Formalizing…definitions

- *T (N) = O (f(N))* if there are positive constants c and n_0 such that $T(N) <= c\ f(N)$ when $N >= n_0$
  upper
- We say that T(N) is of the "order of *f(N)*" or Big-Oh *f(N)* or just O(*f(N)*)
- In words, this means *T(N)* is of the order of *f(N)* if you can find a point n_0 after which *T(N)* is smaller than a linearly scaled version of *f(N)*. Roughly speaking:
  - The point n_0 helps ignore the additive constants
  - The factor *c* helps ignore the multiplicative constants
  - Focus is only on the dominating "N" term

HOMEWORK
(Deadline 17/01/2014)

Which of the following is/are true? Prove:

$40N+400 = O(2N+1)$: Yes, $n_0=1$, $c=400$

$2N+1 = O(40N+400)$: yes, $n_0=1$, $c=1$

$40N+400 = O(40\log N+400)$: No

$40\log N+400 = O(40N+400)$: Yes, $n_0=1$, $c=1$

In general $N = O(N^k)\ \forall k>1$

For complexity, we are interested in smallest k.

Understanding Big-Oh

a=5, b=6, c=7, c1=40

Refer to spreadsheet

Other definitions

O = asymptotic upper bnd
Ω = " lower "

- $T(N) = \Omega(\ f(N)\ )$ if there are positive constants c and n_0 such that $T(N) \geq c\ f(N)$ when $N \geq n_0$,

  lower bound

  - Growth rate of T(N) is more than of g(N) $\{N^2 = \Omega(N)\}$

- $T(N) = \Theta(\ f(N)\ )$ if $T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$

  tight upper & lower bnd

  - Growth rates of T(N) and f(N) are same $\{aN + b = \Theta(N)\}$

- $T(N) = o(\ f(N)\ )$ if for all positive constants c there is an n_0 such that $T(N) < c\ f(N)$ when $N > n_0$.

  strict upper bound

  - Growth rate of T(N) is strictly less than of f(n) $\{N = o(N^2)\}$

Q: If $f(N) = O(f'(N))$,

is $f'(N) = \Omega(f(N))$?

Ans: If $\forall N \geq n_0 \quad f(N) \leq c f'(N)$

then $\forall N \geq n_0 \quad f'(N) \geq \frac{1}{c} f(N)$

positive

Q: Why is definition of o that

$f(N) \neq \Omega(f'(N))$

$\forall c, \exists n_0$ s.t $\forall N \geq n_0$

$f(N) < cf'(N)$ (a)

H/w: Reflect

& NOT

$\exists n_0 \;\&\; c > 0$ s.t $\forall N \geq n_0$

$f(N) < cf'(N)$ (b)

?

$\boxed{\text{Ans. (b) does NOT exclude } f(N) = \Omega(f'(N)) \text{ (\& therefore NOT } f(N) = \Theta(f'(N))\text{) BUT (a) excludes both } f(N) = \Omega(f'(N)) \;\&\; f(N) = \Theta(f'(N))}$

…more on definitions

- *$T(N) = \Omega(f(N))$*
  - f(N) is O(T(N))
- *$T(N) = \Theta (f(N))$*
  - A "tighter" proof – generally done in advanced analysis
- *$T(N) = o(f(N))$:* What is the real difference from big-Oh?
  - If T(N) is O(f(N), it may still be Θ (f(N)) also
  - But if T(N) is o(f(N)) it will <u>not</u> be Θ (f(N))

Examples (of functions)

- 2N+3 is O(N)
  - $T(N) = 2N+3$, $f(N) = N$
  - For $c=6$, $n_0=1$, $T(N) < c\, f(N)$ for $n >= n0$
- Note that 2N+3 is also $O(N^2)$, $O(N^3)$ etc, but by convention we always state the lowest order when we talk about order of complexity of algorithms
- $f(N)$ is also $O(T(N))$ ($c=1$, $n_0=1$)
- So $T(N)$ is $\Theta(f(N))$

(Similar to homework problem)

Examples

- $4N^2 + N + 5$ is conventionally described as $O(N^2)$ although it is also $O(N^2 + N)$
  - Lower order terms usually not mentioned
- $5N + 3 \log N \sim O(N)$
- (We don't do formal proofs of finding *c* and n_0 – just write the order intuitively, based on dominating term)

Idea: drop N in $N^2 + N$: $N = o(N^2)$
drop $\log N$ in $N + \log N$: $\log N = o(N)$

Can there be a more efficient search algo than A or B? Ans: Yes

interpolation search

For an interpolation search to be practical, two assumptions must be satis ed:

1. Each access must be very expensive compared to a typical instruction. For example, the array might be on a disk instead of in memory, and each comparison requires a disk access.
2. The data must not only be sorted, it must also be fairly uniformly distributed. For example, a phone book is fairly uniformly distributed. If the input items are { 1, 2, 4, 8, 16, } , the distribution is not uniform

Key difference from binary search:
Search for "next" instead of "mid"

Replace: a $mid = \dfrac{high - low}{2}$

with:

value being searched

b $next = low + \left[\dfrac{x - a[low]}{a[high] - a[low]} * (high - low - 1) \right]$

PRO: Works well → Assuming uniform distribution in

x

low ←···a - - - → high

CON: b involves floating point operations unlike a and is therefore much more expensive.

Example-1 (analyzing programs)

```
for (n = 0; n <= N - 1; n++)
for (m = n+1; m <= N - 1; m++) {
   temp = A[n][m];
   A[n][m] = A[m][n];
   A[m][n] = temp;
}
```

Constant time execution

Only need to figure out how many times this is executed

1. At n=0, loop executed N-1 times
2. At n=1, N-2 times…

Total: N-1 + N-2 + N-3 … + 1 =

$= (N-1) * N/ 2 = (N^2-N)/2 \sim O(N^2)$

Example-2 (analyzing programs)

```
bool whatDoIDo(int a[], int arraysize) {


int *b,j,i, m, n;
m = a[0];
n = a[0];
b = new int[arraysize];
b[0] =0;
for (i=1; i < arraysize; i++) {
   if (m < a[i] ) m = a[i];
   if (n > a[i] ) n = a[i];
   b[i] = 0;
}
```

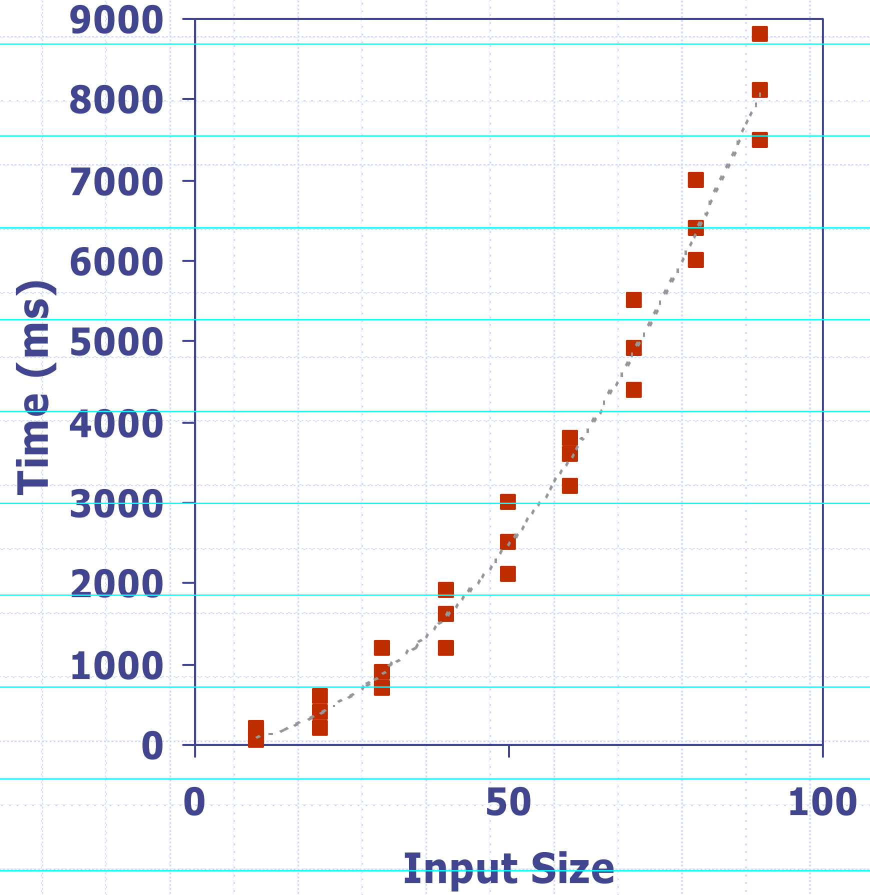
```

if ( (m - n + 1) == arraysize ) {
   for (i=0; i < arraysize; i++) {
       j=a[i];
       if (!b[j-n])
           b[j-n] ++;
       else {
          return false;
       }
   }
}
else
   return false;
return true;
}
```

Let arraysize=N, running time is O(N)

Revisiting concepts

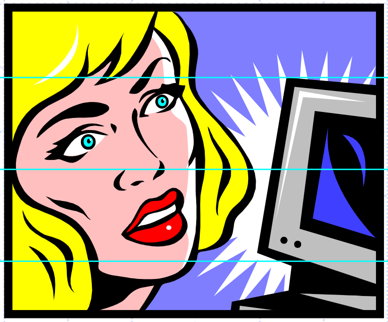
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode (§3.2)

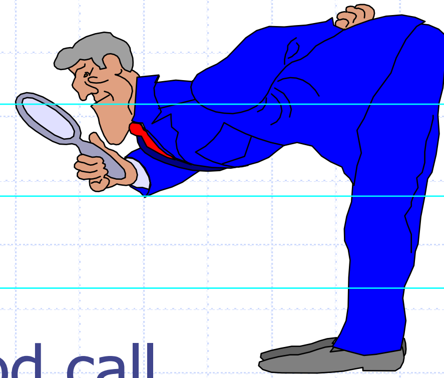
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

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

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

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

  Input …

  Output …

- Method call

  var.method (***arg*** [, ***arg***…])

- Return value

  return ***expression***

- Expressions

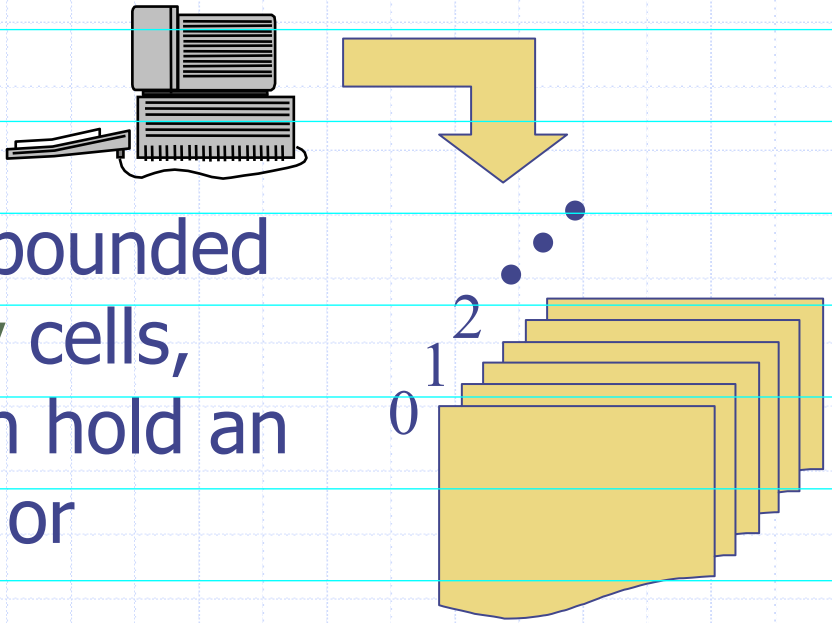
  $\leftarrow$ Assignment (like $=$ in Java)

  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

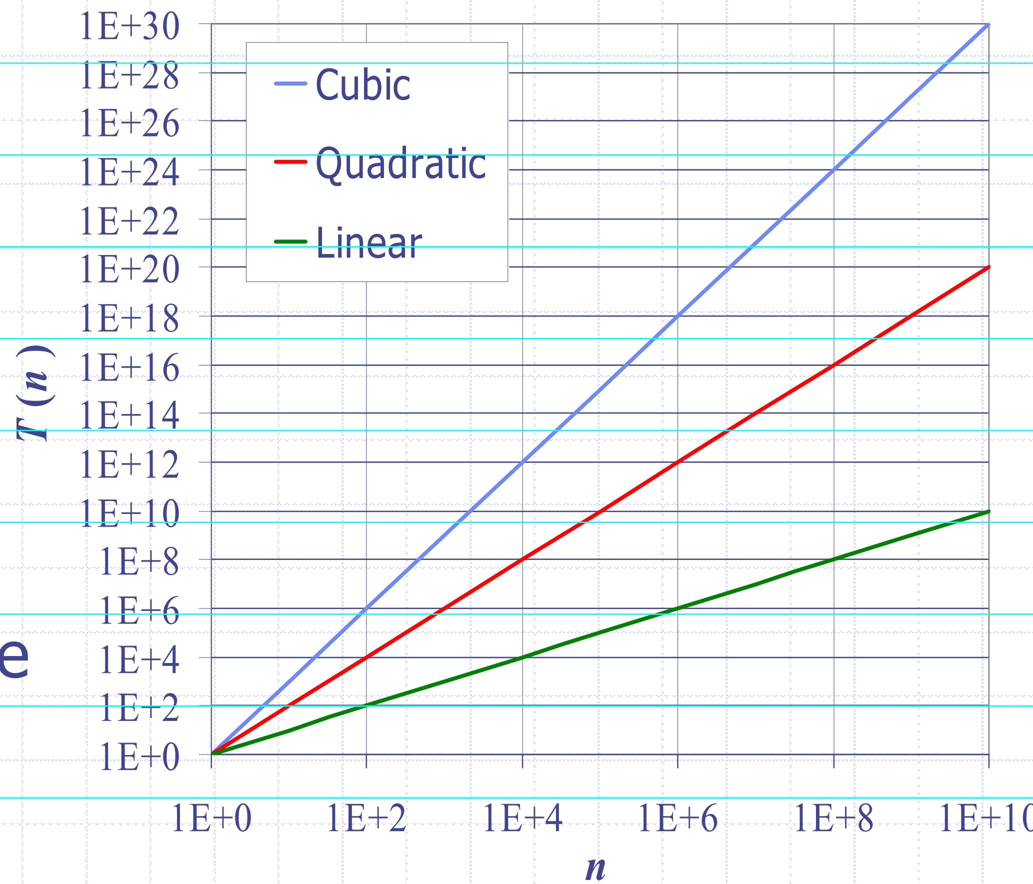
- A **CPU**



- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time.

Seven Important Functions (§3.3)

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ *1*
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

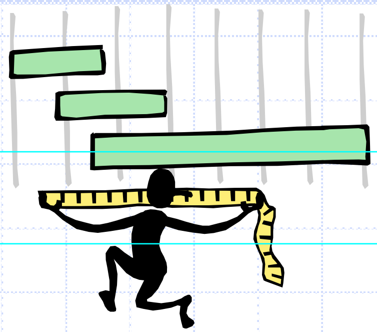
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations (§3.4)

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$8n - 2$

Estimating Running Time



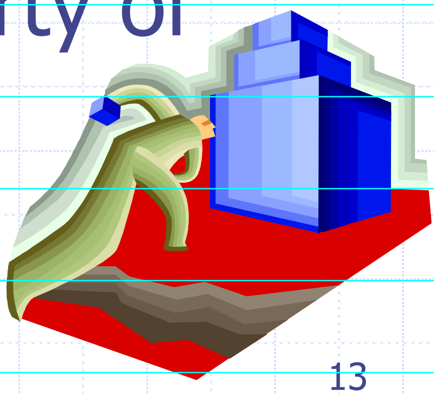
- Algorithm *arrayMax* executes $8n - 2$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of *arrayMax*. Then

$$a\,(8n - 2) \leq T(n) \leq b(8n - 2)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

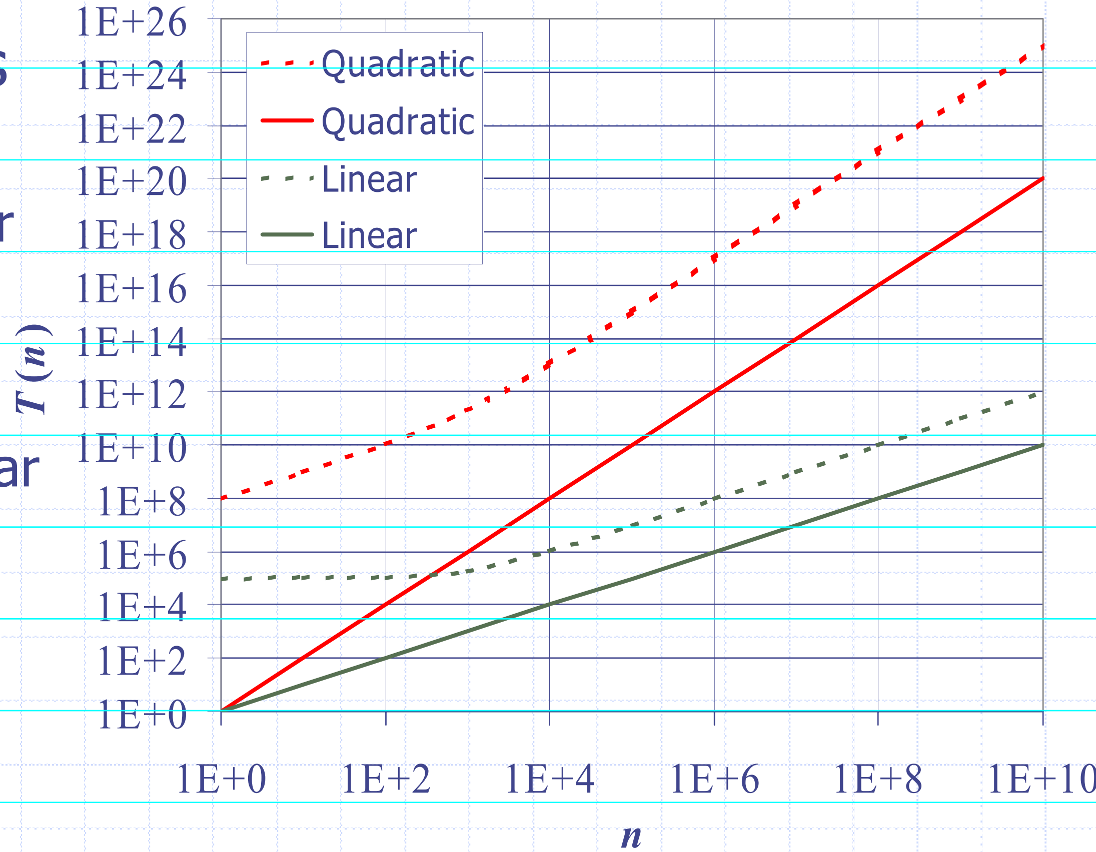
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function

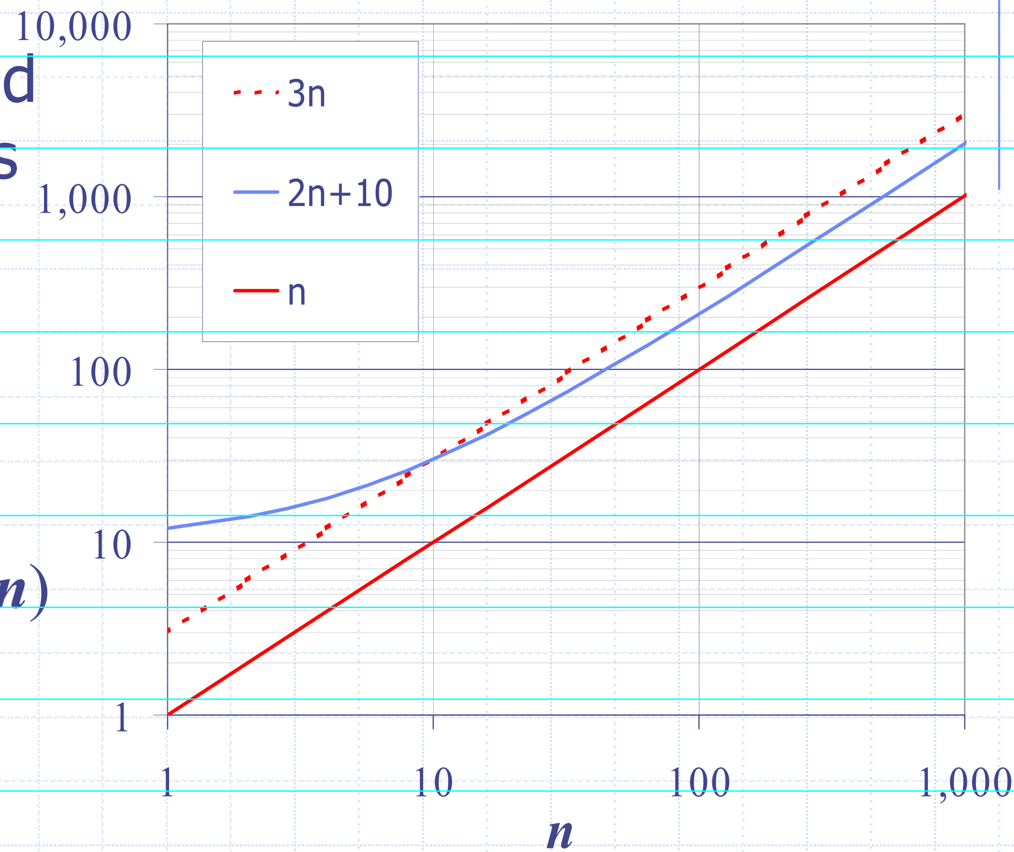


Big-Oh Notation (§3.4)

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

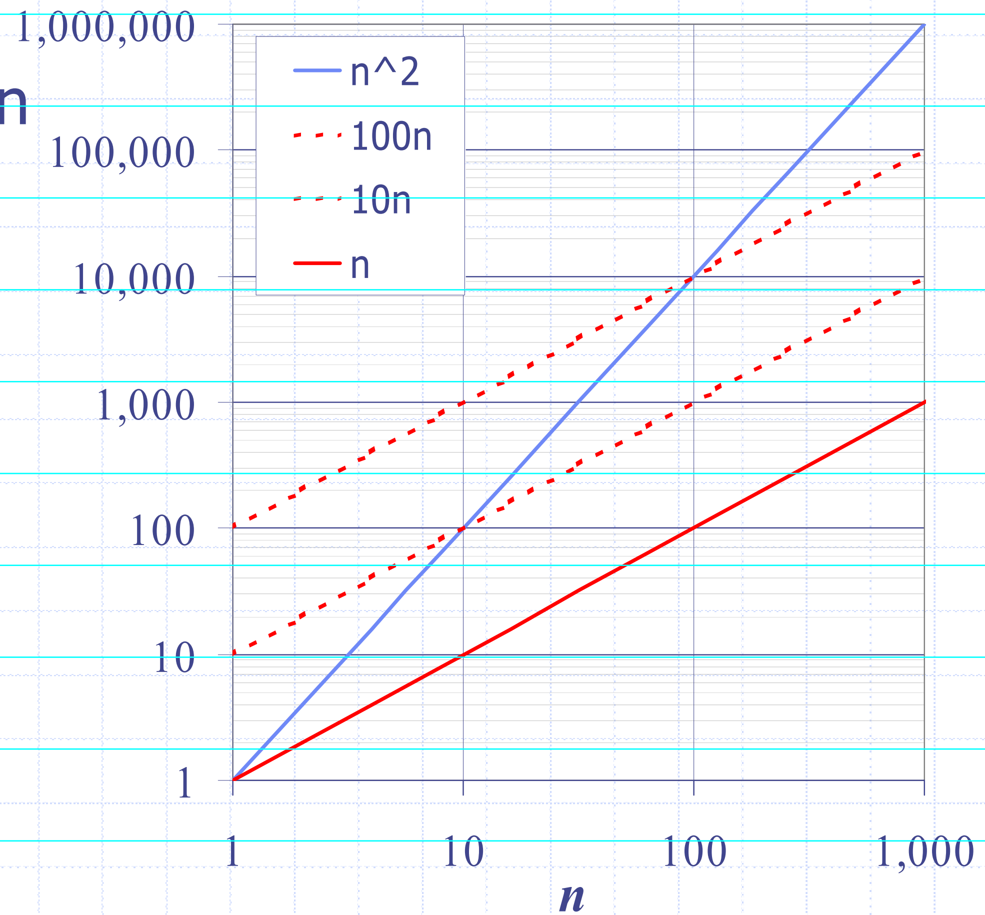
  $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

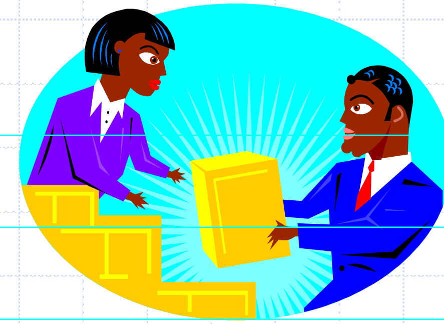


Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples



- $7n-2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

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

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

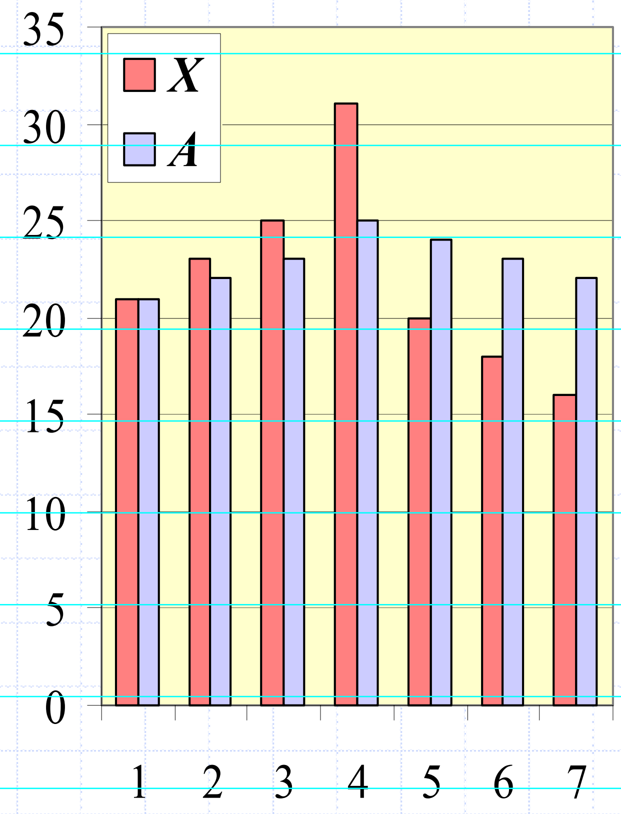
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



HOMEWORK (DEADLINE 22nd JAN 2014):

Present two algorithms for computing the array A of prefix averages of another array X (problem discussed on the previous slide). Analyse the running time of each algorithm.