + 1; j < 10; j++)
    {
        int d = (x[i] - x[j]) * (x[i] - x[j]) +
                (y[i] - y[j]) * (y[i] - y[j]);
        if (d < min)
            min = d;
    }
```

Τα σημεία (3,3) και (2,1) είναι πλησιέστερα το ένα με το άλλο.

Κυβικός χρόνος

Στον κυβικό χρόνο ο χρόνος εκτέλεσης είναι $O(n^3)$.

Παράδειγμα $O(n^3)$ χρόνου: Δίνεται ένας πίνακας n διακριτών ακεραίων και ζητείται να βρεθούν όλες οι τριάδες τιμών με άθροισμα μηδέν.

```
int a[] = {-4, 1, 6, -2, 7, -5, -1, 9, 5, 2};
int n = sizeof(a) / sizeof(int);
for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
        for (int k = j + 1; k < n; k++)
            if (a[i] + a[j] + a[k] == 0)
                cout    << a[i] << " " << a[j]
                        << " " << a[k] << endl;
```

```
-4 6 -2
-4 -5 9
-4 -1 5
6 -5 -1
-2 7 -5
```

Εκθετικός χρόνος

Στον εκθετικό χρόνο ο χρόνος εκτέλεσης είναι $O(a^n)$ *με* $a > 1$.

Υπολογισμός του n-οστού αριθμού Fibonacci με αναδρομή.

```
int fibo(int n)
{
    if (n <= 1)
        return n;
    else
        return fibo(n - 2) + fibo(n - 1);
}
```

Αποδεικνύεται ότι ο παραπάνω κώδικας έχει πολυπλοκότητα $O(1.6^n)$.

Δυνάμεις

- $n^0 = 1$
- $n^1 = n$
- $n^{-1} = \frac{1}{n}$
- $n^a \cdot n^b = n^{a+b}$
- $\frac{n^a}{n^b} = n^{a-b}$
- $(n^a)^b = (n^b)^a = n^{ab}$

Λογάριθμοι (1/2)

- Ο λογάριθμος με βάση b ενός αριθμού a ισούται με την τιμή του εκθέτη x στην οποία θα πρέπει να υψωθεί το b έτσι ώστε να ληφθεί ως αποτέλεσμα το a. Δηλαδή: $\log_b a = x \Leftrightarrow b^x = a$.
- Το πεδίο ορισμού της λογαριθμικής συνάρτησης είναι το $(0, +\infty)$ και το πεδίο τιμών της είναι το $\mathbb{R}$.
- Η βάση b του λογαρίθμου $\log_b a$ μπορεί να λαμβάνει μόνο θετικές τιμές, διαφορετικές του 1. Η συμπεριφορά του λογαρίθμου είναι διαφορετική για $b > 1$ από ότι για $0 < b < 1$.
- Συνηθισμένοι λογάριθμοι:
  - $\log_{10} x$ δεκαδικός ή κοινός λογάριθμος
  - $\log_e x = \ln x$ φυσικός λογάριθμος, $e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2,71828...$
  - $\log_2 x = \lg x$ **δυαδικός λογάριθμος** (στην επιστήμη υπολογιστών οι λογάριθμοι, εφόσον δεν δίνεται διευκρίνηση, θεωρείται ότι είναι δυαδικοί)
- Παραδείγματα:
  - $\log_{10} 100 = 2$
  - $\log_2 8 = 3$
  - $\log_8 2 = \frac{1}{3}$
  - $\log_2 \frac{1}{8} = -3$
  - $\log_8 \frac{1}{2} = -\frac{1}{3}$

Βασικές ιδιότητες λογαρίθμων (2/2)

- Η συνάρτηση του λογαρίθμου είναι αντίστροφη της εκθετικής συνάρτησης.
- $\log_b 1 = 0$
- $\log_b b = 1$
- $\log_b a = \frac{1}{\log_a b}$ (εναλλαγή βάσης και ορίσματος)
- $b^{\log_b a} = a$
- $\log_b a^n = n \log_b a$
- $\log_b(x * y) = \log_b x + \log_b y$
- $\log_b(\frac{x}{y}) = \log_b x - \log_b y$
- $n^{\log_a b} = b^{\log_a n}$

Αλλαγή βάσης λογαρίθμου από b_2 *σε* b_1*:* $log_{b_1} a = \frac{\log_{b_2} a}{\log_{b_2} b_1}$*. Για παράδειγμα αν έχουμε μια υπολογιστική μηχανή που έχει τη δυνατότητα να υπολογίζει μόνο λογαρίθμους με βάση 10 μπορούμε να υπολογίσουμε το* $\log_2 1000$ *ως εξής:* $log_2 1000 = \frac{\log_{10} 1000}{\log_{10} 2} = \frac{3}{0.301} = 9.966$

Παραγοντικό (1/2)

Το παραγοντικό (factorial) ορίζεται αναδρομικά για μη αρνητικούς ακέραιους αριθμούς $n \geq 0$ ως εξής:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

- $n! = 1 * 2 * 3 * \cdots * n$
  - $5! = 120$
  - $10! = 3628800$
  - $100! = 9.332621544E + 157$
- $n! < n^n$ για $n \geq 2$
- Προσέγγιση του Stirling για το παραγοντικό : $n! = \sqrt{2\pi n}(\frac{n}{e})^n$

Παραγοντικό (2/2)

Μερικές χρήσεις του παραγοντικού:

- Με πόσους διαφορετικούς τρόπους μπορεί να διαταχθεί ένα σύνολο n αντικειμένων; Απάντηση: Υπάρχουν n επιλογές για το πρώτο αντικείμενο, $(n-1)$ επιλογές για το δεύτερο αντικείμενο, $(n-2)$ επιλογές για το τρίτο αντικείμενο κ.ο.κ. Άρα οι διαφορετικές διατάξεις είναι $n * (n-1) * (n-2) * \cdots * 1 = n!$.
  - Για παράδειγμα για 3 αντικείμενα A,B,C οι διατάξεις είναι οι εξής $1 * 2 * 3 = 6$: ABC, ACB, BAC, BCA, CAB, CBA.
- Με πόσους διαφορετικούς τρόπους μπορούν να επιλεγούν k αντικείμενα από ένα σύνολο n αντικειμένων; Απάντηση: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
  - Για παράδειγμα για 8 παίκτες, οι συνδυασμοί 5 παικτών που μπορούν να δημιουργηθούν είναι $\binom{8}{5} = \frac{8!}{5!3!} = 56$.

Αθροίσματα

- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1}-1}{x-1}$
  - $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
- $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ εάν $0 \leq x \leq 1$
  - $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$
  - $\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$
- $\sum_{i=1}^{n} a_i - a_{i-1} = a_n - a_0$
- $\sum_{i=1}^{n} a_i - a_{i+1} = a_0 - a_n$
- $H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \ln n$ αρμονικός αριθμός n