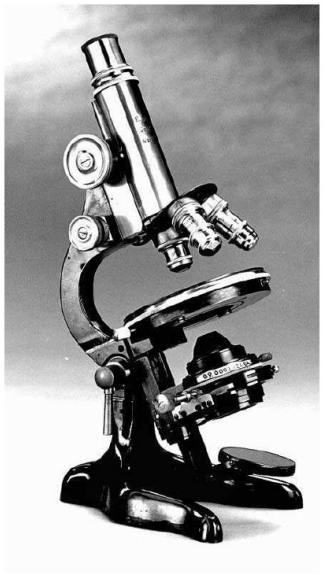
Παρουσίαση για χρήση με το σύγγραμμα, Αλγόριθμοι Σχεδίαση και Εφαρμογές, των M. T. Goodrich and R. Tamassia, Wiley, 2015 (στα ελληνικά από εκδόσεις Μ. Γκιούρδας)

Ανάλυση αλγορίθμων

Κλιμάκωση

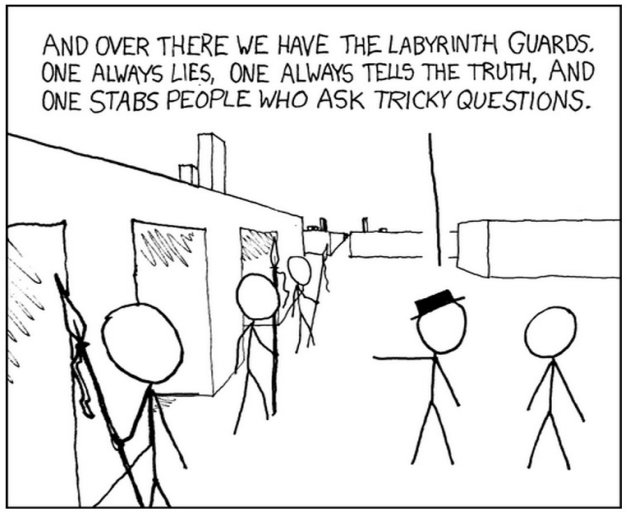
- Οι επιστήμονες συχνά αντιμετωπίζουν διάφορες κλίμακες, από μικροσκοπικές έως αστρονομικά μεγάλες.
- Οι επιστήμονες της πληροφορικής επίσης αντιμετωπίζουν διάφορες κλίμακες, αλλά αφορούν κυρίως το μέγεθος των δεδομένων πάρα τον όγκο φυσικών αντικειμένων.
- Η **κλιμάκωση** αφορά την ικανότητα ενός συστήματος να διαχειριστεί αυξανόμενο όγκο δεδομένων ή φόρτου εργασίας.



Microscope: U.S. government image, from the N.I.H. Medical Instrument Gallery, DeWitt Stetten, Jr., Museum of Medical Research. Hubble Space Telescope: U.S. government image, from NASA, STS-125 Crew, May 25, 2009.

Εφαρμογή: Συνεντεύξεις

- Η εταιρίες υψηλής τεχνολογίας συνηθίζουν να κάνουν ερωτήσεις για **αλγόριθμους και δομές δεδομένων** στις συνεντεύξεις.
- Οι ερωτήσεις για αλγορίθμους μπορεί να είναι σύντομες αλλά απαιτούν κριτική σκέψη, δημιουργικότητα και γνώση του αντικειμένου.
  - Οι ερωτήσεις στην ενότητα ερωτήσεων «Εφαρμογές» του κεφαλαίου 1 του βιβλίου των Goodrich-Tamassia είναι ερωτήσεις πραγματικών συνεντεύξεων για πρόσληψη εργαζόμενων.



xkcd "Labyrinth Puzzle." http://xkcd.com/246/
Used with permission under Creative Commons 2.5 License.

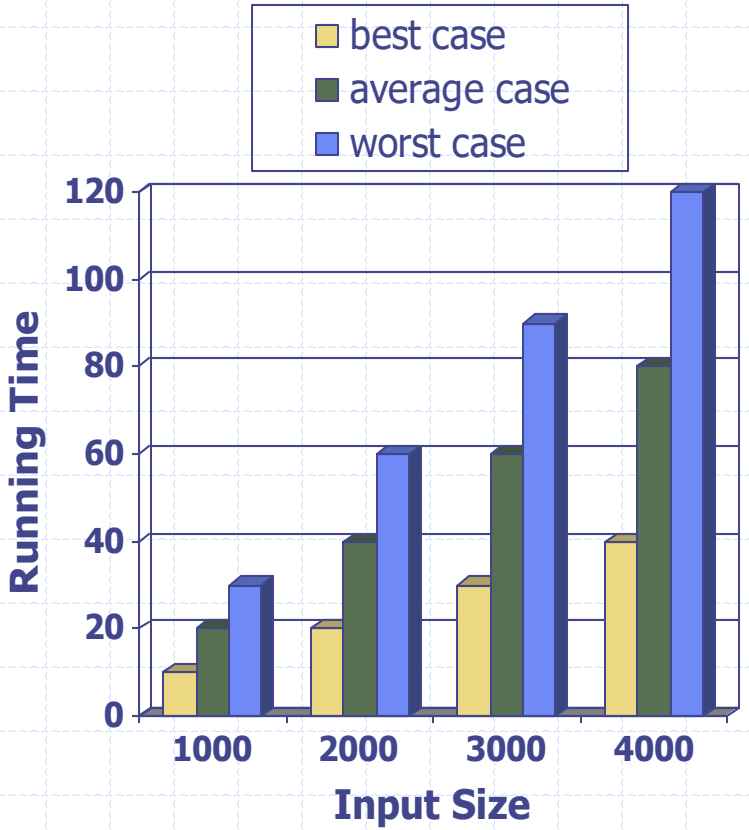
Αλγόριθμοι και δομές δεδομένων

- Ένας **αλγόριθμος** είναι μία διαδικασία βήμα προς βήμα για την επίτευξη κάποιας εργασίας σε πεπερασμένο χρονικό διάστημα.
  - Συνήθως, ένας αλγόριθμος δέχεται κάποιες εισόδους και βάση αυτών παράγει κάποια έξοδο.

- Μία **δομή δεδομένων** είναι ένας συστηματικός τρόπος οργάνωσης και πρόσβασης της πληροφορίας.

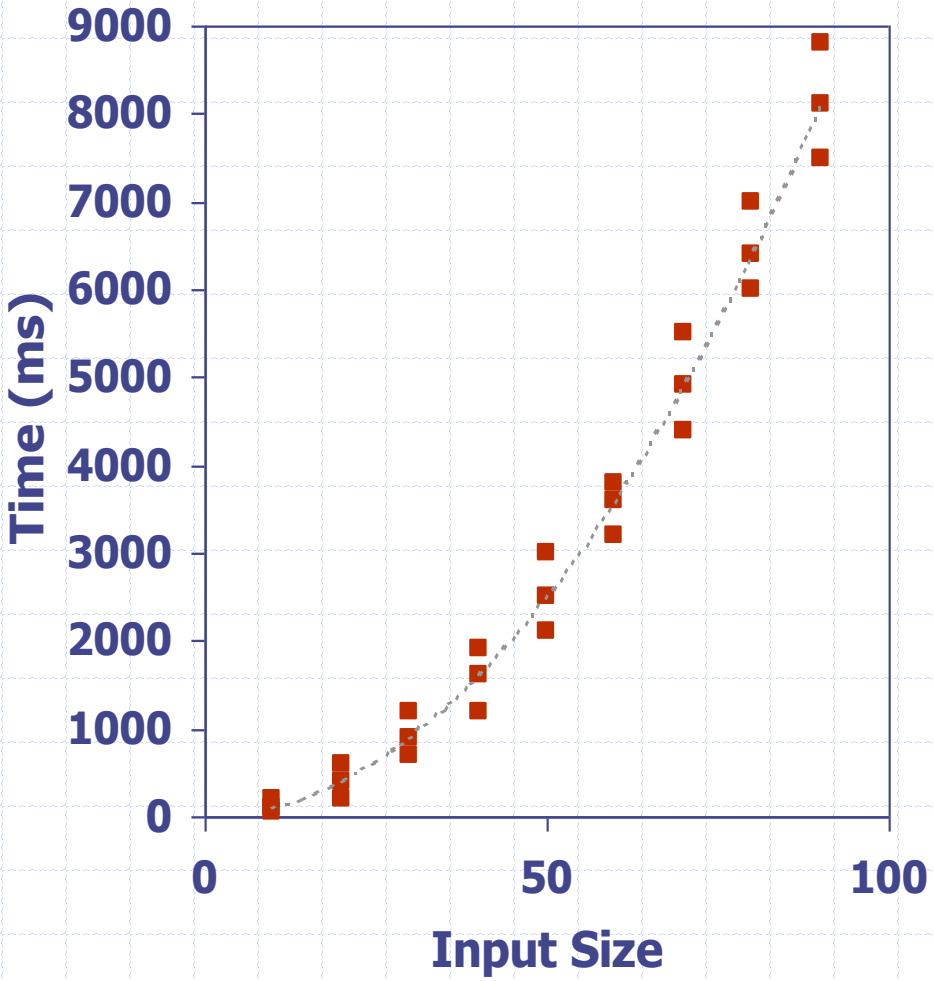
Χρόνος εκτέλεσης

- Οι περισσότεροι αλγόριθμοι μετατρέπουν εισόδους σε εξόδους.
- Ο χρόνος εκτέλεσης συνήθως αυξάνεται με το μέγεθος της εισόδου.
- Συχνά είναι δύσκολο να οριστεί κάποιος μέσος χρόνος.
- Κυρίως ασχολούμαστε με **τον χρόνο εκτέλεσης στην χειρότερη δυνατή περίπτωση**.
  - Πιο εύκολη ανάλυση.
  - Κρίσιμο για εφαρμογές σε παιχνίδια, οικονομικά και ρομποτική.



Πειραματικές μελέτες

- Υλοποιούμε τον αλγόριθμο.
- Εκτελούμε τον αλγόριθμο με εισόδους διαφόρων μεγεθών και περιεχομένων και σημειώνουμε το χρόνο που απαιτείται:
- Σχεδιάζουμε τη γραφική παράσταση.



Περιορισμοί των πειραμάτων

- Πρέπει να υλοποιηθεί ο αλγόριθμος, που μπορεί να είναι δύσκολο
- Τα αποτελέσματα μπορεί να μην είναι αντιπροσωπευτικά για άλλες εισόδους που δεν συμπεριλάβαμε στο πείραμα.
- Προκειμένου να συγκρίνουμε δύο αλγόριθμους πρέπει να έχουμε το ίδιο λογισμικό και υλικό.

Θεωρητική ανάλυση

- Χρειάζεται μία υψηλού επιπέδου περιγραφή του αλγορίθμου και όχι η υλοποίηση του
- Χαρακτηρίζει το χρόνο εκτέλεσης βάσει του μεγέθους της εισόδου, **n**
- Λαμβάνει υπόψιν όλες τις πιθανές εισόδους
- Μας επιτρέπει να υπολογίσουμε την ταχύτητα ενός αλγορίθμου ανεξαρτήτως περιβάλλοντος λογισμικού/υλικού.

Ψευδοκώδικας

- Υψηλού-επιπέδου περιγραφή ενός αλγορίθμου
- Περισσότερο δομημένος από πεζό κείμενο
- Λιγότερο λεπτομερής από κώδικα
- Προτιμάται για την περιγραφή αλγορίθμων
- Αποκρύπτει θέματα σχεδίασης ενός προγράμματος

Λεπτομέρειες ψευδοκώδικα

- Έλεγχος ροής
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Οι εσοχές αντικαθιστούν τα άγκιστρα
- Δήλωση μεθόδων

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Κλήση μεθόδων

  method (***arg*** [, ***arg***…])
- Επιστροφή τιμών

  return ***expression***
- Εκφράσεις:

  ← Ανάθεση

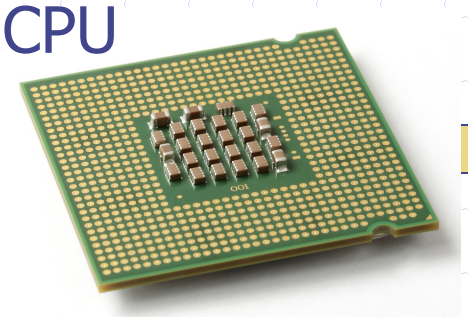
  = Σύγκριση ισότητας

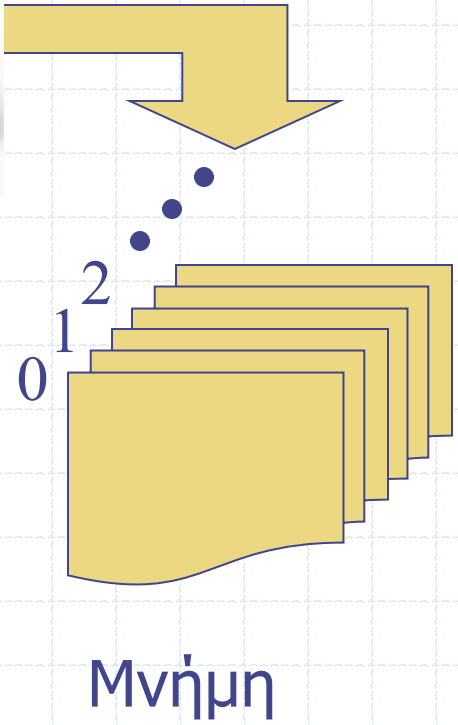
  n^2 Επιτρέπονται μαθηματικοί συμβολισμοί όπως η ύψωση σε δύναμη κ.α.

Το μοντέλο μηχανής τυχαίας προσπέλασης (Random Access Machine RAM)

Το μοντέλο **RAM** αποτελείται από:

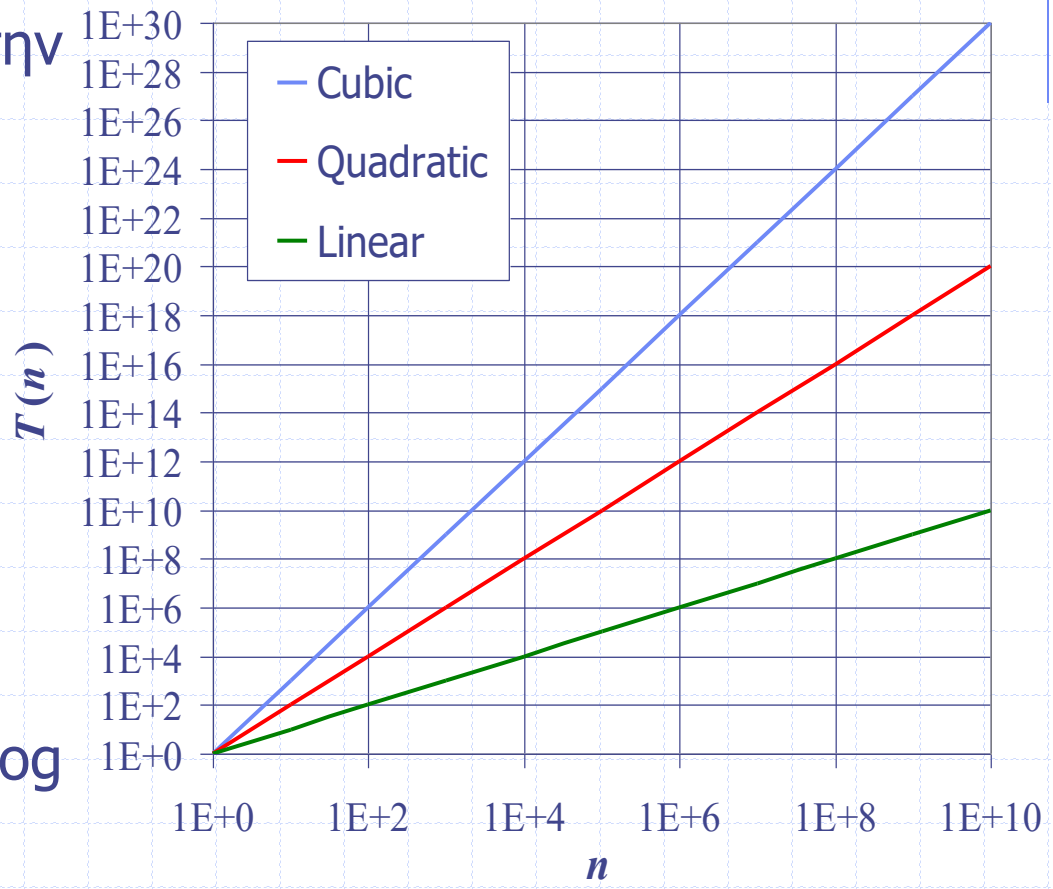
- Μία CPU
- Μια εν δυνάμει άπειρη συλλογή από κελιά μνήμης, με το καθένα από αυτά να μπορεί να περιέχει έναν αριθμό ή χαρακτήρα
- Τα κελιά μνήμης είναι αριθμημένα και η προσπέλαση οποιουδήποτε κελιού απαιτεί 1 μονάδα χρόνου





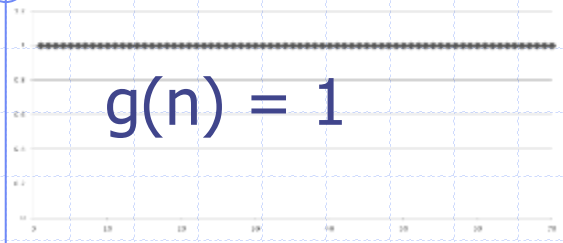
Επτά σημαντικές συναρτήσεις

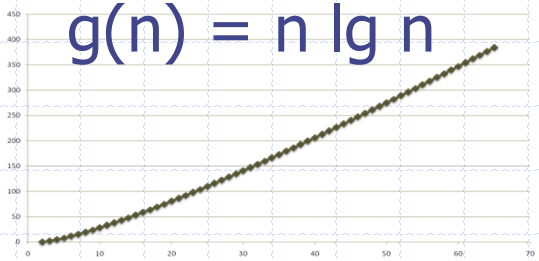
- Επτά συναρτήσεις που εμφανίζονται συχνά στην ανάλυση αλγορίθμων:
  1. Σταθερή ≈ 1
  2. Λογαριθμική ≈ log n
  3. Γραμμική ≈ n
  4. N-Log-N ≈ n log n
  5. Τετραγωνική ≈ n^2
  6. Κυβική ≈ n^3
  7. Εκθετική ≈ 2^n
- Σε ένα διάγραμμα log-log η κλίση δείχνει τον ρυθμό αύξησης.

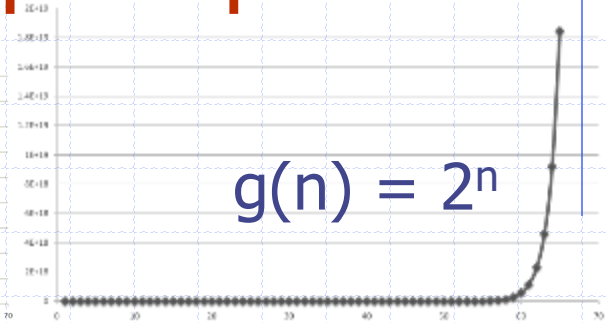


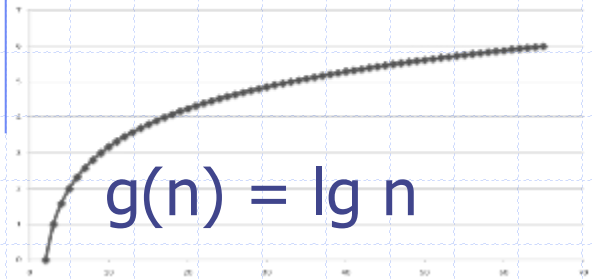
Συναρτήσεις σχεδιασμένες σε «κανονική» κλίμακα.

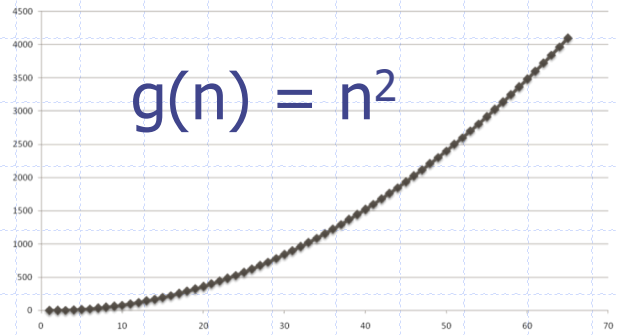
Slide by Matt Stallmann included with permission.

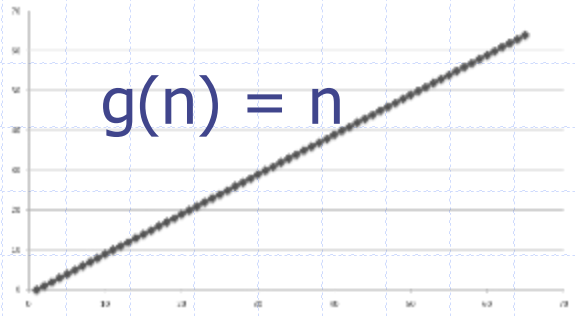


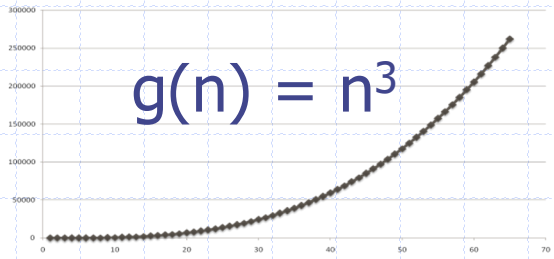












Πρωτογενείς λειτουργίες

- Βασικοί υπολογισμοί ενός αλγορίθμου
- Συναντώνται στο ψευδοκώδικα
- Σε μεγάλο βαθμό δεν εξαρτώνται από τη γλώσσα προγραμματισμού
- Ο ακριβής προσδιορισμός τους δεν είναι σημαντικός (θα δούμε γιατί αργότερα)
- Θεωρούμε ότι καταναλώνουν σταθερό χρόνο στο μοντέλο RAM

- Παραδείγματα:
  - Υπολογισμός μίας έκφρασης
  - Ανάθεση μίας τιμής σε μεταβλητή
  - Δεικτοδότηση πίνακα
  - Κλήση μίας μεθόδου
  - Επιστροφή από μία μέθοδο

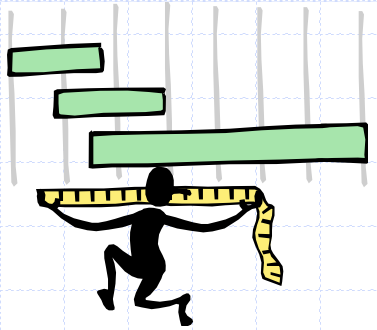
Μέτρηση βασικών βημάτων υπολογιστή

- Παράδειγμα: Εξετάζοντας τον ψευδοκώδικα μπορούμε να ορίσουμε το μέγιστο αριθμό βασικών πρωτογενών λειτουργιών, ως συνάρτηση του μεγέθους εισόδου

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

$currentMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $currentMax < A[i]$ **then**
    $currentMax \leftarrow A[i]$
return $currentMax$

Εκτίμηση χρόνου εκτέλεσης



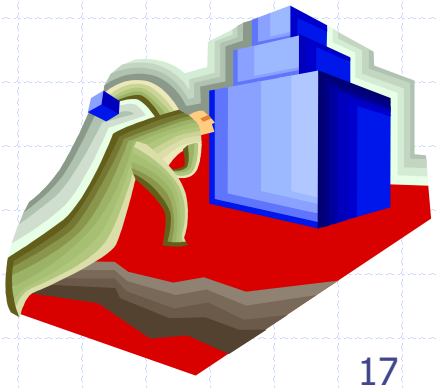
- Ο αλγόριθμος arrayMax εκτελεί $7n - 2$ πρωτογενών λειτουργιών στη χειρότερη περίπτωση, $5n$ στην καλύτερη περίπτωση. Ορίζουμε:
  - a = Ο χρόνος που χρειάζεται η ταχύτερη πρωτογενής λειτουργία
  - b = Ο χρόνος που χρειάζεται η αργότερη πρωτογενής λειτουργία
- Έστω $T(n)$ ο χρόνος εκτέλεσης στη χειρότερη περίπτωση του arrayMax. Τότε

$$a(5n) \leq T(n) \leq b(7n - 2)$$

- Έτσι, οριοθετούμε τον χρόνο εκτέλεσης $T(n)$ ανάμεσα σε δύο γραμμικές συναρτήσεις

Ρυθμός αύξησης του χρόνου εκτέλεσης

- Αλλάζοντας το λογισμικό/υλικό
  - Επηρεάζεται ο $T(n)$ με έναν σταθερό συντελεστή, αλλά
  - Δεν αλλάζει ο ρυθμός αύξησης του $T(n)$
- Ο γραμμικός ρυθμός αύξησης του $T(n)$ είναι μία εγγενής ιδιότητα του αλγόριθμου arrayMax



Slide by Matt Stallmann included with permission.

Γιατί έχει σημασία ο ρυθμός αύξησης

Εάν ο χρόνος εκτέλεσης είναι:	Χρόνος για n + 1	Χρόνος για 2 n	Χρόνος για 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

Ο χρόνος εκτέλεσης τετραπλασιάζεται όταν το μέγεθος του προβλήματος διπλασιάζεται

Αναλύοντας αναδρομικούς αλγορίθμους

- Βρείτε μία συνάρτηση, T(*n*), για να ορίσετε **μια σχέση αναδρομής** που χαρακτηρίζει τον χρόνο εκτέλεσης του αλγόριθμου σε σχέση με το *n*.

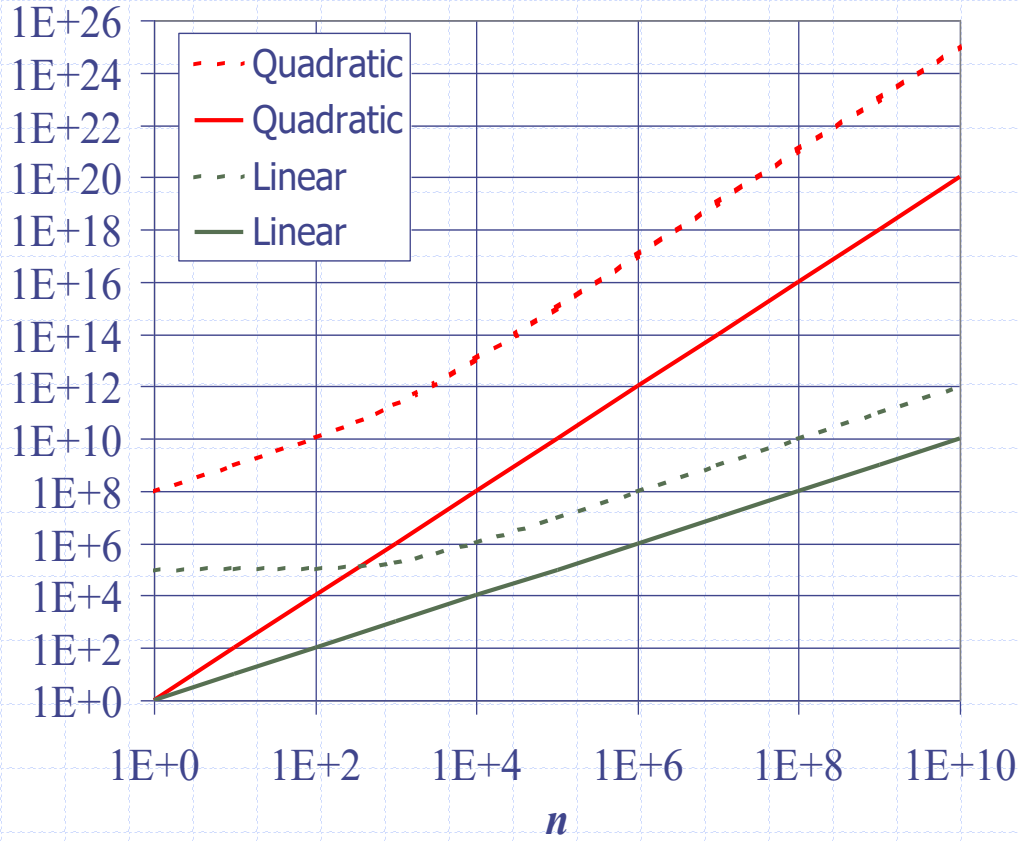
Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

if $n = 1$ **then**
return $A[0]$
return $\max\{$recursiveMax$(A, n-1),\ A[n-1]\}$

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise,} \end{cases}$$

Σταθεροί παράγοντες

- Ο ρυθμός αύξησης επηρεάζεται ελάχιστα από:
  - Σταθερούς παράγοντες ή
  - Όρους χαμηλότερης τάξης
- Παραδείγματα
  - Η $10^2n + 10^5$ είναι μία γραμμική συνάρτηση
  - Η $10^5n^2 + 10^8n$ είναι μία τετραγωνική συνάρτηση

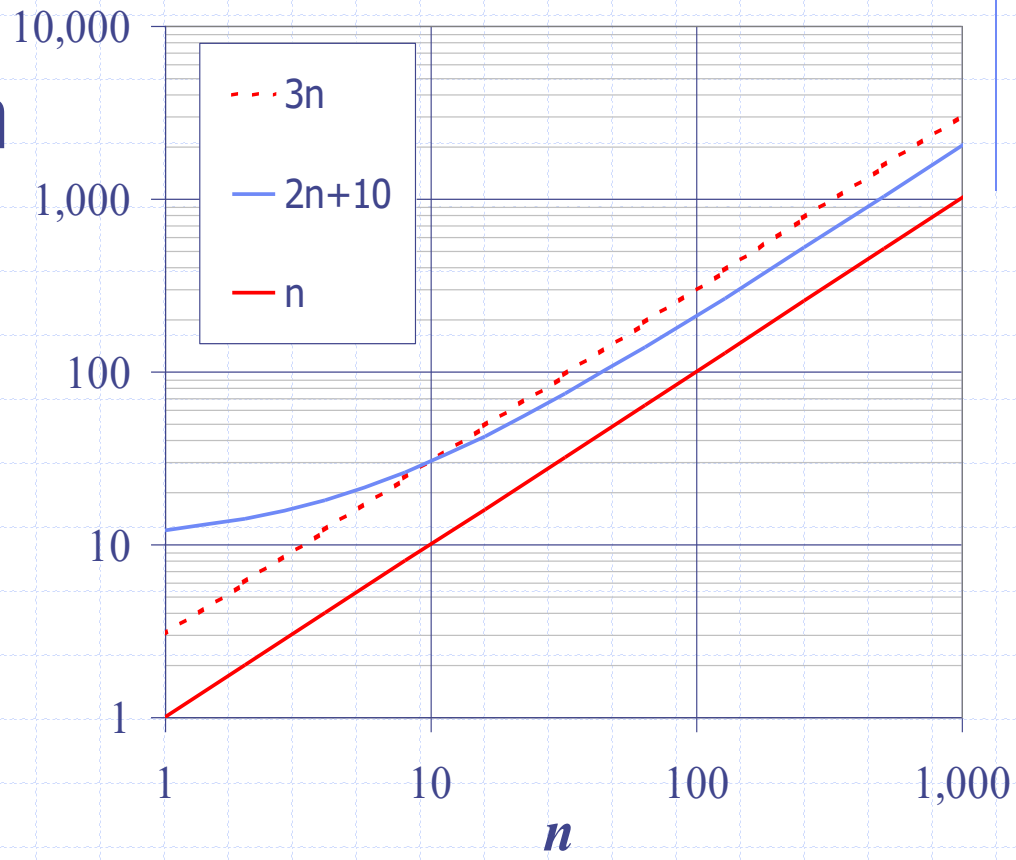


Ο συμβολισμός του μεγάλου Όμικρον (Big-Oh)

- Έστω οι συναρτήσεις $f(n)$ και $g(n)$, λέμε ότι η $f(n)$ είναι $O(g(n))$ εάν υπάρχουν θετικές σταθερές c και n_0 έτσι ώστε:

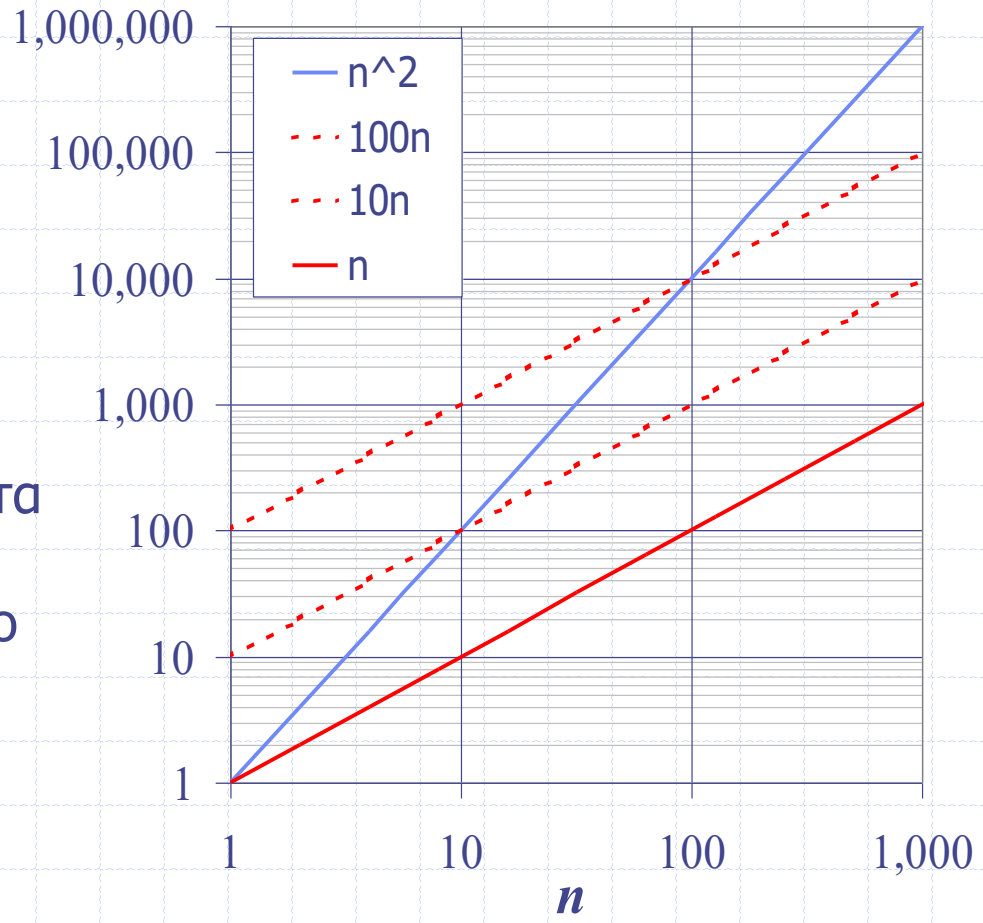
  $f(n) \leq cg(n)$ για $n \geq n_0$

- Παράδειγμα: $2n + 10$ είναι $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Επιλογή $c = 3$ και $n_0 = 10$

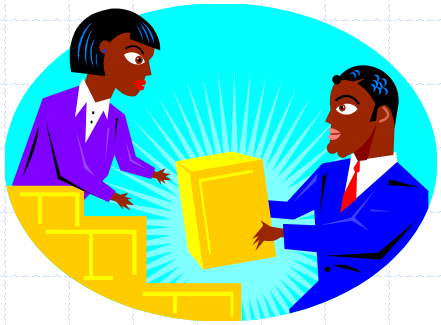


Big-Oh Παράδειγμα

- Παράδειγμα: η συνάρτηση n^2 ΔΕΝ είναι $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - Η παραπάνω ανισότητα δεν μπορεί να ικανοποιηθεί καθώς το c πρέπει να είναι σταθερά



Περισσότερα παραδείγματα Big-Oh



- $7n - 2$

  Η **7n-2** είναι O(n)

  Χρειαζόμαστε $c > 0$ και $n_0 \geq 1$ έτσι ώστε $7n - 2 \leq c\,n$ για $n \geq n_0$

  Ισχύει για $c = 7$ και $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  Η **$3n^3 + 20n^2 + 5$** είναι $O(n^3)$

  Χρειαζόμαστε $c > 0$ και $n_0 \geq 1$ έτσι ώστε $3n^3 + 20n^2 + 5 \leq c\,n^3$ για $n \geq n_0$

  Ισχύει για $c = 4$ και $n_0 = 21$

- $3 \log n + 5$

  Η **3 log n + 5** είναι O(log n)

  Χρειαζόμαστε $c > 0$ και $n_0 \geq 1$ έτσι ώστε $3 \log n + 5 \leq c \log n$ για $n \geq n_0$

  Ισχύει για $c = 8$ και $n_0 = 2$

Big-Oh και ρυθμός αύξησης

- Ο συμβολισμός του μεγάλου όμικρον (big-Oh notation) δίνει ένα άνω όριο για το ρυθμό αύξησης μίας συνάρτησης
- Η δήλωση "$f(n)$ είναι $O(g(n))$" σημαίνει ότι ο ρυθμός αύξησης της $f(n)$ δεν είναι μεγαλύτερος από το ρυθμό αύξησης της $g(n)$
- Μπορούμε να χρησιμοποιήσουμε το big-Oh notation για να ταξινομήσουμε συναρτήσεις βάσει του ρυθμού αύξησης τους

	$f(n)$ είναι $O(g(n))$	$g(n)$ είναι $O(f(n))$
$g(n)$ αυξάνεται περισσότερο	Ναι	Όχι
$f(n)$ αυξάνεται περισσότερο	Όχι	Ναι
Ίδια αύξηση	Ναι	Ναι

Κανόνες Big-Oh

- Εάν η $f(n)$ είναι πολυωνυμική βαθμού d, τότε η $f(n)$ είναι $O(n^d)$, δηλ.,
  1. Αφαιρούμε όρους χαμηλότερου βαθμού
  2. Αφαιρούμε σταθερές
- Χρησιμοποιούμε τη χαμηλότερη δυνατή κλάση:
  - Λέμε "$2n$ είναι $O(n)$" αντί "$2n$ είναι $O(n^2)$"
- Χρησιμοποιούμε την πιο απλή έκφραση
  - Λέμε "$3n + 5$ είναι $O(n)$" αντί "$3n + 5$ είναι $O(3n)$"

Ασυμπτωτική ανάλυση αλγορίθμων

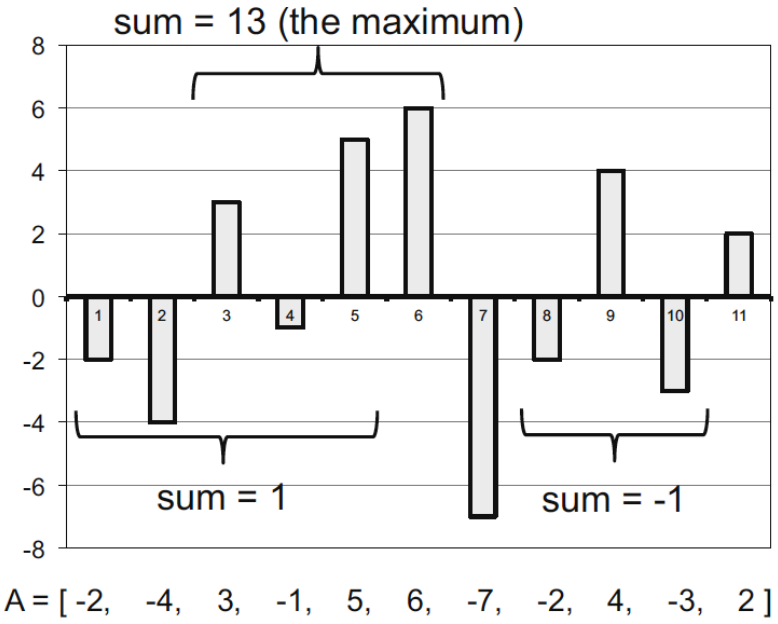
- Η ασυμπτωτική ανάλυση αλγόριθμου ορίζει το χρόνο εκτέλεσης σε big-Oh notation
- Για να πραγματοποιήσουμε ασυμπτωτική ανάλυση αλγόριθμου:
  - Βρίσκουμε τον αριθμό πρωτογενών λειτουργιών στη χειρότερη περίπτωση ως συνάρτηση του μεγέθους εισόδου
  - Εκφράζουμε αυτή τη συνάρτηση σε big-Oh notation
- Παράδειγμα:
  - Λέμε ότι ο αλγόριθμος arrayMax "εκτελείται σε χρόνο $O(n)$"
- Από τη στιγμή που οι σταθερές και οι όροι χαμηλότερης τάξης εν τέλει θα αφαιρεθούν, μπορούμε να τους παρακάμψουμε όταν μετράμε βασικά βήματα υπολογιστή

Μελέτη περίπτωσης στην ανάλυση αλγορίθμων

- Έστω ένας πίνακας *n* ακεραίων, βρείτε τον υπό-πίνακα, A[j:k] που μεγιστοποιεί το άθροισμα

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.$$

- Πέρα από το ότι είναι μια ερώτηση που ζητείται να απαντηθεί σε συνεντεύξεις εργασιών, το **πρόβλημα μέγιστου υπό-πίνακα (maximum subarray problem)** έχει εφαρμογή στην ανάλυση μοτίβων ψηφιακών εικόνων.



Μία πρώτη (Αργή) Λύση

Υπολογίζουμε το άθροισμα κάθε πιθανού υπο-πίνακα ξεχωριστά.

Algorithm MaxsubSlow(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
m ← 0    // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s ← 0    // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

- Ο εξωτερικός βρόχος, για το δείκτη j, θα εκτελεστεί n φορές, ο εσωτερικός βρόχος, για το δείκτη k, θα εκτελεστεί το πολύ n φορές, ο πιο εσωτερικός βρόγχος, για το δείκτη i, θα εκτελεστεί το πολύ n φορές.
- Έτσι, ο χρόνος εκτέλεσης του αλγόριθμου MaxsubSlow είναι $O(n^3)$.

Ένας βελτιωμένος αλγόριθμος

- Ένας πιο αποδοτικός τρόπος για να υπολογίσουμε αυτά τα αθροίσματα είναι να χρησιμοποιήσουμε τα **προθεματικά αθροίσματα (prefix sums)**

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i$$

- Εάν έχουμε αυτά τα προθεματικά αθροίσματα (και υποθέτοντας ότι $S_0=0$), μπορούμε να υπολογίσουμε κάθε άθροισμα $s_{j,k}$ σε σταθερό χρόνο:

$$s_{j,k} = S_k - S_{j-1}$$

Ένας βελτιωμένος αλγόριθμος, συνέχεια

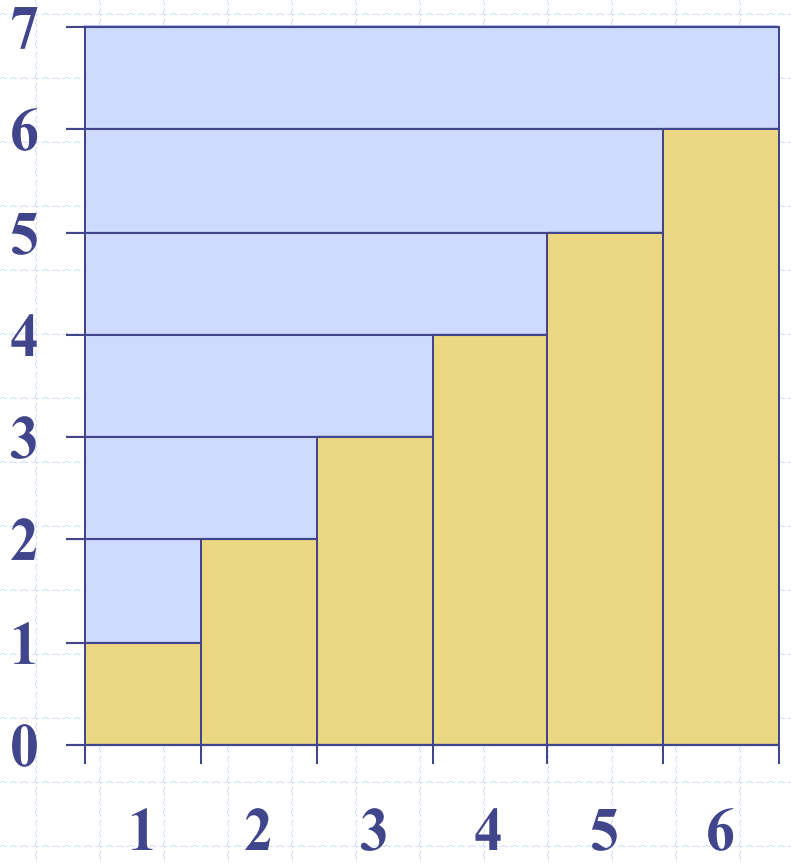
- Υπολογίζουμε όλα τα προθεματικά αθροίσματα
- Υπολογίζουμε όλα τα αθροίσματα υπό-πινάκων

Algorithm MaxsubFaster(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
S_0 ← 0      // the initial prefix sum
for i ← 1 to n do
    S_i ← S_{i−1} + A[i]
m ← 0        // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s = S_k − S_{j−1}
        if s > m then
            m ← s
return m
```

Αριθμητική πρόοδος

- Ο χρόνος εκτέλεσης του MaxsubFaster είναι $O(1 + 2 + \ldots + n)$
- Το άθροισμα των πρώτων n ακεραίων είναι $n(n + 1) / 2$
  - Υπάρχει μία απλή οπτική απόδειξη
- Οπότε, ο αλγόριθμος MaxsubFaster εκτελείται σε χρόνο $O(n^2)$



Ένας αλγόριθμος γραμμικού χρόνου

- Αντί να υπολογίσουμε το προθεματικό άθροισμα $S_t = s_{1,t}$, ας υπολογίσουμε το μέγιστο επιθεματικό άθροισμα, M_t, που είναι η μεγαλύτερη τιμή από το 0 και το μέγιστο $s_{j,t}$ για $j = 1,\ldots, t$.

$$M_t = \max\{0, \max_{j=1,\cdots,t}\{s_{j,t}\}\}$$

- εάν $M_t > 0$, τότε είναι η τιμή του αθροίσματος για τον μέγιστο υπό-πίνακα που τελειώνει στο t, και αν $M_t = 0$, τότε μπορούμε με ασφάλεια να αγνοήσουμε κάθε υπό-πίνακα που τελειώνει στο t.
- εάν γνωρίζουμε όλες τις τιμές M_t, για t = 1, 2, …, n, τότε η λύση στο πρόβλημα του μέγιστου υπό-πίνακα θα είναι η μέγιστη από αυτές τις τιμές.

Ένας αλγόριθμος γραμμικού χρόνου συνέχεια

- για t ≥ 2, εάν έχουμε έναν μέγιστο υπό-πίνακα που τελειώνει στο t και έχει θετικό πρόσημο, τότε είναι είτε A[t : t] είτε δημιουργείται από το μέγιστο υπό-πίνακα που τελειώνει στο t − 1 συν το A[t]. Έτσι ορίζουμε το $M_0 = 0$ και

$$M_t = \max\{0,\ M_{t-1} + A[t]\}$$

- **Εάν δεν ίσχυε, τότε θα μπορούσαμε να δημιουργήσουμε έναν υπό-πίνακα με μεγαλύτερο άθροισμα ανταλλάσσοντας αυτόν που επιλέξαμε να τελειώνει στο t − 1 με το μέγιστο να τελειώνει στο t − 1, κάτι όμως που αντικρούει το γεγονός ότι έχουμε τον μέγιστο υπό-πίνακα που τελειώνει στο t.**
- Επίσης, εάν στην τιμή του μέγιστου υπό-πίνακα που τελειώνει στο t − 1 προσθέτοντας το A[t] το άθροισμα δεν είναι πλέον θετικός αριθμός, τότε $M_t = 0$, γιατί δεν υπάρχει υπό-πίνακας που να τελειώνει στο t με θετικό άθροισμα.

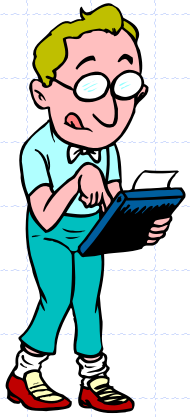
Ένας αλγόριθμος γραμμικού χρόνου, συνέχεια

Algorithm MaxsubFastest(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

$M_0 \leftarrow 0$ // the initial prefix maximum
for $t \leftarrow 1$ **to** n **do**
$M_t \leftarrow \max\{0,\ M_{t-1} + A[t]\}$
$m \leftarrow 0$ // the maximum found so far
for $t \leftarrow 1$ **to** n **do**
$m \leftarrow \max\{m,\ M_t\}$
return m

- Ο αλγόριθμος MaxsubFastest αποτελείται από δύο βρόχους, ο καθένας εκτελείται n φορές και απαιτεί O(1) χρόνο σε κάθε επανάληψη. Έτσι, ο συνολικός χρόνος εκτέλεσης του αλγόριθμου MaxsubFastest είναι O(n).

Επανάληψη που πρέπει να κάνετε στα μαθηματικά



- Αθροίσματα
- Δυνάμεις
- Λογάριθμοι
- Τεχνικές απόδειξης
- Βασικές πιθανότητες

- Ιδιότητες δυνάμεων:
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c*\log_a b}$
- Ιδιότητες λογαρίθμων:
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b x^a = a\log_b x$
  - $\log_b a = \log_x a/\log_x b$

Συγγενείς του Big-Oh

big-Omega

- f(n) είναι $\Omega(g(n))$ εάν υπάρχει μία σταθερά $c > 0$ και μία ακέραια σταθερά $n_0 \geq 1$ έτσι ώστε

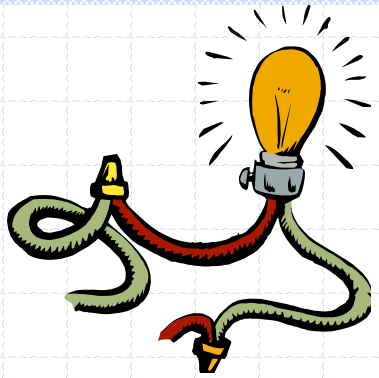
$$f(n) \geq c\, g(n) \text{ για } n \geq n_0$$

big-Theta

- f(n) είναι $\Theta(g(n))$ εάν υπάρχουν σταθερές $c' > 0$ και $c'' > 0$ και μία ακέραια σταθερά $n_0 \geq 1$ έτσι ώστε

$$c'g(n) \leq f(n) \leq c''g(n) \text{ για } n \geq n_0$$

Ασυμπτωτική σημειογραφία



big-Oh

- Η f(n) είναι O(g(n)) εάν f(n) είναι ασυμπτωτικά μικρότερη ή ίση από την g(n)

big-Omega

- Η f(n) είναι Ω(g(n)) εάν η f(n) είναι ασυμπτωτικά μεγαλύτερη ή ίση από την g(n)

big-Theta

- Η f(n) είναι Θ(g(n)) εάν η f(n) είναι ασυμπτωτικά ίση με την g(n)

Παραδείγματα των συγγενών του Big-Oh

- **$5n^2$ είναι $\Omega(n^2)$**

  H $f(n)$ είναι $\Omega(g(n))$ εάν υπάρχει σταθερά $c > 0$ και μία ακέραια σταθερά $n_0 \geq 1$ έτσι ώστε $f(n) \geq c\, g(n)$ για $n \geq n_0$

  Ισχύει για $c = 4$ και $n_0 = 1$

- **$5n^2$ είναι $\Omega(n)$**

  H $f(n)$ είναι $\Omega(g(n))$ εάν υπάρχει σταθερά $c > 0$ και μία ακέραια σταθερά $n_0 \geq 1$ έτσι ώστε $\mathrm{f}(n) \geq c\, g(n)$ για $n \geq n_0$

  Ισχύει για $c = 1$ και $n_0 = 1$

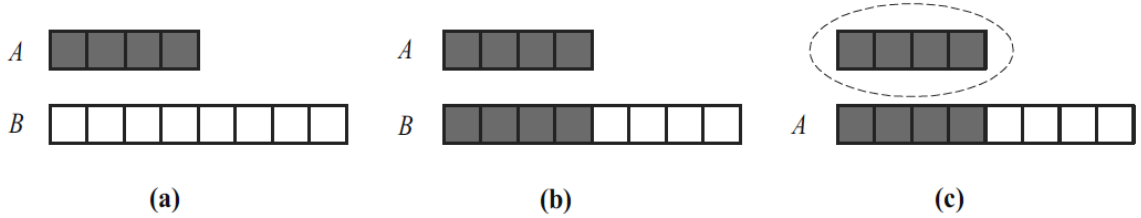
- **$5n^2$ είναι $\Theta(n^2)$**

  H $f(n)$ είναι $\Theta(g(n))$ εάν είναι $\Omega(n^2)$ και $O(n^2)$. Η $f(n)$ είναι $O(g(n))$ εάν υπάρχει σταθερά $c > 0$ και μία ακέραια σταθερά $n_0 \geq 1$ έτσι ώστε $\mathrm{f}(n) \leq c\, g(n)$ for $n \geq n_0$

  Ισχύει για $c = 6$ και $n_0 = 1$

Επιμερισμός

- Ο **επιμερισμένος χρόνος εκτέλεσης** μίας λειτουργίας για μία σειρά λειτουργιών είναι ο χρόνος εκτέλεσης μιας σειράς λειτουργιών προς το πλήθος των λειτουργιών, στη χειρότερη δυνατή περίπτωση.
- Παράδειγμα: Ένας επεκτάσιμος πίνακας, A. Όταν χρειάζεται να μεγαλώσει:
  a. Δημιουργία ενός νέου πίνακα B μεγαλύτερης χωρητικότητας.
  b. Αντιγραφή του A[i] στο B[i], για i = 0, . . . , n − 1, όπου n το μέγεθος του A.
  c. A = B



Περιγραφή επεκτάσιμου πίνακα

- Έστω add(e) η λειτουργία που προσθέτει το e στο τέλος του πίνακα
- Όταν ο πίνακας γεμίσει τον αντικαθιστούμε με έναν μεγαλύτερο
- Πόσο μεγάλος πρέπει να είναι ο νέος πίνακας;
  - Στρατηγική βηματικής αύξησης: αύξηση του μεγέθους του πίνακα κατά μία σταθερά c
  - Στρατηγική διπλασιασμού: διπλασιασμός του μεγέθους

```
Algorithm add(e)
  if t = A.length – 1 then
    B ← new array of
          size …
    for i ← 0 to n–1 do
      B[i] ← A[i]
    A ← B
  n ← n + 1
  A[n–1] ← e
```

Σύγκριση στρατηγικών

- Συγκρίνουμε τη στρατηγική της βηματικής αύξησης με τη στρατηγική του διπλασιασμού αναλύοντας το συνολικό χρόνο $T(n)$ που χρειάζονται n προσθήκες
- Υποθέτουμε ότι ξεκινάμε με μία άδεια λίστα, που αναπαρίσταται από έναν επεκτάσιμο πίνακα μεγέθους 1
- Ορίζουμε ως επιμερισμένο χρόνο εκτέλεσης μίας λειτουργίας add το μέσο χρόνο που απαιτείται από μία λειτουργία add για μια σειρά λειτουργιών, δλδ., $T(n)/n$

Ανάλυση στρατηγικής βηματικής αύξησης

- Για n λειτουργίες add, αντικαθιστούμε τον πίνακα $k = n/c$ φορές, όπου c είναι μία σταθερά
- Ο συνολικός χρόνος $T(n)$ μίας σειράς n add λειτουργιών είναι ανάλογος του

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Επειδή η c είναι σταθερά, η $T(n)$ είναι $O(n + k^2)$, δλδ., $O(n^2)$
- Έτσι, ο επιμερισμένος χρόνος μίας λειτουργίας add είναι $O(n)$

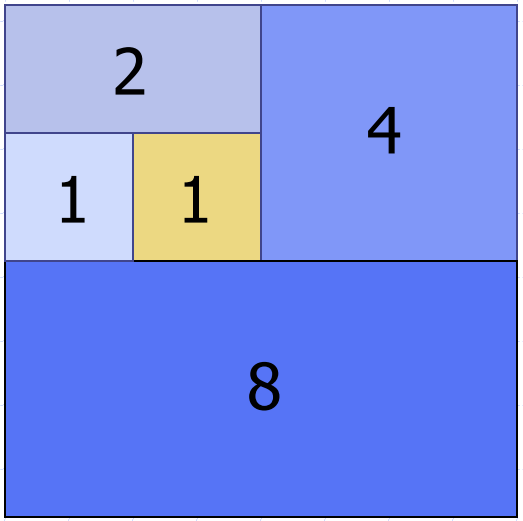
Ανάλυση στρατηγικής διπλασιασμού

- Αντικαθιστούμε τον πίνακα $k = \log_2 n$ φορές
- Ο συνολικός χρόνος $T(n)$ μίας σειράς από n λειτουργίες είναι ανάλογος με

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- Η $T(n)$ είναι $O(n)$
- Ο χρόνος απόσβεσης μίας λειτουργίας add είναι $O(1)$

γεωμετρική σειρά



Μέθοδος λογιστή για τη στρατηγική διπλασιασμού

- Θεωρούμε τον υπολογιστή ως μία συσκευή που λειτουργεί με κέρματα και απαιτεί 1 **cyber-δολάριο** για σταθερή ποσότητα υπολογιστικού χρόνου.
- Χρεώνοντας 3 cyber-δολάρια για κάθε λειτουργία add, ο επιμερισμένος χρόνος εκτέλεσης θα είναι O(1).
  - Υπερχρεώνουμε κάθε λειτουργία add που δεν προκαλεί υπερχείλιση με 2 cyber-δολάρια.
  - Σκεφτείτε τα 2 cyber-δολάρια που κερδήθηκαν σαν να «αποθηκεύονται» στο στοιχείο που εισάγεται.
  - Μία υπερχείλιση θα συμβεί όταν ο πίνακας A έχει 2^i στοιχεία.
  - Έτσι, ο διπλασιασμός θα απαιτήσει 2^i cyber- δολάρια.
  - Αυτά τα cyber- δολάρια βρίσκονται στα κελιά 2^{i-1} μέχρι 2^i-1.