Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

DS286 | 2016-08-31,09-02

L6,7: Time & Space Complexity

Yogesh Simmhan

simmhan@cds.iisc.ac.in

Slides courtesy Venkatesh Babu, CDS

CDS
The Department of Computational and Data Science

Algorithm Analysis

- Algorithms can be evaluated on two performance measures
- Time taken to run an algorithm
- Memory space required to run an algorithm
- …for a given input size

- *Why are these important?*

Space Complexity

- Space complexity = Estimate of the amount of memory required by an algorithm to run to completion, for a given input size
  - Core dumps = the most often encountered cause is “memory leaks” – the amount of memory required larger than the memory available on a given system
- Some algorithms may be more efficient if data completely loaded into memory
  - Need to look also at system limitations

Space Complexity

- **Fixed part**: The size required to store certain data/variables, that is independent of the size of the problem:
  - e.g., given the definition for every word in a given book
  - Constant size for storing “dictionary”, e.g., 50MB
- **Variable part**: Space needed by variables, whose size is dependent on the size of the problem:
  - e.g., actual book size may vary, 1MB … 1GB

Analyzing Running Time

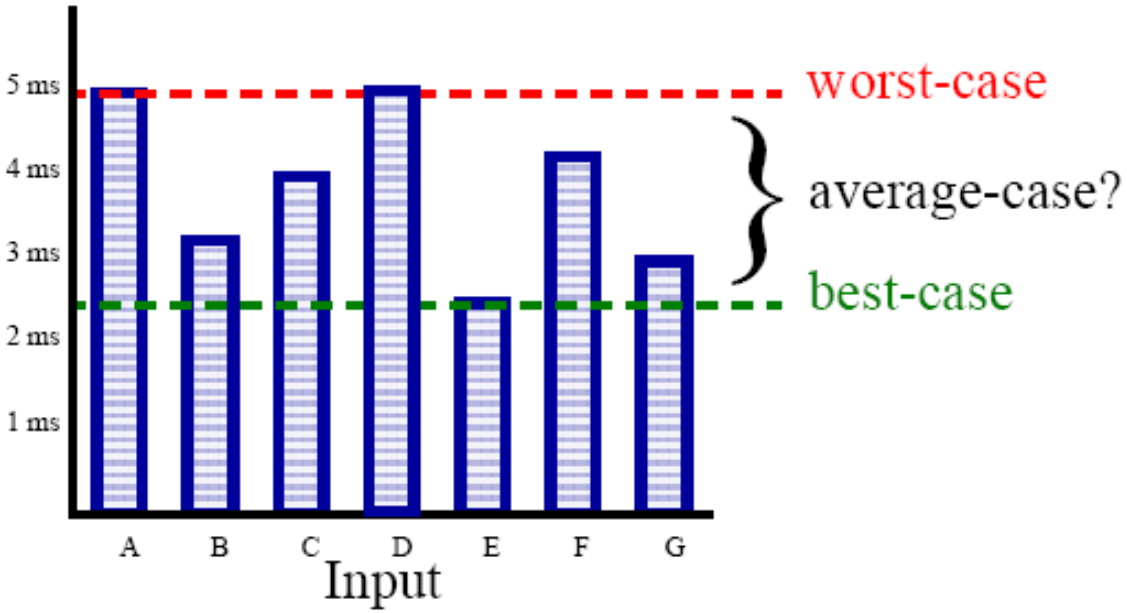
- Write program
- Run Program
- Measure actual running time with some methods like *System.currentTimeMillis(), gettimeofday()*

- *Is that good enough as a programmer?*

Limitation of Experimental Running Time Measurements

- Need to implement the algorithm.
- Cannot exhaust all possible inputs
  - Experiments can be done only on a limited to set of inputs, and may not be indicative of the running time for other inputs.
- Harder to compare two algorithms
  - Same hardware/environments to be used

Running Time



- Suppose the program includes an *if-then statement that may* execute or not → variable running time
- Typically algorithms are measured by their ***worst case***

Develop General Methodology for Analysis

- Uses High Level Description instead of implementation
- Takes into account for all possible inputs
- Allows one to evaluate the efficiency independent of hardware/software environment.

Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm

```
arrayMax(A,n)
   Max=A[0]
   for i=1 to n-1 do
      If Max < A[i] then Max = A[i]
   Return Max
```

Pseudo-Code

- More structured and less formal
- Expressions
  - Standard Math symbols (numeric/boolean)
- Method Declarations
  - **Algorithm** name(param1,param2)

Pseudo Code

- Programming Constructs:
  - If … then …else
  - While-loop
  - for-loop
  - Array : A[i]; A[I,j]
- Methods
  - Calls: method(args)
  - Returns: return value

Analysis of Algorithms

- Analyze time taken by *Primitive Operations*
- Low level operations independent of programming language
  - Data movement (assign..)
  - Control (branch, subroutine call, return…)
  - Arithmetic/logical operations (add, compare..)

- By inspecting the pseudo-code, we can **count** the number of primitive operations executed by an algorithm

Example: Array Transpose

```
function Transpose(A[][], n)
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      tmp = A[i][j]
      A[i][j] = A[j][i]
      A[j][i] = tmp
    end
  end
end
```

	j=0			*j=3*
i=0	0,0	0,1	0,2	0,3
	1,0	1,1	1,2	1,3
	2,0	2,1	2,2	2,3
i=3	3,0	3,1	3,2	3,3

Estimated time for A[n][n] = (**n(n-1)/2**).(3+2) + 2.n

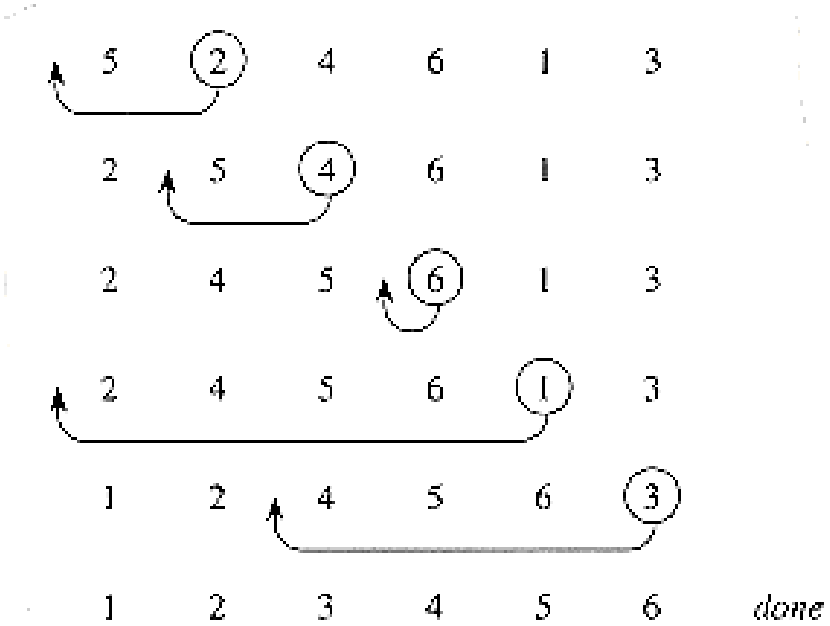
Is this constant for a given 'n'?

Example: Sorting

- Correctness:
  - For any given input the algorithm stops with the output {b1 < b2 < b3 … < bn} which is a permutation of the input {a1, a2, … an}

- Running time depends on:
  - Number of elements (n)
  - How partially sorted
  - Algorithm used

Insertion Sort



Insertion Sort

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]                          ──────────→ (N-1) times
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key              ──────────→ T_j times
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Analysis of Insertion Sort

# of Sorted Elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
…	…	…
n-1	1	n-1
	n-1	**n(n-1)/2**

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': 1001 = 1000
  - $3n^2 = n^2$
- How the running time of an algorithm increases with the size of input in the limit.
  - Asymptotically more efficient algorithms are best *for all but* small inputs.

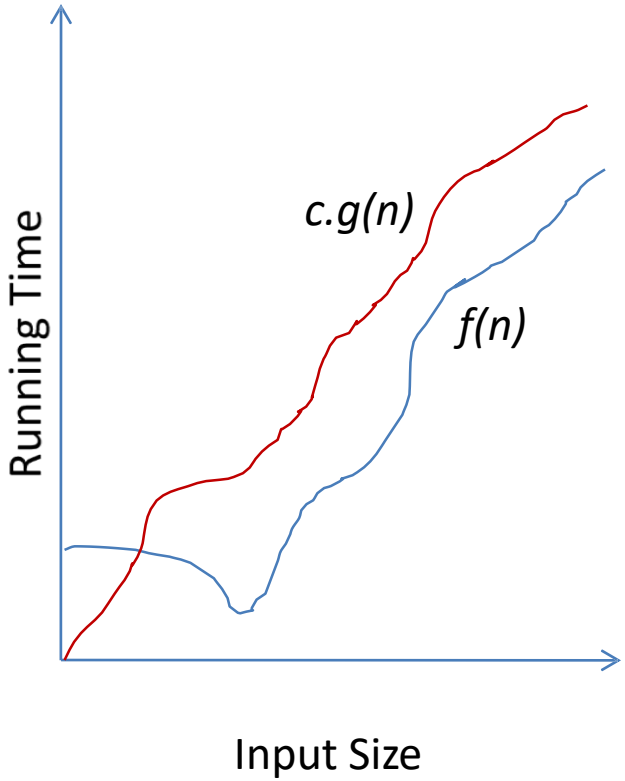
Asymptotic Notation: "Big O"

Definition 3.1 *Let $p(n)$ and $q(n)$ be two nonnegative functions. $p(n)$ is* **asymptotically bigger** *($p(n)$ asymptotically dominates $q(n)$) than the function $q(n)$ iff*

$$\lim_{n\to\infty} \frac{q(n)}{p(n)} = 0 \qquad (3.1)$$

TB, Sahni

- ***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 ≥ n_0
  - *f(n) and g(n) are functions over non negative intergers*
- Used for *worst-case* analysis
  - g(n) is the **asymptotic upper bound** of actual time taken



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors
  - $(n(n-1)/2).(3+2) + 2.n$ is $\mathbf{O(n^2)}$
  - $23.n.\log(n)$ is $\mathbf{O(n.\log(n))}$
  - $9n-6$ is $\mathbf{O(n)}$
  - $6n^2.\log(n) + 3n^2 + n$ is $\mathbf{O(n^2.\log(n))}$
- Note: It is expected that the approximation should be as small an order as possible

Asymptotic Analysis of Running Time

- Use *O* notation to express number of primitive operations executed as a function of input size.
- Hierarchy of functions

$$1 < \log n < n < n^2 < n^3 < 2^n$$

- Warning! Beware of large constants (say 1M).
  - This might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis

- Input: An array X[n] of numbers.
- Output: An array A[n] of numbers s.t A[k]=mean(X[0]+X[1]+…+X[k-1])

```
for i=0 to (n-1) do
   a=0
   for j=0 to i do
      a = a + X[j]
   end
   A[i] = a/(i+1)
end
return A
```

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

A Better Algorithm

```
s=0
for i=0 to n do
   s = s + X[i]
   A[i] = s/(i+1)
end
return A
```

- Analysis: running time is $O(n)$

Asymptotic Notation

- Special Cases of algorithms
  - Logarithmic $O(\log n)$
  - Linear $O(n)$
  - Quadratic $O(n^2)$
  - Polynomial $O(n^k)$, $k > 1$
  - Exponential $O(a^n)$, $a > 1$

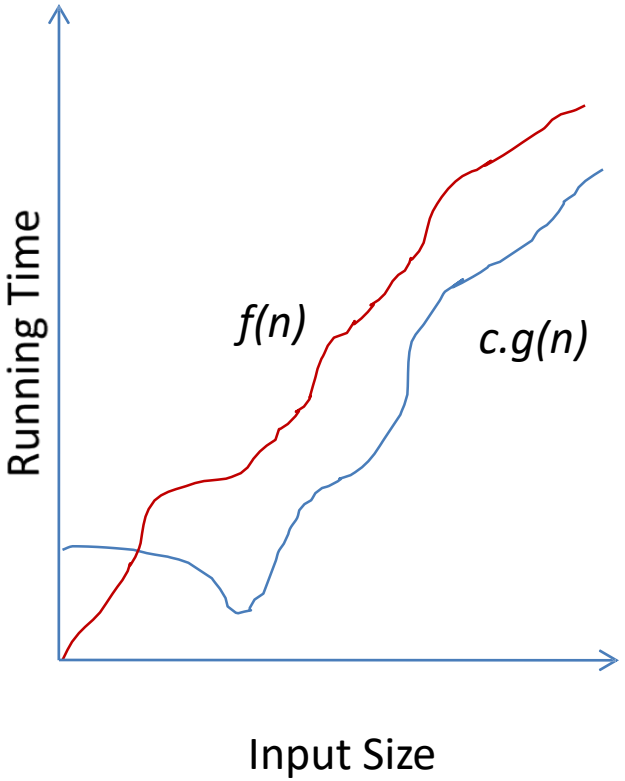
Comparison

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Asymptotic Notation: Lower Bound

- The "big-Omega" Ω notation
  - asymptotic *lower* bound
  - f(n) = Ω(g(n)) if there exists const. c and n_0 s.t.
    - **c.g(n) ≤ f(n)** for n ≥ n_0

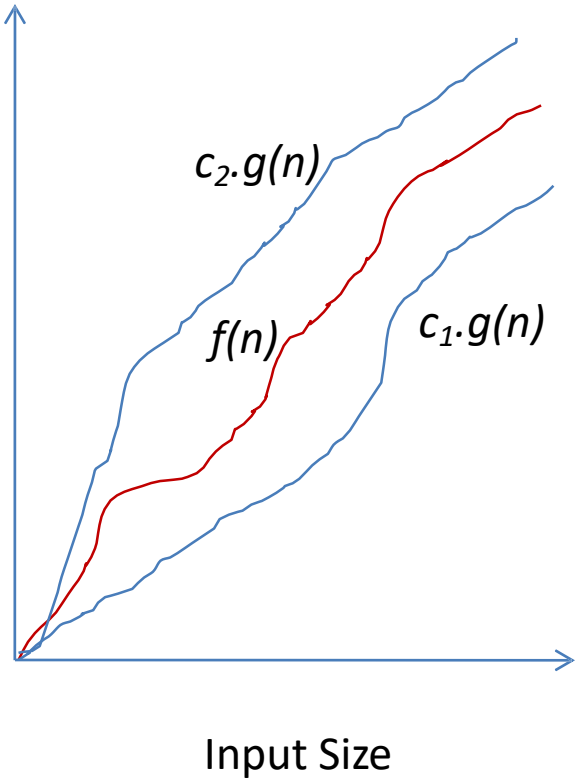
– Used to describe *best-case asymptotic running* times
  - E.g., lower-bound of *searching* an unsorted array; lower bound for *sorting* an array



Asymptotic Notation: Tight Bound

- The "big-Theta" θ-Notation
  - Asymptotically tight bound
  - f(n) = θ(g(n)) if there exists consts. c_1, c_2 and n_0 s.t. $\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$ for $n \geq n_0$
- *f(n) = θ (g(n))* **iff**

  f(n)=O(g(n)) and *f(n)=Ω(g(n))*



Small "o"

- ***o*** Notation
  - Asymptotic **strict upper** bound
  - *f(n)=O(g(n)),* if there exists constants *c* and n_0, s.t.
    - $f(n) < c.g(n)$ for $n \geq n_0$

*Similarly small omega, **ω**, is strict lower bound*

Asymptotic Notation

- Analogy with real numbers
  - $f(n) = O(g(n))$ → $f \leq g$
  - $f(n) = \Omega(g(n))$ → $f \geq g$
  - $f(n) = \theta(g(n))$ → $f = g$
  - $f(n) = o(g(n))$ → $f < g$
  - $f(n) = \omega(g(n))$ → $f > g$

Polynomial and Intractable Algorithms

- **Polynomial Time complexity**
  - An algorithm is said to be polynomial if it is ***O(n^d)***
    - for some integer ***d***
  - Polynomial algorithms are said to be efficient
    - They solve problems in reasonable times!
- ***Intractable Algorithms***
  - **Algorithms for which there is no *known***
    - **polynomial time algorithm.**

Tasks

- Submit CodeChef handle & profile link **2 Sep 2016** to mailing list

- Self study (Sahni Textbook)
  - **Check**: Have you read Chapters 5 & 6 "Linear Lists—Array & Linked Representations"
  - **Read**: Chapter 3 & 4 "Asymptotic Notation" & "Performance Measurement"
  - **Try:** Exercise 18 from Chapter 3 of textbook
- Finish Assignment 2 by Sun Sep 11 *(50 points)*
  - All submissions MUST work (compile, run) on turing cluster!

Questions?