MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

16 Algorithms Analysis (cont.)

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

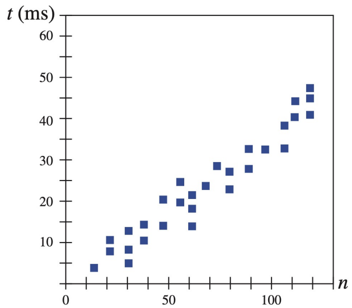
February 28, 2022

Administration

- Please pay special attention to this week's tutorial
- Start the assignment as early as possible. I will not be able to answer questions during coming weekend!
  - If you have questions, ask early.

Review

- Runtime Complexity Analysis
  - Experimental approach
    - Implement, run with varying input sizes, and measure run-times



  - Theoretical Approach
    - Allows us to evaluate the **relative efficiency** of any two algorithms **independent of** the hardware/software environment
    - For each algorithm, we will end up with a function **f(n)** that characterizes the running time of the algorithm as a function of the input size **n.**
    - We started by looking at primitive operations to get an idea of what operations an algorithm perform on input

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$

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

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$\boxed{currMax \leftarrow A[0]}$ <-------- 2 operations

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

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

1. accessing A[0] (indexing in array)
2. assigning A[0] to currMax

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

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

$currMax \leftarrow A[0]$ <-------- 2 operations
$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

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

$currMax \leftarrow A[0]$ <-------- 2 operations
$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

The for loop repeats **n** times, why?
Each time it has **2** operations, why?
each time it involves an **assignment** and a **comparison**

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration

for example: n = 4

for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

$\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$ <----- 2n operations

  if $currMax < A[i]$ **then**

    $currMax \leftarrow A[i]$

return $currMax$

The for loop will repeat **n** times, why?

We account for the last increment to **n** in which the for loop identifies it should exit before entering next iteration

for example: n = 4

for ***i*** from ***1*** to ***3*** ➔ in the last iteration ***i*** becomes ***4*** and will be compared to ***3***

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations

  if $currMax < A[i]$ **then** <-- 2(n-1) operations

    $currMax \leftarrow A[i]$ <-- 2(n-1) operations

  i++ <-- 2(n-1) operations

return $currMax$

The body of for loop will repeat **n-1** times

The increment of ***i*** is performed at the end of each iteration

Primitive Operations

- We define a set of primitive operations such as the following:
  - Assigning a value to a variable
  - Calling a function
  - Performing an arithmetic operation
  - Comparing two numbers
  - Indexing into an array
  - Following an object reference
  - Returning from a function

Algorithm arrayMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

$currMax \leftarrow A[0]$ <-------- 2 operations

for $i \leftarrow 1$ **to** $n-1$ **do** <----- 2n operations

  if $currMax < A[i]$ **then** <-- 2(n-1) operations

    $currMax \leftarrow A[i]$ <-- 2(n-1) operations

  i++ <-- 2(n-1) operations

return $currMax$ <-- 1 operation

We will have a total of 8n - 3 operations

n is the input size!

Estimating Runtime

- Algorithm **arrayMax** executes 8n - 3 primitive operations in total

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

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

- Suppose:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation

- The running time of **arrayMax** is bounded by two linear functions
  - **a**(8n - 3) <= **T(n)** <= **b**(8n - 3)
- 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**

Asymptotic Notation

- Big-picture approach
  - In algorithm analysis, we focus on the growth rate of the running time as a function of the input size **n**.
  - It is often enough just to know that the running time of an algorithm such as arrayMax, grows proportionally to **n**, with its true running time being **n** times a **constant factor** that depends on the specific computer. (was 8n - 3)
  - We characterize the running times of algorithms by using functions that map the size of the input, **n**, to values that correspond to the main factor that determines the growth rate in terms of **n**.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

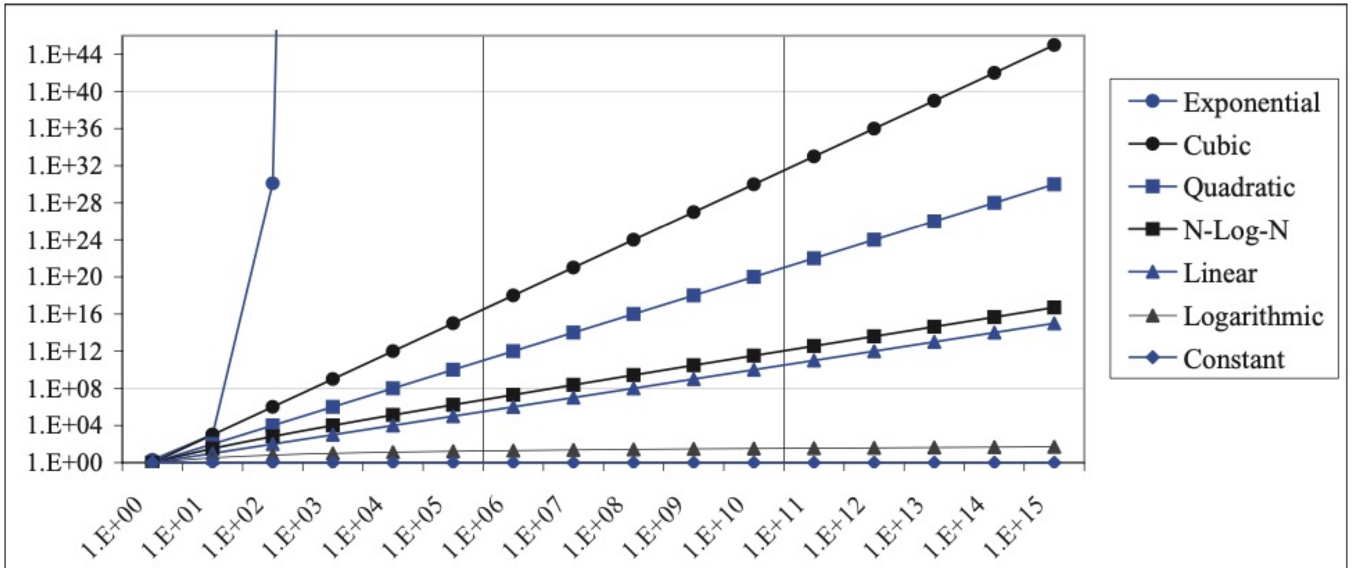
Why Growth Rate Matters

- **Classes of functions**

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

- Ideally, we would like:
  - data structure operations to run in times proportional to the **constant** or **logarithm** function.
  - algorithms to run in linear or n-log-n time.
  - Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the small-sized inputs.

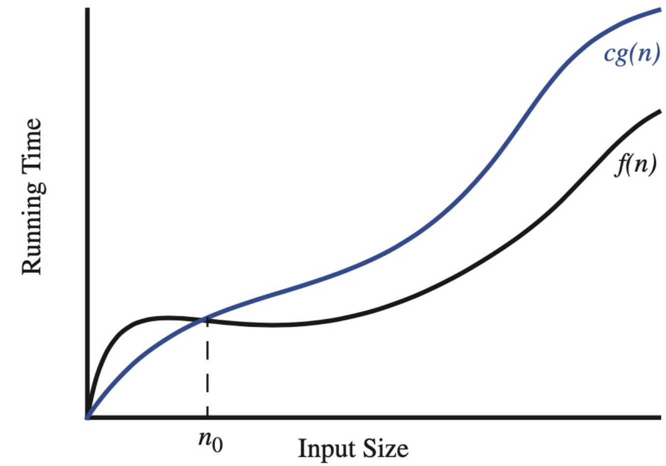
log-log chart

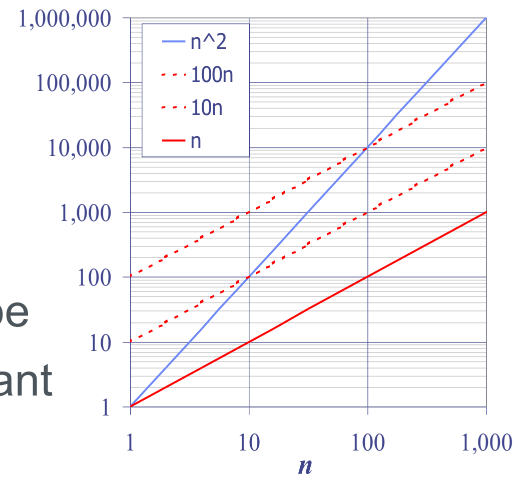


Growth rates for the seven fundamental functions used in algorithm analysis

Big-Oh Notation

- 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 $n_0 = 10$
- Example: ArrayMax: $8n - 3$ is $O(n)$
  - $8n - 3 \le cn$
  - Pick $c = 8$, and $n_0 = 1$
- Example: n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant





Asymptotic Analysis of Algorithms

- Now we can write the following mathematically precise statement on the running time of algorithm arrayMax for ***any*** computer:
  - The Algorithm arrayMax, for computing the maximum element in an array of **n** integers, runs in **O(n)** time.
  - proof: The number of primitive operations executed by algorithm **arrayMax** in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm **arrayMax** on an input of size **n** is **at most a constant times n**, that is, we may conclude that the **running time of algorithm arrayMax is O(n)**.
- The asymptotic analysis
  - identify the running time in Big-Oh notation
  - 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

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
$currMax \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $currMax < A[i]$ **then**
    $currMax \leftarrow A[i]$
return $currMax$

Big-Oh Notation Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$:
  - Drop lower-order terms
  - Drop constant factors
    - $3n^3 + 20n^2 + 5$ 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 $n_0 = 21$
- 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 $O(3n)$"
- Think about hidden constant factors!

The big-Oh notation gives an upper bound on the growth rate of a function

Asymptotic Analysis - Example

- Prefix Averages
- The i-th prefix average of an array X is average of the first $(i+1)$ elements of X:
  $A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

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 $X[0], \ldots, X[i]$.

Let A be an array of n numbers.
for $i \leftarrow 0$ **to** $n-1$ **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  $A[i] \leftarrow a/(i+1)$
return array A

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 $X[0], \ldots, X[i]$.

Let A be an array of n numbers.
$s \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $s \leftarrow s + X[i]$
  $A[i] \leftarrow s/(i+1)$
return array A

Questions?