MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

17 Algorithms Analysis (cont.2)

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

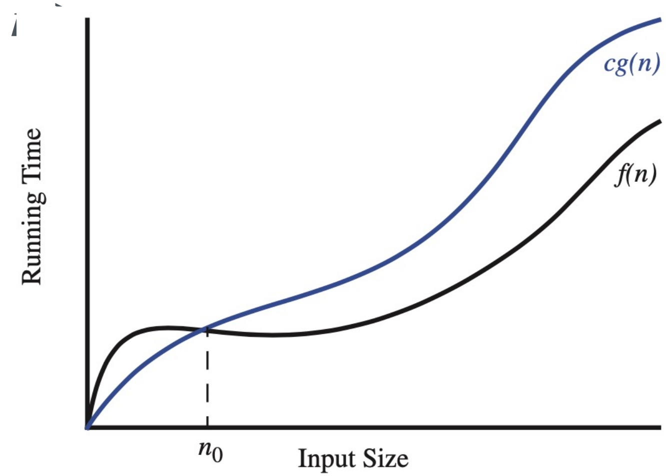
March 2, 2022

Administration

- My Office Hour:
  - Today at 15:00 in ITB-159 in-person (or virtually using teams)
- Please **read the questions carefully** and watch the video if needed for the assignment 2

Review

- 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
- Example: $2n + 10$ is $O(n)$
- Example: ArrayMax: $8n - 3$ is $O(n)$
- Example: n^2 is <u>not</u> $O(n)$



- Known class of functions:

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

- Desired complexities:
  - Linear or n-log-n for algorithms
    - sort, search
  - Constant or Logarithm operations for DS
    - add, remove, indexing

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 <u>constant</u>. 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 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    sequence of statements
}
```

is O(n)

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 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
        this is true for $c = 4$ and $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sequence of statements
    }
}
```

is $O(n^2)$

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 $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < i; j++) {
        sequence of statements
    }
}
```

- The outer loops is O(n)
- The statements in the inner loop executed i+1 times: i.e.: 1 + 2 + 3 + … + n times which is n(n+1)/2 which is $O(n^2)$

So, this is $O(n^2)$

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 $n_0 = 21$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sequence of statements
    }
}
for (k = 0; k < n; k++) {
    sequence of statements
}
```

is $O(n^2+n)$ which is $O(n^2)$
we select the main component that affects the growth

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 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
        this is true for $c = 4$ and $n_0 = 21$

```
for (j = 0; j < n; j++)
    f(j);
```

is O(n)

- function ***f*** takes a constant time

```
for (j = 0; j < n; j++)
    g(j);
```

is $O(n^2)$

- function ***g*** takes a linear time proportional to its parameter

```
for (j = 0; j < n; j++)
    g(k);
```

is $O(k \cdot n)$
or O(n), if ***k*** is not very large or has a relative size to ***n***

Big-Oh Notation Rules

- 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

Complex Examples on Growth Rates

- You need some math to reason about some functions
- math you need to review
  - properties of logarithms:
    - $\log_b(x\,y) = \log_b x + \log_b y$
    - $\log_b (x / y) = \log_b x - \log_b y$
    - $\log_b x^a = a \log_b x$
    - $\log_b a = \log_x a / \log_x b$
  - properties of exponentials:
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{\log_a b}$
    - $b^c = a^{c \log_a b}$

for example:

$$(\sqrt{2})^{\log n} = (2^{0.5})^{\log n} = (2^{\log n})^{0.5} = \sqrt{n}$$

$$2^{100 \log n} = 2^{\log n^{100}} = n^{100}$$

$$\log(n!) = \log(\mathrm{n}(\mathrm{n}-1)(\mathrm{n}-2)\ldots) =$$
$$= \log(n) + \log(n-1) + \cdots =$$
is $O(\log(n))$

Asymptotic Analysis - Example

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

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

Input Array X:

0	1	2	3	4
2	8	5	13	7

Prefix Average Array A:

0	1	2	3	4
2	5	5	7	7

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

	0	1	2	3	4
X	2	8	5	13	7

	0	1	2	3	4
A	2				

i = 0
a = 0
after inner for loop (i + 1 times)
a = 2
A[0] = 2 / 1 = 2

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

	0	1	2	3	4
X	2	8	5	13	7

	0	1	2	3	4
A	2	5			

i = 1
a = 0
after inner for loop (i + 1 times)
a = 10
A[0] = 10 / 2 = 5

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

	0	1	2	3	4
X	2	8	5	13	7

	0	1	2	3	4
A	2	5	5		

i = 2
a = 0
after inner for loop (i + 1 times)
a = 15
A[0] = 15 / 3 = 5

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

	0	1	2	3	4
X	2	8	5	13	7

	0	1	2	3	4
A	2	5	5	7	7

Asymptotic Analysis - Example

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

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

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. <------------------ O(n)

for $i \leftarrow 0$ **to** $n-1$ **do** <---------------------------------O(n) operations

    $a \leftarrow 0$ <----------------------------O(n) operations

    for $j \leftarrow 0$ **to** i **do** <-- O(n^2) operations

        $a \leftarrow a + X[j]$ <-- O(n^2) operations

    $A[i] \leftarrow a/(i+1)$ <------------------ O(n) operations

return array A <-- O(1) operation (in the textbook it is O(n) considering return by value)

so prefixAverages1 is O(n^2)

Asymptotic Analysis - Example

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

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

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

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

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

Asymptotic Analysis - Example

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

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

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

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. <------------------ O(n)

$s \leftarrow 0$ <----------------------------O(1) operations

for $i \leftarrow 0$ **to** $n-1$ **do** <--- O(n) operations

  $s \leftarrow s + X[i]$ <------- O(n) operations

  $A[i] \leftarrow s/(i+1)$ <----- O(n) operations

return array A <-- O(1) operation (in the textbook it is O(n) considering return by value)

so prefixAverages2 is O(n)

Using Sorting as a Problem-Solving Tool

- Sometimes we can use sorting as a tool to ease our problem-solving and even make the algorithm faster
- Element uniqueness problem:
  - Given an array, identify if all elements are unique (there are no duplicates)
    - Naïve algorithm:
      - compare each pair of elements in the array
        - $O(n^2)$
    - Better algorithm
      - sort the array first, then traverse the array once, look for duplicates among consecutive elements.
        - Best sorting algorithm runs in $O(n\log n)$
        - The algorithm will be $O(n\log n + n)$ which is $(n\log n)$

Final Remarks

- Given an algorithm
  - What is the order of this algorithm? (runtime complexity)
- Try to reduce the running time as much as possible
  - Sometimes based on some observations that we already had we can easily reduce the complexity (like previous slide)
- This analysis technique is not the whole story!
  - There are situations where a worst-case $O(n^2)$ is faster than $O(n\log n)$
  - If you want, you can study more.
- There are some problems for which there is **NO** polynomial-time algorithm found (up until now)
  - We say that they “NP-hard” or “NP-complete”
  - If someone can find a polynomial-time solution one of them, that solution may work for many others as well.

Questions?