Lecture 3: Θ, Big-Ω and the RAM Model

COMS10007 - Algorithms

Dr. Christian Konrad

03.02.2020

Limitations/Strengths of Big-O

O-notation: Upper Bound

- Runtime $O(f(n))$ means on any input of length n the runtime is bounded by some function in $O(f(n))$
- If runtime is $O(n^2)$, then the actual runtime could also be in $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n\sqrt{n})$, etc...

This is a Strong Point:

- Worst case running time: A runtime of $O(f(n))$ guarantees that algorithm won't be slower, but may be faster
- Example: FAST-PEAK-FINDING often faster than $5 \log n$

How to Avoid Ambiguities

- Θ-notation: Growth is precisely determined up to constants
- Ω-notation: Gives us a lower bound

Θ-notation

"Theta"-notation:
Growth is precisely determined up to constants

Definition: Θ-notation ("Theta")

Let $g : \mathbb{N} \rightarrow \mathbb{N}$ be a function. Then $\Theta(g(n))$ is the set of functions:

$$\Theta(g(n)) = \{f(n) : \text{There exist positive constants } c_1, c_2 \text{ and } n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

$f \in \Theta(g)$: "f is asymptotically sandwiched between constant multiples of g"

Symmetry of Θ

Lemma

The following statements are equivalent:

1. $f \in \Theta(g)$
2. $g \in \Theta(f)$

Proof. Suppose that $f \in \Theta(g)$. We need to prove that there are positive constants C_1, C_2, N_0 such that

$$0 \leq C_1 f(n) \leq g(n) \leq C_2 f(n), \text{for all } n \geq N_0 \ . \tag{1}$$

Since $f \in \Theta(g)$, there are positive constants c_1, c_2, n_0 s.t.

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{for all } n \geq n_0. \tag{2}$$

Setting $C_1 = \frac{1}{c_2}$, $C_2 = \frac{1}{c_1}$, and $N_0 = n_0$, then (1) follows immediately from (2). Reverse direction is equivalent. □

Further Properties of Θ

More on Theta

Lemma (Relationship between Θ and Big-O)

The following statements are equivalent:

1. $f \in \Theta(g)$
2. $f \in O(g)$ *and* $g \in O(f)$

Proof. $\rightarrow$ Exercise.

Runtime of Algorithm in $\Theta(f(n))$?

- Only makes sense if alg. *always* requires $\Theta(f(n))$ steps, i.e., both *best-case* and *worst-case* runtime are $\Theta(f(n))$
- This is not the case in FAST-PEAK-FINDING
- However, correct to say that worst-case runtime of alg. is $\Theta(f(n))$

Ω-notation

Big Omega-Notation:

Definition: Ω-notation ("Big Omega")

Let $g : \mathbb{N} \rightarrow \mathbb{N}$ be a function. Then $\Omega(g(n))$ is the set of functions:

$$\Omega(g(n)) = \{f(n) : \text{There exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$f \in \Omega(g)$: *"f grows asymptotically at least as fast as g up to constants"*

Properties of Ω

Lemma

The following statements are equivalent:

1. $f \in \Omega(g)$
2. $g \in O(f)$

Proof. $\rightarrow$ Exercise.

Examples: Big Omega

- $10n^2 \in \Omega(n)$
- $6^{n \log n} \in \Omega(n^8)$
- Reverse examples for Big-O to obtain more examples

Runtime of Algorithm in $\Omega(f)$?

Only makes sense if best-case runtime is in $\Omega(f)$

Using O, Ω, Θ in Equations

Notation

- O, Ω, Θ are often used in equations
- $\in$ is then replaced by $=$

Examples

- $4n^3 = O(n^3)$
- $n + 10 = n + O(1)$
- $10n^2 + 1/n = 10n^2 + O(1)$

Observe

- Sloppy but very convenient
- When using O, Θ, Ω in equations then details get lost
- This allows us to focus on the essential part of an equation
- Not reversible! E.g., $n + 10 = n + O(1)$ but $n + O(1) \neq n + 10$...

The RAM Model

Algorithms

Muhammad ibn Musa **al-Khwarizmi**
~ 780 - ~ 850
($\approx$ Algorithms)

What is an Algorithm?

- Computational procedure to solve a computational problem
- Mathematical abstraction of a computer programme

Discussion Points?

- Which individual steps can an algorithm do?
  Depends on computer, programming language, ...
- How long do these steps take?
  Depends on computer, compiler optimization, ...

Models of Computation

Real Computers are complicated

Memory hierachy, floating point operations, garbage collector, how long does x^y take?, compiler optimizations, different programming languages, ...

Models of Computation:

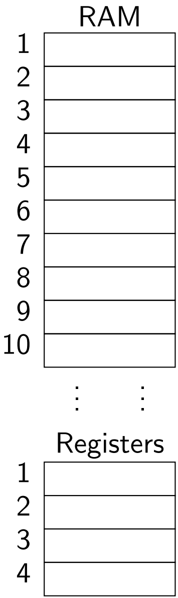
- Simple abstraction of a Computer
- Defines the "Rules of the Game":
  - Which operations is an algorithm allowed to do?
  - What is the cost of each operation?
  - Cost of an algorithm $= \sum$ cost of all its operations

See also: **COMS11700 Theory of Computation**

RAM Model

RAM Model: Random Access Machine Model

- Infinite Random Access Memory (an array), each cell has a unique address
- Each cell stores one *word*, e.g., an integer, a character, an address, etc.
- **Input:** Stored in RAM
- **Output:** To be written into RAM
- A finite (constant) number of registers (e.g., 4)



In a single Time Step we can:

- Load a word from memory into a register
- Compute $(+, -, *, /)$, bit operations, comparisons, etc. on registers
- Move a word from register to memory

RAM Model (2)

Algorithm in the RAM Model
Sequence of elementary operations (similar to assembler code)

Example: Compute the sum of two integers

- Assume that $M[0]$ and $M[1]$ contain the integers
- Write output to position $M[2]$

Cost of an Algorithm:

- **Runtime:** Total number of elementary operations
- **Space:** Total number of memory cells used (excluding the cells that contain the input)

Assumption:

- Input for algorithm is stored on read-only cells
- This space is not accounted for

Specifying an Algorithm

How to specify an Algorithm

- We specify algorithms using pseudo code or English language
- **We however always bear in mind that every operation of our algorithm can be implemented in $O(1)$ elementary operations in the RAM model**
- O-notation gives us the necessary flexibility for a meaningful definition of runtime

Exercise: How to implement in RAM model?

Require: Array of n integers A
  $S \leftarrow 0$
  for $i = 0, \dots, n-1$ **do**
    $S \leftarrow S + A[i]$
  return S

Notions of Runtime

- **Runtime on a specific input**
  Given a specific input X, how many elementary operations does the algorithm perform?
- **Worst-case**
  Consider the set of all inputs of length n. What is the maximum number of elementary operations the algorithm performs when run on all inputs of this set?
- **Best-case**
  Consider the set of all inputs of length n. What is the minimum number of elementary operations the algorithm performs when run on all inputs of this set?
- **Average-case**
  Consider a set of inputs (e.g. the set of all inputs of length n). What is the average number of elementary operations the algorithm performs when run on all inputs of this set?

$$\text{Best-case} = O(\text{Average-case}) = O(\text{Worst-case})$$

Runtime/Space Analysis of Algorithms

Runtime/Space Analysis

Goals:

- **Runtime:** Count number of elementary operations when implemented in RAM model
- **Space:** Count number of cells used when implemented in RAM model

However...

- Algorithms are usually not stated to run in RAM model
- We would like to state and analyze our algorithms in pseudo code (or a programming language, natural language, ...)

Solution:

- Analyze algorithm as specified in pseudo code directly
- Make sure that every instruction can be implemented in the RAM model using $O(1)$ elementary operations

Example

Require: Integer array A of length n

$s \leftarrow 0$
for $i \leftarrow 0 \dots n-1$ **do**
  $s \leftarrow s + A[i]$
return s

Example

Require: Integer array A of length n

$s \leftarrow 0$
for $i \leftarrow 0 \dots n-1$ **do**
  $s \leftarrow s + A[i]$
return s

Example

Require: Integer array A of length n

$s \leftarrow 0$ **$O(1)$**

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

$s \leftarrow s + A[i]$

return s

Example

Require: Integer array A of length n

$s \leftarrow 0$ **O(1)**

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

$s \leftarrow s + A[i]$

return s

Example

Require: Integer array A of length n

```
s ← 0                          O(1)
for i ← 0 ... n − 1 do
    s ← s + A[i]               O(1)
return s
```

Example

Require: Integer array A of length n

$s \leftarrow 0$ $\mathbf{O(1)}$

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

  $s \leftarrow s + A[i]$ $\mathbf{O(1)}$

return s

Example

```
Require: Integer array A of length n
  s ← 0                          O(1)
  for i ← 0 ... n − 1 do
    s ← s + A[i]                 O(1)
  return s                       O(1)
```

Example

Require: Integer array A of length n

$s \leftarrow 0$ **O(1)**

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

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

return s **O(1)**

Example

Require: Integer array A of length n

$s \leftarrow 0$	$\mathbf{O(1)}$
for $i \leftarrow 0 \ldots n-1$ **do**	**n times**
$\quad s \leftarrow s + A[i]$	$\mathbf{O(1)}$
return s	$\mathbf{O(1)}$

Example

Require: Integer array A of length n

$s \leftarrow 0$	$\mathbf{O(1)}$
for $i \leftarrow 0 \ldots n-1$ **do**	**n times**
$\quad s \leftarrow s + A[i]$	$\mathbf{O(1)}$
return s	$\mathbf{O(1)}$

Runtime: $O(1) + n \cdot O(1) + O(1) = O(1) + O(n) + O(1) = O(n)$.

Example 2

Require: Integer array A of length n

```
s ← 0
for i ← 0 ... n − 1 do
    for j ← i ... 2i do
        s ← s + A[i]
return s
```

Example 2

```
Require: Integer array A of length n
  s ← 0                                   O(1)
  for i ← 0 ... n − 1 do
    for j ← i ... 2i do
      s ← s + A[i]                        O(1)
  return s                                O(1)
```

Example 2

Require: Integer array A of length n

```
s ← 0                          O(1)
for i ← 0 ... n − 1 do
    for j ← i ... 2i do
        s ← s + A[i]           O(1)
return s                       O(1)
```

Example 2

Require: Integer array A of length n

$s \leftarrow 0$	$\mathbf{O(1)}$
for $i \leftarrow 0 \ldots n-1$ **do**	
for $j \leftarrow i \ldots 2i$ **do**	$\mathbf{i+1}$ **times**
$s \leftarrow s + A[i]$	$\mathbf{O(1)}$
return s	$\mathbf{O(1)}$

Example 2

Require: Integer array A of length n

```
s ← 0                          O(1)
for i ← 0 ... n − 1 do
    for j ← i ... 2i do        i + 1 times
        s ← s + A[i]           O(1)
return s                       O(1)
```

Example 2

Require: Integer array A of length n	
$s \leftarrow 0$	$\mathbf{O(1)}$
for $i \leftarrow 0 \ldots n-1$ **do**	**n times**
for $j \leftarrow i \ldots 2i$ **do**	$\mathbf{i + 1}$ **times**
$s \leftarrow s + A[i]$	$\mathbf{O(1)}$
return s	$\mathbf{O(1)}$

Example 2

Require: Integer array A of length n

$s \leftarrow 0$	$\mathbf{O(1)}$
for $i \leftarrow 0 \ldots n-1$ **do**	**n times**
for $j \leftarrow i \ldots 2i$ **do**	$\mathbf{i+1}$ **times**
$s \leftarrow s + A[i]$	$\mathbf{O(1)}$
return s	$\mathbf{O(1)}$

Runtime:

$$O(1) + \sum_{i=0}^{n-1}((i+1) \cdot O(1)) + O(1) = O(1) + O(1)\sum_{i=0}^{n-1}(i+1)$$

$$= O(1) + O(1)\sum_{i=1}^{n} i = O(1) + O(1)\frac{n(n+1)}{2}$$

$$= O(1) + O(\frac{n^2}{2} + \frac{n}{2}) = O(1) + O(n^2) = O(n^2) \ .$$

Example 3

Algorithm: Given is an integer array of length n. Run through the array from left to right and maintain the minimum seen so far.

Runtime: $O(n)$