Data Structures and Algorithms

Lecture 2: *Analysis of Algorithms, Asymptotic notation*

Lilia Georgieva

Outline

- Pseudocode
- Theoretical Analysis of Running time
  - Primitive Operations
  - Counting primitive operations
- Asymptotic analysis of running time

Pseudocode

- In this course, we will mostly use pseudocode to describe an algorithm
- Pseudocode is a 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 …

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

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
return currentMax
```

Pseudocode Details

- 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

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 $currentMax$

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion?
  3) How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria
- In this course we are interested in the second and third criteria

Comparing Algorithms

- Time complexity
  - The amount of time that an algorithm needs to run to completion
- Space complexity
  - The amount of memory an algorithm needs to run
- We will occasionally look at space complexity, but we are mostly interested in time complexity in this course
- Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

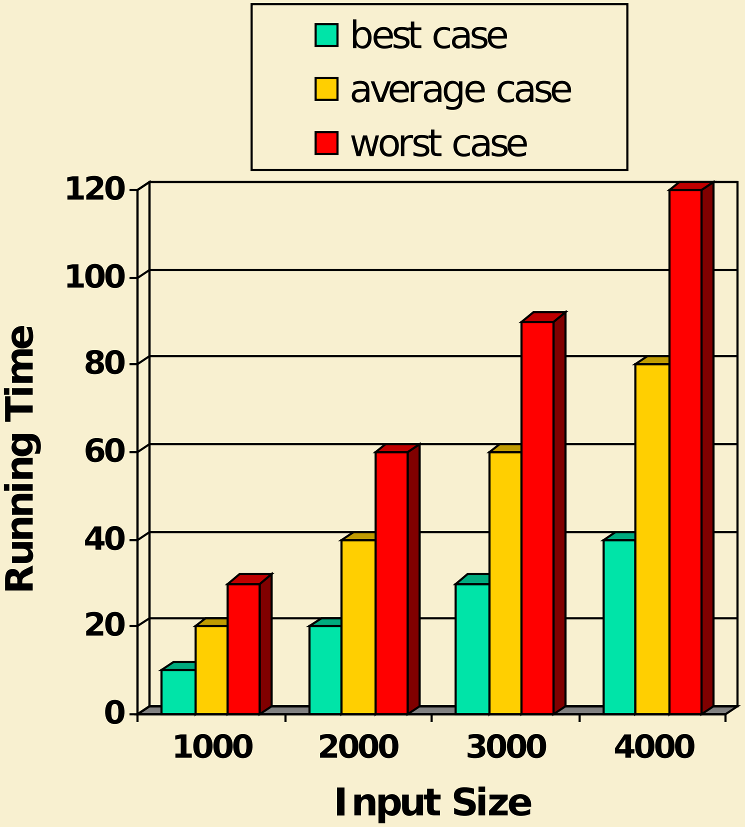
- The running time of an algorithm typically grows with the input size
  - idea: analyze running time as a function of input size

How to Calculate Running Time

- Even on inputs of the same size, running time can be very different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right
- Idea: analyze running time in the
  - best case
  - worst case
  - average case

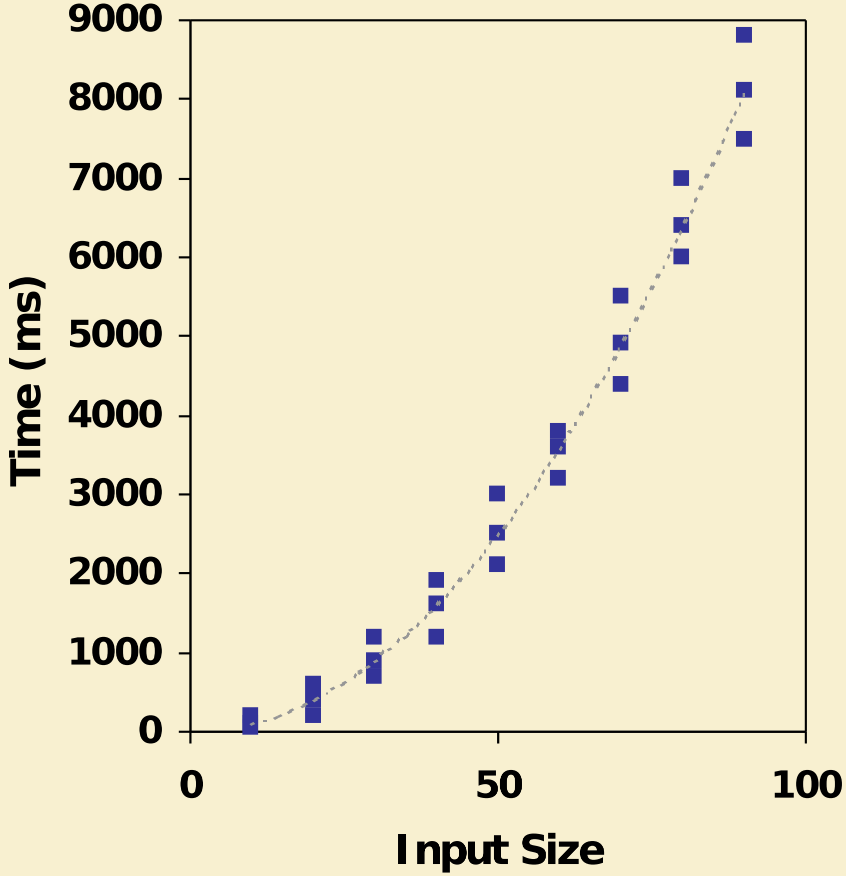
How to Calculate Running Time

- Best case running time is usually useless
- Average case time is very useful but often difficult to determine
- We focus on the **worst case** running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Evaluation of Running Time

- 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

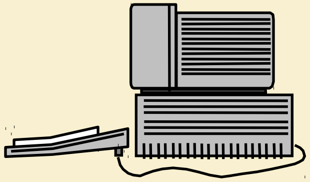
- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult
  - Results may not be indicative of the running time on other inputs not included in the experiment
  - In order to compare two algorithms, the same hardware and software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

Theoretical Analysis of Running Time

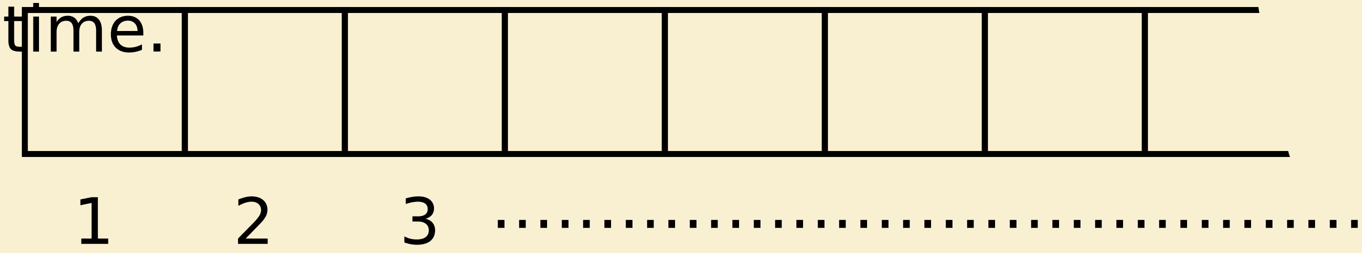
- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

RAM: The Random Access Machine

- For theoretical analysis, we assume RAM model for our “theoretical” computer



- Our RAM model consists of:
  - a **CPU**
  - a 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.



Primitive Operations

- For theoretical analysis, we will count **primitive** or **basic** operations, which are simple computations performed by an algorithm
- Basic operations are:
  - 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

Primitive Operations

- Examples of primitive operations:
  - Evaluating an expression x^2+e^y
  - Assigning a value to a variable $cnt \leftarrow cnt+1$
  - Indexing into an array A[5]
  - Calling a method mySort(A,n)
  - Returning from a method return(cnt)

Counting Primitive Operations

- 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***)	
currentMax $\leftarrow$ ***A***[0]	2
for ***i*** $\leftarrow$ 1 **to** ***n*** $-$1 **do**	2+***n***
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	7***n*** $-$1

Estimating Running Time

- Algorithm ***arrayMax*** executes $7\boldsymbol{n} - 1$ primitive operations in the worst case. Define:
  - $\boldsymbol{a}$ = Time taken by the fastest primitive operation
  - $\boldsymbol{b}$ = Time taken by the slowest primitive operation
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax.*** Then

$$\boldsymbol{a}\,(7\boldsymbol{n} - 1) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(7\boldsymbol{n} - 1)$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ 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 $T(n)$
- Thus we focus on the big-picture which is the **growth rate** of an algorithm
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***
  - algorithm ***arrayMax*** grows proportionally with n, with its true running time being n times a constant factor that depends on the specific computer

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
- How do we get rid of the constant factors to focus on the essential part of the running time?

Big-Oh Notation Motivation

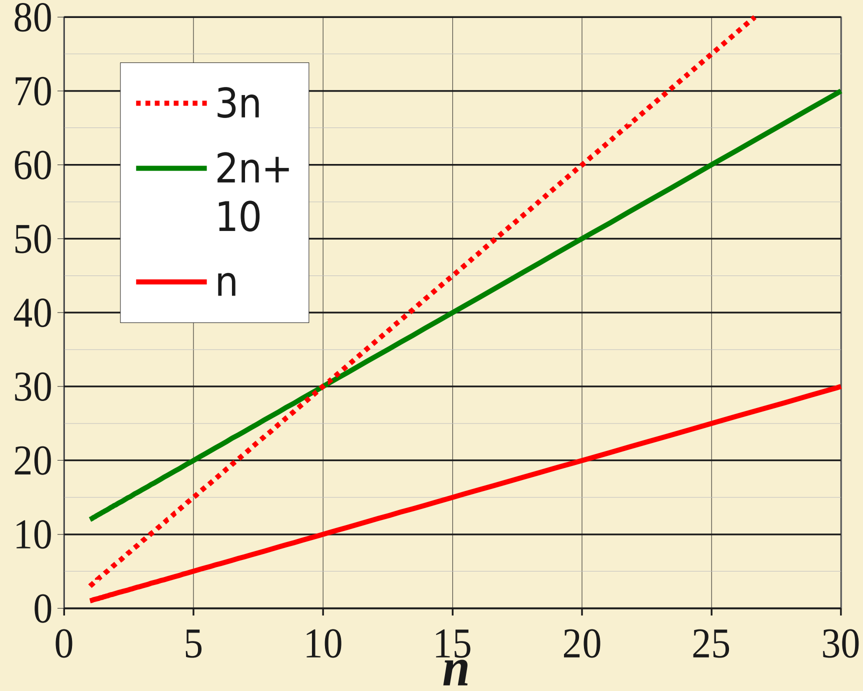
- The big-Oh notation is used widely to characterize running times and space bounds
- The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function which affect its growth

Big-Oh Notation Definition

- 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

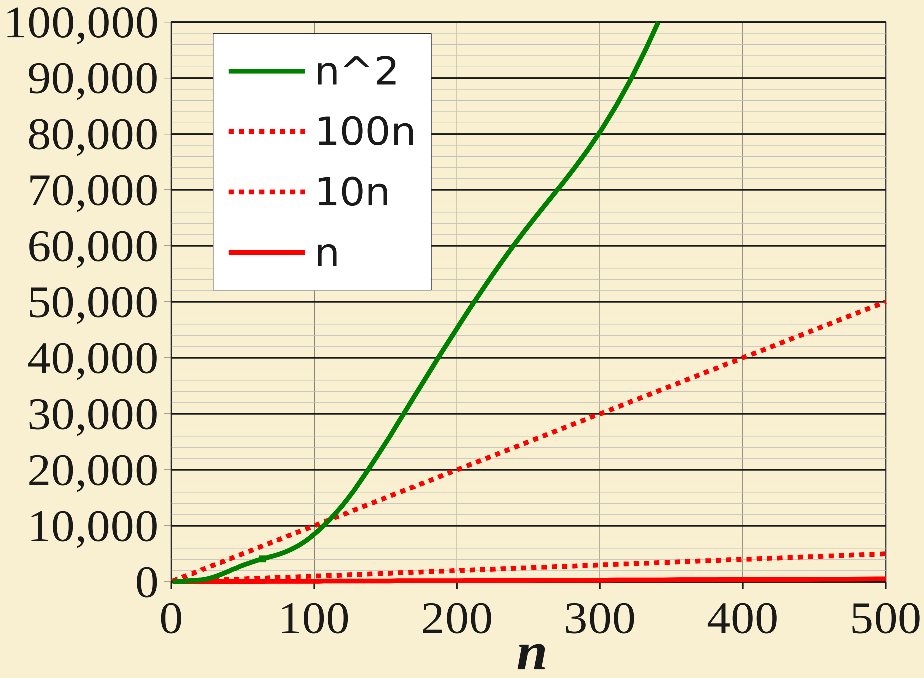
  $f(n) \le cg(n)$ for $n \ge n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c-2)\, n \ge 10$
  - $n \ge 10/(c-2)$
  - 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 $n_0 \geq 1$ s.t. $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ 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 $n_0 \geq 1$ s.t. $3 \log n + 5 \leq c \bullet \log n$ 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 $O(3n)$"

Big-Oh Rules

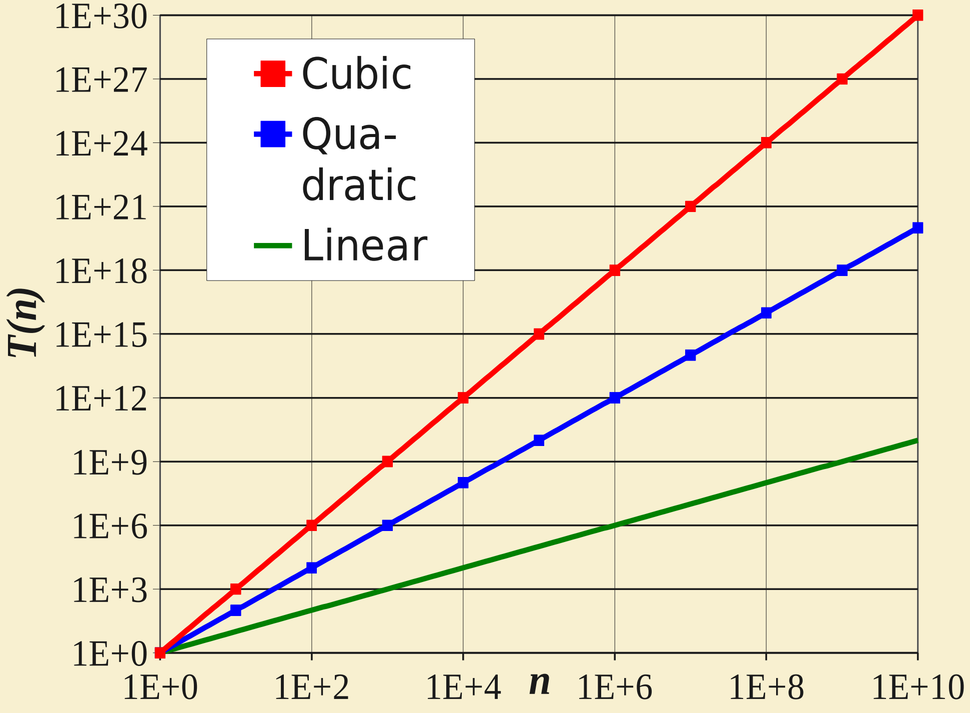
- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" 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 $7n-1$ 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

Seven Important Functions

- 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



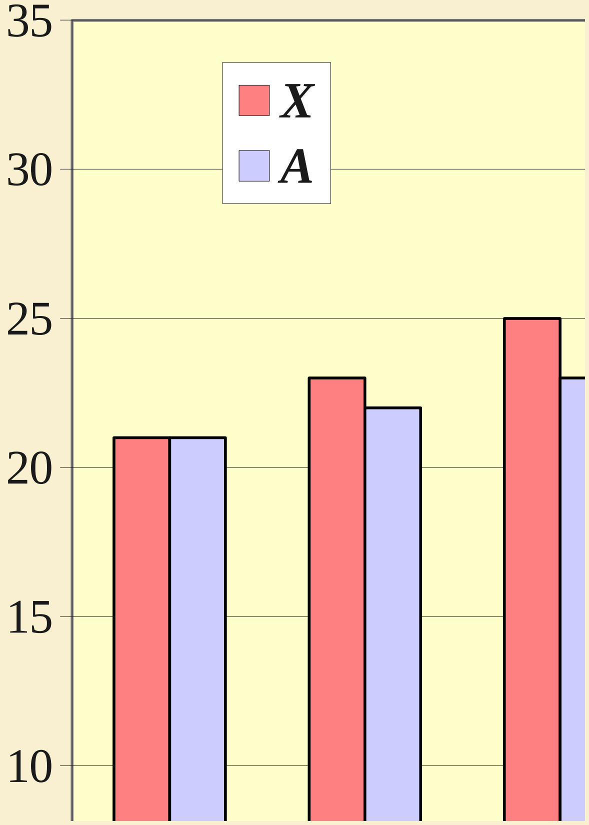
- In a log-log chart, the slope of the line corresponds to the growth rate of the function

Computing Prefix Averages

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

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

- Computing the array $\boldsymbol{A}$ of prefix averages of another array $\boldsymbol{X}$ has applications to financial analysis



Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm ***prefixAverages1***(***X, n***)

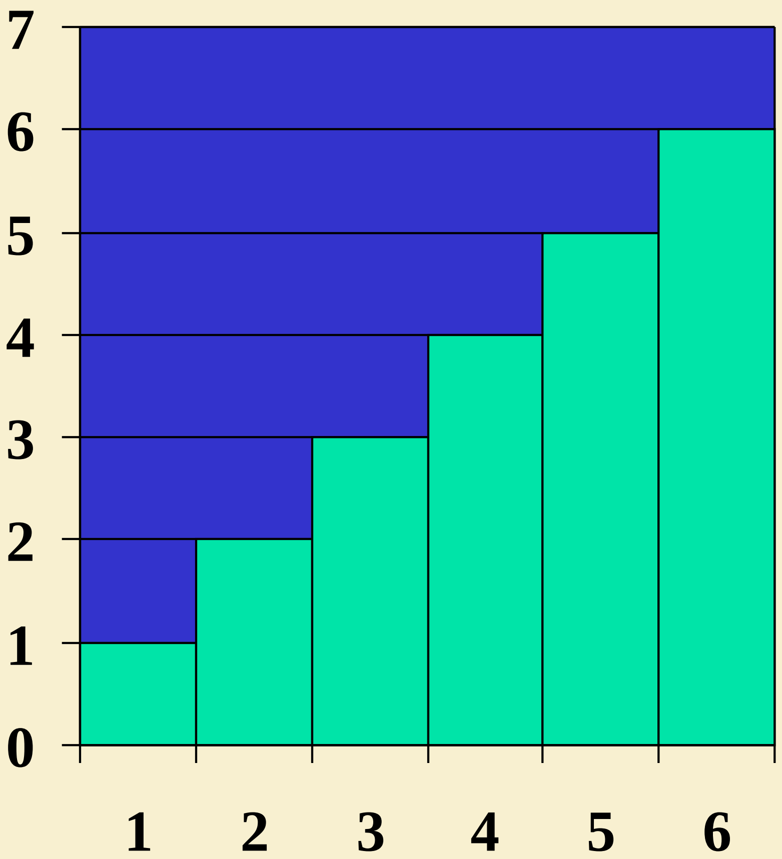
Input array ***X*** of ***n*** integers

Output array ***A*** of prefix averages of ***X***

	#operations
A ← new array of ***n*** integers	***n***
for ***i*** ← 0 **to** ***n*** −1 **do**	***n***
s ← ***X***[0]	***n***
for ***j*** ← 1 **to** ***i*** **do**	1 + 2 + …+ (***n*** −1)
s ← ***s*** + ***X***[***j***]	1 + 2 + …+ (***n*** −1)
A[***i***] ← ***s*** / (***i*** + 1)	***n***
return ***A***	1

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm ***prefixAveragesZ***(***X, n***)	
Input array ***X*** of ***n*** integers	
Output array ***A*** of prefix averages of ***X***	
	#operations
$\boldsymbol{A} \leftarrow$ new array of $\boldsymbol{n}$ integers	$\boldsymbol{n}$
$\boldsymbol{s} \leftarrow 0$	1
for $\boldsymbol{i} \leftarrow 0$ **to** $\boldsymbol{n} - 1$ **do**	$\boldsymbol{n}$
$\quad \boldsymbol{s} \leftarrow \boldsymbol{s} + \boldsymbol{X}[\boldsymbol{i}]$	$\boldsymbol{n}$
$\quad \boldsymbol{A}[\boldsymbol{i}] \leftarrow \boldsymbol{s} / (\boldsymbol{i} + 1)$	$\boldsymbol{n}$
return $\boldsymbol{A}$	1

- Algorithm ***prefixAverages2*** runs in $O(n)$ time

More Examples

Algorithm ***SumTripleArray***(X, n)

Input triple array ***X*[ ][ ][ ]** of ***n by n by n*** integers

Output sum of elements of ***X*** #operations

$s \leftarrow 0$ 1

for $i \leftarrow 0$ **to** $n-1$ **do** n

for $j \leftarrow 0$ **to** $n-1$ **do** $n+n+\ldots+n=n^2$

for $k \leftarrow 0$ **to** $n-1$ **do** $n^2+n^2+\ldots+n^2 = n^3$

$s \leftarrow s + X[i][j][k]$ $n^2+n^2+\ldots+n^2 = n^3$

return s 1

- Algorithm ***SumTripleArray*** runs in $O(n^3)$ time

Useful Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

- If $\mathbf{d(n)}$ is $O(\mathbf{f}(n))$ and $e(\mathbf{n})$ is $O(g(n))$ then
  - $\mathbf{d(n)}+e(\mathbf{n})$ is $O(\mathbf{f}(n)+\mathbf{g}(n))$
  - $\mathbf{d(n)}e(\mathbf{n})$ is $O(f(n)\,\mathbf{g}(n))$
- If $\mathbf{d(n)}$ is $\mathbf{O(f(n))}$ and $\mathbf{f(n)}$ is $\mathbf{O(g(n))}$ then $\mathbf{d(n)}$ is $\mathbf{O(g(n))}$
- If $p(n)$ is a polynomial in n then $\log p(n)$ is $O(\log(n))$

Relatives of Big-Oh

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)
- Note that f(n) is Ω(g(n)) if and only if g(n) is O(f(n))

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)
- Note that f(n) is Θ(g(n)) if and only if if g(n) is O(f(n)) **and** if f(n) is O(g(n))

Example Uses of the Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \leq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Math you need to Review

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**

    $\log_b(xy) = \log_b x + \log_b y$

    $\log_b (x/y) = \log_b x - \log_b y$

    $\log_b xa = a\log_b x$

    $\log_b a = \log_x a/\log_x b$
  - **properties of exponentials**:

    $a^{(b+c)} = a^b a^c$

    $a^{bc} = (a^b)^c$

    $a^b /a^c = a^{(b-c)}$

    $b = a^{\log_a b}$

    $b^c = a^{c*\log_a b}$

Final Notes

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally
- Suppose we have 2 algorithms
  - Algorithm A has running time 30000n
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster