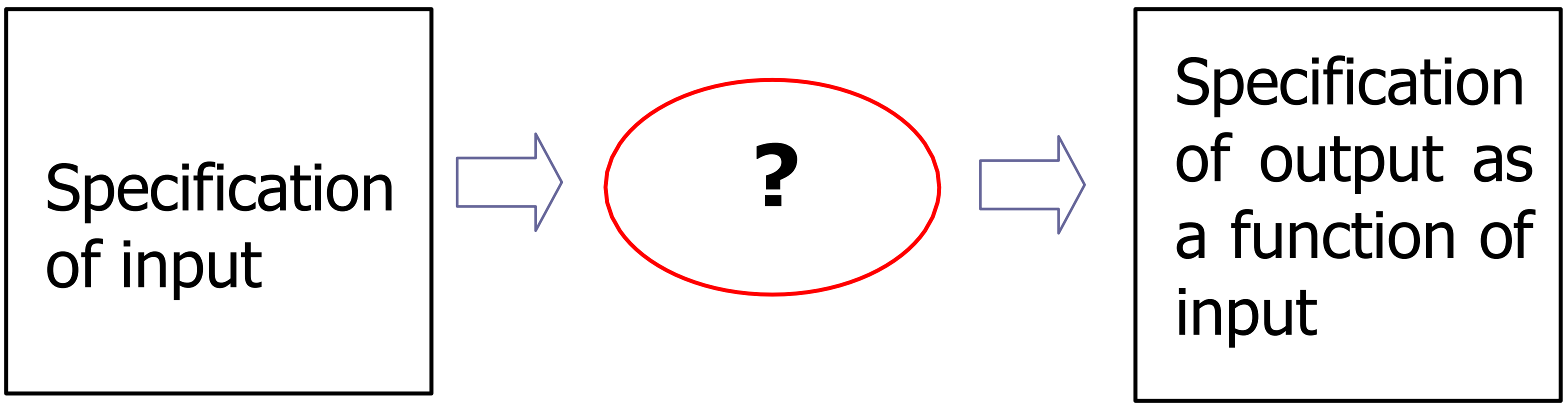
1.1 Introduction to Time complexity, O notation

Data Structures and Algorithms

- **Algorithm**: Outline, the essence of a computational procedure, step-by-step instructions
- **Program**: an implementation of an algorithm in some programming language
- **Data structure**: **Organization** of data needed to solve the problem

Algorithmic problem



- Infinite number of input *instances* satisfying the specification. For eg: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
  - 1, 20, 908, 909, 100000, 1000000000.
  - 3.

Algorithmic Solution

Input instance, adhering to the specification

Algorithm

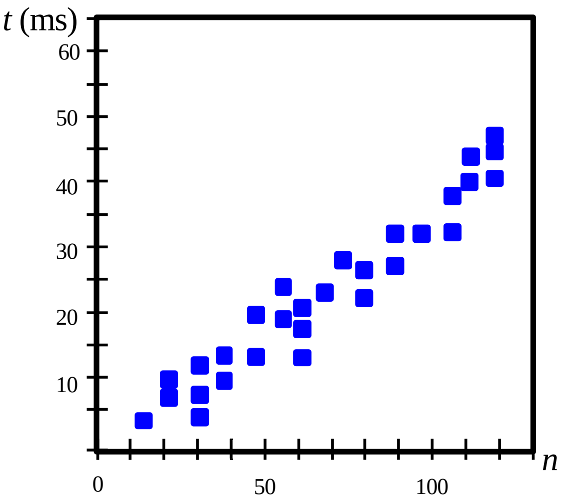
Output related to the input as required

- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- **Efficient**:
  - Running time
  - Space used
- **Efficiency as a function of input size**:
  - The number of bits in an input number
  - Number of data elements (numbers, points)

Measuring the Running Time



How should we measure the running time of an algorithm?

Experimental Study

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition.
- Use library function method like clock() to get an accurate measure of the actual running time.

Limitations of Experimental Studies

- It is necessary to implement and test the algorithm in order to determine its running time.
- Experiments can be done only on a limited set of inputs, and 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 should be used.

Beyond Experimental Studies

We will develop a general methodology for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of testing one of its implementations.
- Takes into account all possible inputs.
- Allows one to evaluate the efficiency of any algorithm in a way that is independent of the hardware and software environment.

Pseudo-Code

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm.
- Eg: **Algorithm** arrayMax(A, n):

  Input*:* An array A storing n integers.

  Output*:* The maximum element in A.

  currentMax←A[0]

  for i←1 **to** n-1 **do**

      if currentMax < A[i] **then** currentMax ← A[i]

  return currentMax

Pseudo-Code

It is more structured than usual prose but less formal than a programming language

- Expressions:
  - use standard mathematical symbols to describe numeric and boolean expressions
  - Use $\leftarrow$ for assignment (“=” in Java)
  - use = for the equality relationship (“==” in C)
- Method Declarations:
  - **Algorithm** name(param1**,** param2)

Pseudo Code

- Programming Constructs:
  - decision structures: **if ... then ... [else ...]**
  - while-loops: **while ... do**
  - repeat-loops: **repeat ... until ...**
  - for-loop: **for ... do**
  - array indexing: **A[i], A[i,j]**
- Methods:
  - calls: object method(args)
  - returns: **return** value

Analysis of Algorithms

- **Primitive Operation:** Low-level operation independent of programming language. Can be identified in pseudo-code. For eg:
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - arithmetic an logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

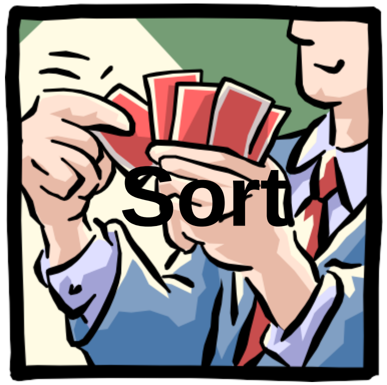
Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sort

A	3	4	6	8	9	7	2	5	1
	1					j →			n
					← i				

Strategy

- Start “empty handed”
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤ A[2]≤ …≤A[n]

```
for j←2 to n do
    key ← A[j]
    Insert A[j] into the sorted sequence
    A[1..j-1]
    i←j-1
    while i>0 and A[i]>key
      do A[i+1]←A[i]
         i--
    A[i+1]←key
```

Analysis of Insertion Sort

	cost	**times**
for j←2 **to** *n* **do**	c_1	n
key←A[j]	c_2	n-1
Insert A[j] into the sorted sequence A[1..j-1]		
i←j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]←A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1] ← key	c_7	n-1

$$\text{Total time} = n(c_1+c_2+c_3+c_7) + \sum_{j=2}^{n} t_j \; (c_4+c_5+c_6) - (c_2+c_3+c_5+c_6+c_7)$$

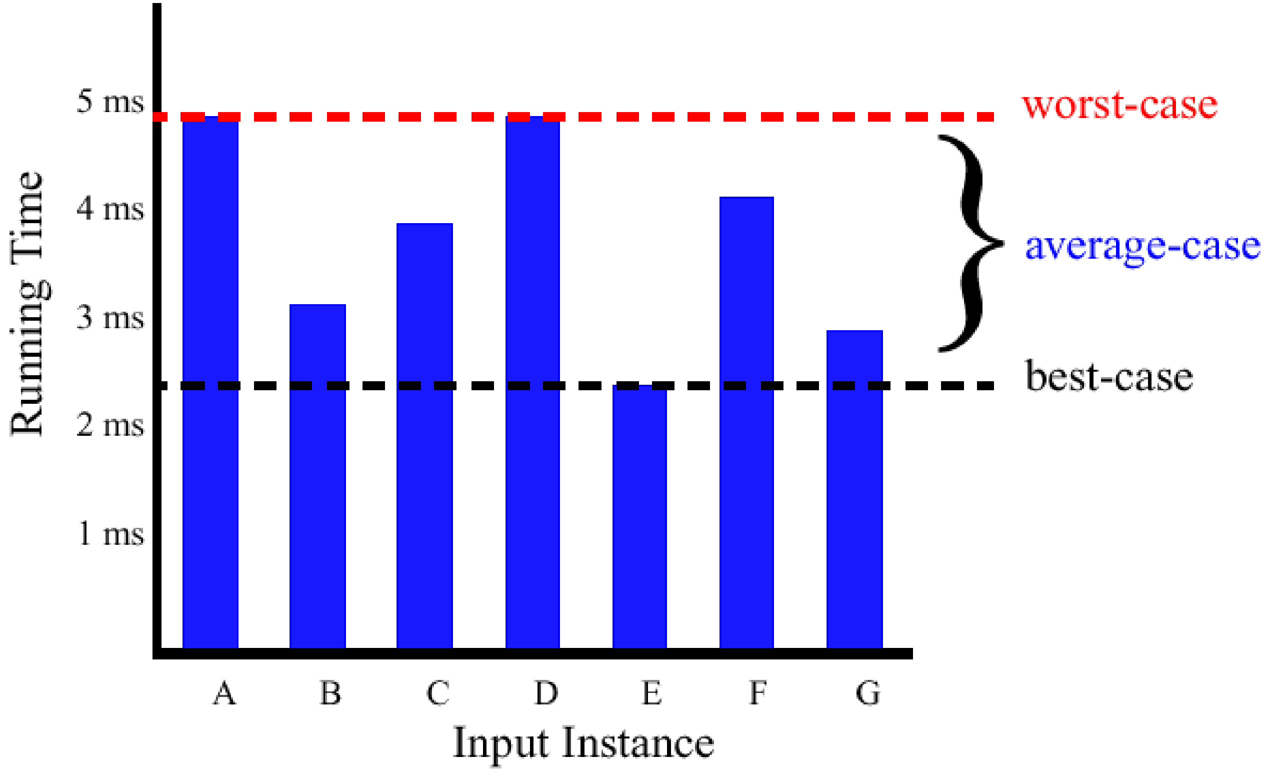
Best/Worst/Average Case

Total time = $n(c_1+c_2+c_3+c_7) + \sum^{n}_{j=2} t_j \ (c_4+c_5+c_6)$
$- (c_2+c_3+c_5+c_6+c_7)$

- **Best case**: elements already sorted; t_j=1, running time = f(n), i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order; t_j=j, running time = f(n^2), i.e., *quadratic* time
- **Average case**: t_j=j/2, running time = f(n^2), i.e., *quadratic* time

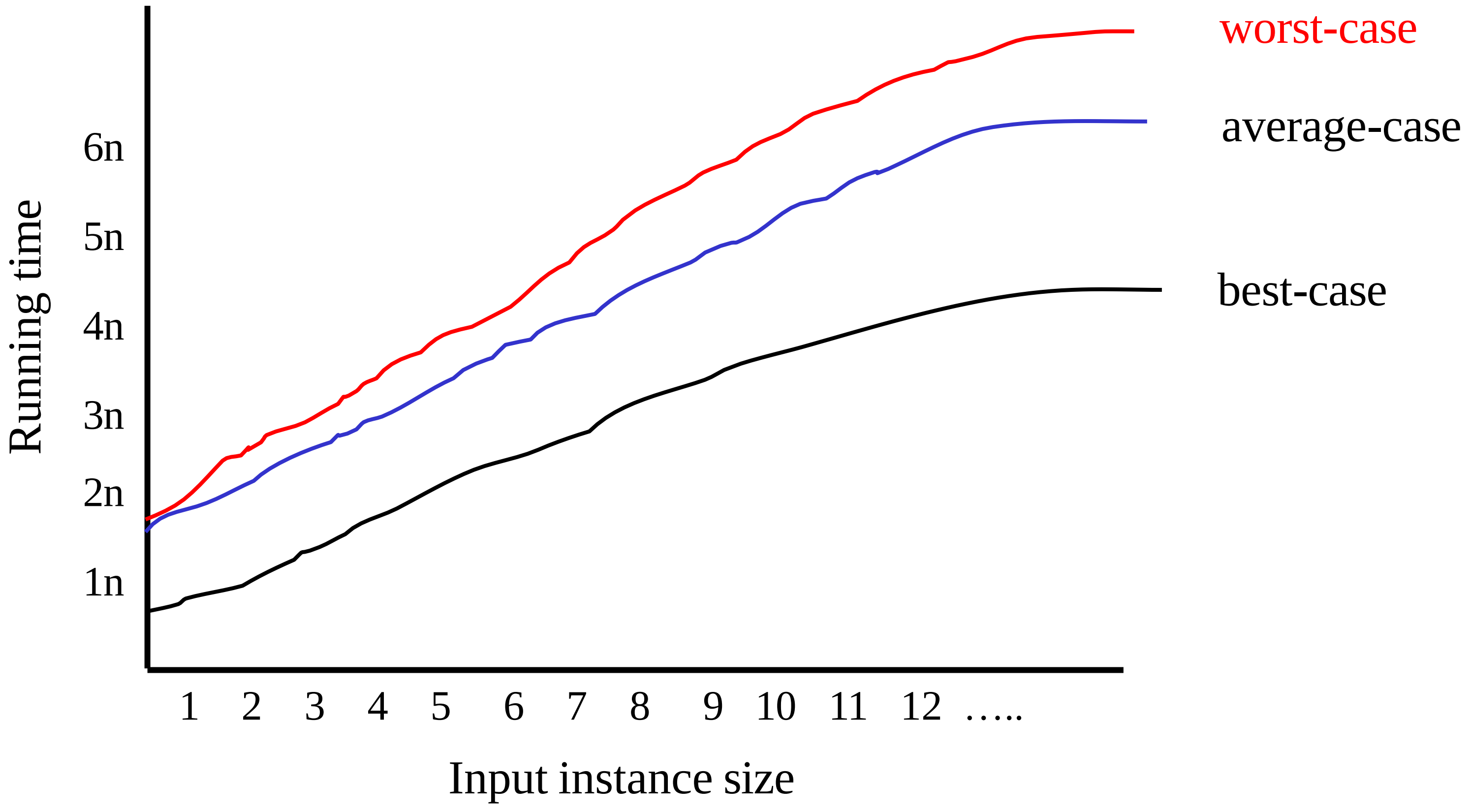
Best/Worst/Average Case (2)

- For a specific size of input n, investigate running times for different input instances:



Best/Worst/Average Case (3)

For inputs of all sizes:



Best/Worst/Average Case (4)

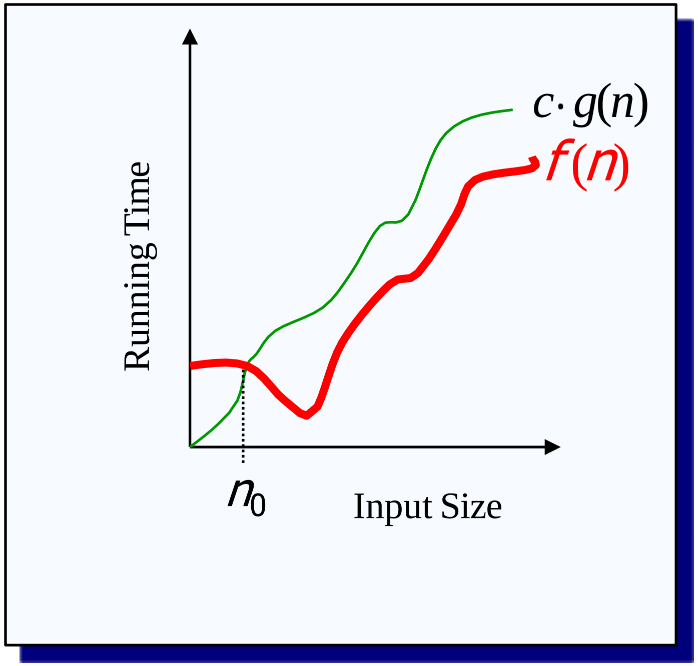
- **Worst case** is usually used: It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
- For some algorithms **worst case** occurs fairly often
- **Average case** is often as bad as the **worst case**
- Finding **average case** can be very difficult

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of ”details”, which may be affected by specific implementation and hardware
  - like “rounding”: 1,000,001 $\approx$ 1,000,000
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
  - Asymptotically more efficient algorithms are best for all but small inputs

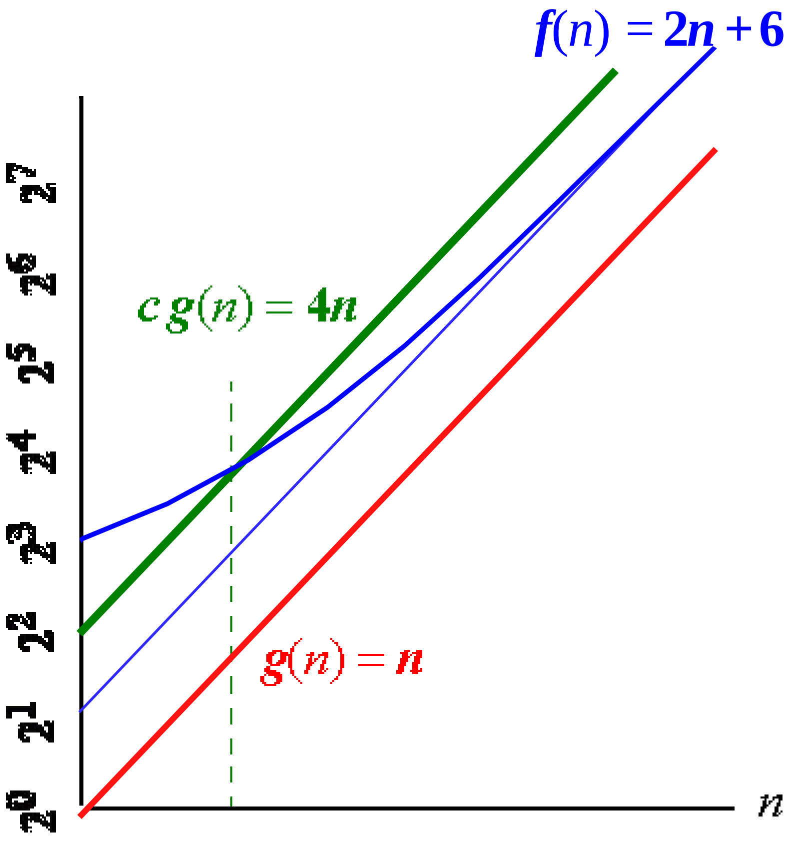
Asymptotic Notation

- The “big-Oh” *O*-Notation
  - asymptotic upper bound
  - f(n) = O(g(n)), if there exists constants *c* and n_0, *s.t.* **f(n) $\leq$ c g(n)** for n $\geq n_0$
  - f(n) and g(n) are functions over non-negative integers
- Used for *worst-case* analysis



Example

For functions f(n) and g(n) there are positive constants c and n_0 such that: $f(n) \le c\, g(n)$ for $n \ge n_0$



conclusion:

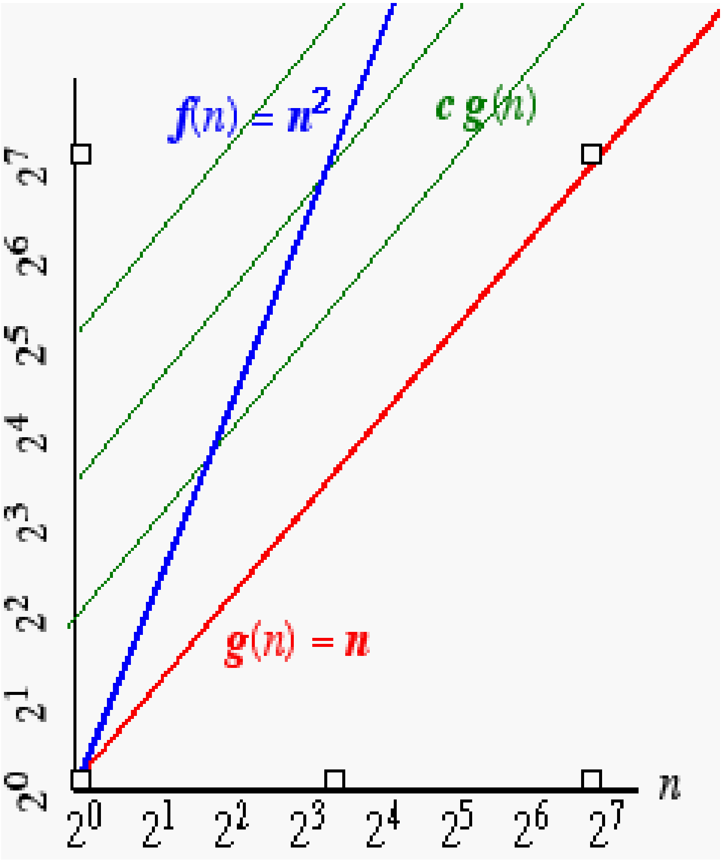
2n+6 is O(n).

Another Example

On the other hand…
n^2 is not O(n) because there is no c and n_0 such that:

$$n^2 \le cn \text{ for } n \ge n_0$$

The graph to the right illustrates that no matter how large a c is chosen there is an n big enough that $n^2 > cn$)



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors.
  - $50\,n \log n$ is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Even though $(50\,n \log n)$ is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

Asymptotic Analysis of Running Time

- Use *O*-notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times
  - an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time
  - similarly, $O(\log n)$ is better than $O(n)$
  - hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still $O(n)$ but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

Input: An n-element array X of numbers.

Output: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

for i ← 0 **to** n-1 **do**

  a ← 0

  for j ← 0 **to** i **do**

    a ← a + X[j] ← 1 step

  A[i] ← a/(i+1)

i iterations with i=0,1,2...n-1

n iterations

return array A

Analysis: running time is $O(n^2)$

A Better Algorithm

Algorithm prefixAverages2(X):

Input: An *n*-element array X of numbers.

Output: An *n*-element array A of numbers such that A[*i*] is the average of elements X[0], ... , X[*i*].

$s \leftarrow 0$

for i $\leftarrow$ 0 **to** n **do**

  $s \leftarrow s + X[i]$

  $A[i] \leftarrow s/(i+1)$

return array A

Analysis: Running time is O(n)

Asymptotic Notation *(terminology)*

- Special classes of algorithms:
  - Logarithmic: O(log n)
  - Linear: O(n)
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k)$, $k \geq 1$
  - Exponential: $O(a^n)$, $a > 1$
- "Relatives" of the Big-Oh
  - Ω (f(n)): Big Omega -asymptotic *lower* bound
  - Θ (f(n)): Big Theta -asymptotic *tight* bound