Introduction to Algorithms
Lecture 9

By: Darmen Kariboz

Asymptotic Notations

- Overview
- “ Big O” notation
- “Omega” notation
- “Theta” notation
- General rules

Why should we care about Time Complexity?

- Prime numbers

1. i=2 to n-1

2. i=2 to $\sqrt{n}$

Why should we care about Time Complexity?

- Prime numbers

1. $i=2$ to $n-1$ ($n-2$ times)
2. $i=2$ to $\sqrt{n}$ ($\sqrt{n} - 1$ times)

Why should we care about Time Complexity?

- Prime numbers

• n=11	1. 9 ms	2. 2 ms
• n=101	1. 99 ms	2. 9 ms
• n=1000006	1. 1000 sec	2. 1 sec
• $n=10^{10}$	1. 115 days	2. 1.6 min
• n = infinity	1. n	2. $\sqrt{n}$

How to analyze Time Complexity?

- Running time depends on :

1. Processor (single vs multi)
2. Read/write speed to memory
3. 32 bt vs 64 bit
4. Input

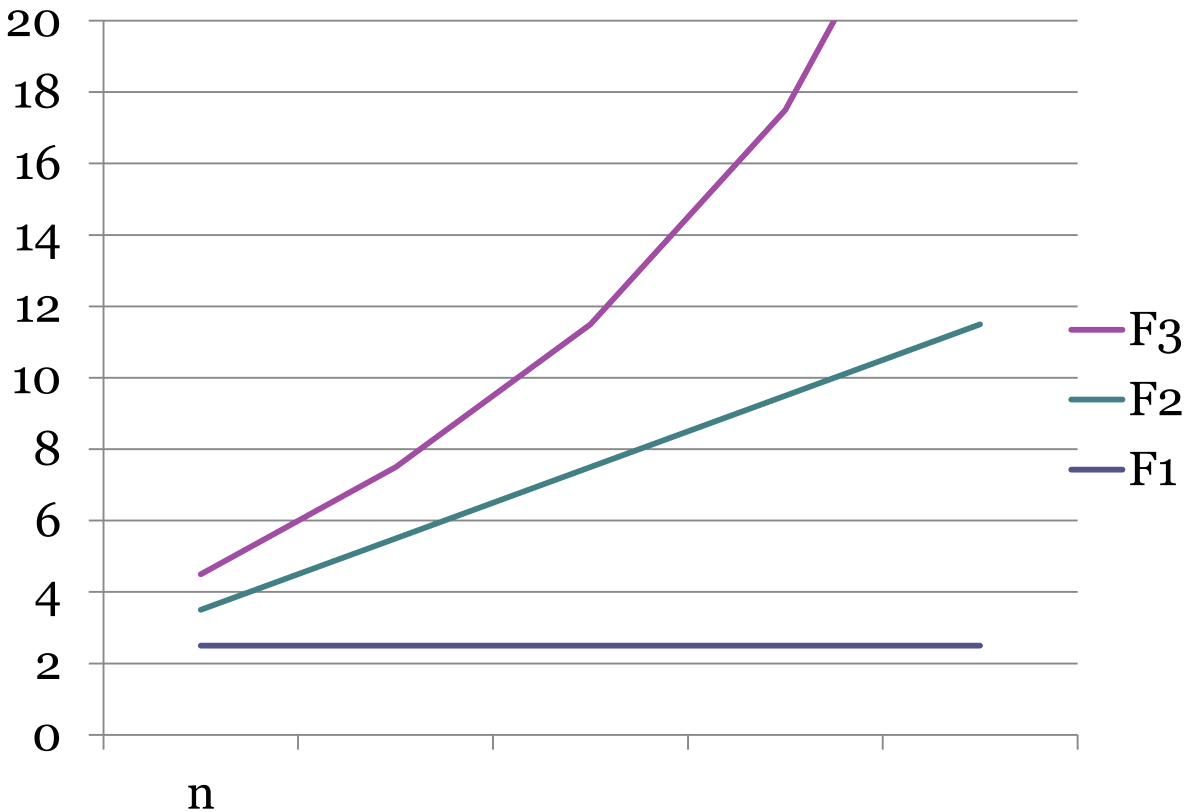
Model Machine

- Single processor
- 32 bit
- 1 unit time for arithmetical and logical operations
- 1 unit for assignment and return

- F1 - > Sum of A and B
- F2 -> Sum of list
- F3 -> Sum of matrix

- F1 - > Sum of A and B $T = k$
- F2 -> Sum of list $T = an + b$
- F3 -> Sum of matrix $T = an^2 + bn + c$

- F1 - > Sum of A and B $T = k$
- F2 -> Sum of list $T = an + b$
- F3 -> Sum of matrix $T = an^2 + bn + c$



Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or O (n)
  It is, at most, roughly ***proportional*** to n.

- $f_B(n)=n^2+1$ is *order* n^2, or O (n^2). It is, at most, roughly ***proportional*** to n^2.

- In general, any O(n^2) function is <u>faster- growing than</u> any O(n) function.

Example

- $n^4 + 100n^2 + 10n + 50$
- $10n^3 + 2n^2$
- $n^3 - n^2$
- constants

  10

  1273

Example

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- constants

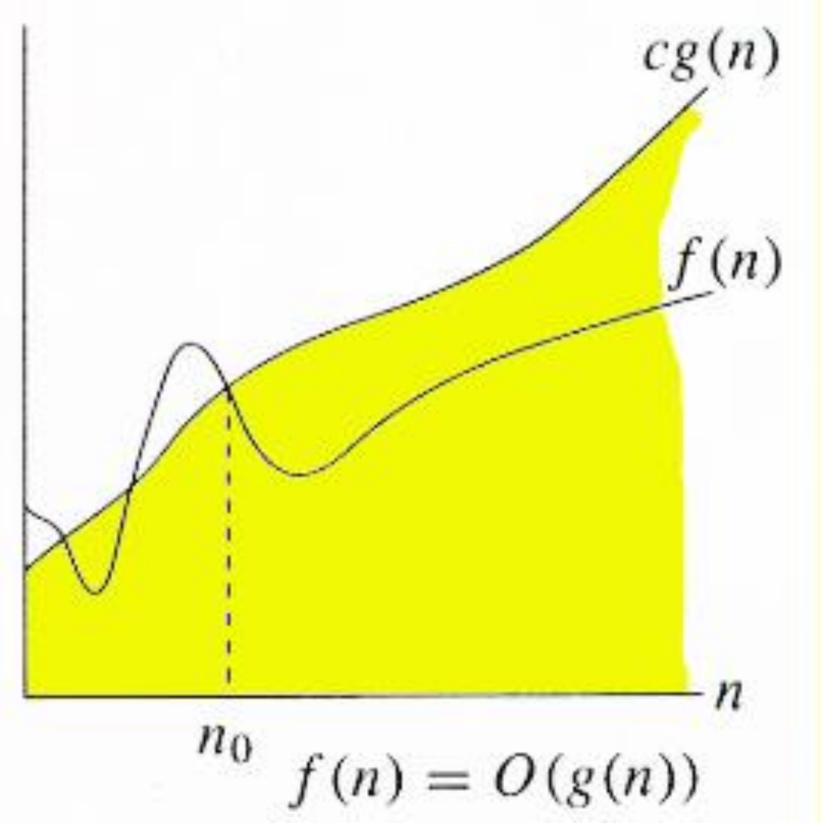
  10 is $O(1)$

  1273 is $O(1)$

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

Examle

- $f(n) = 5n^2 + 2n + 1$
- $g(n) = n^2$

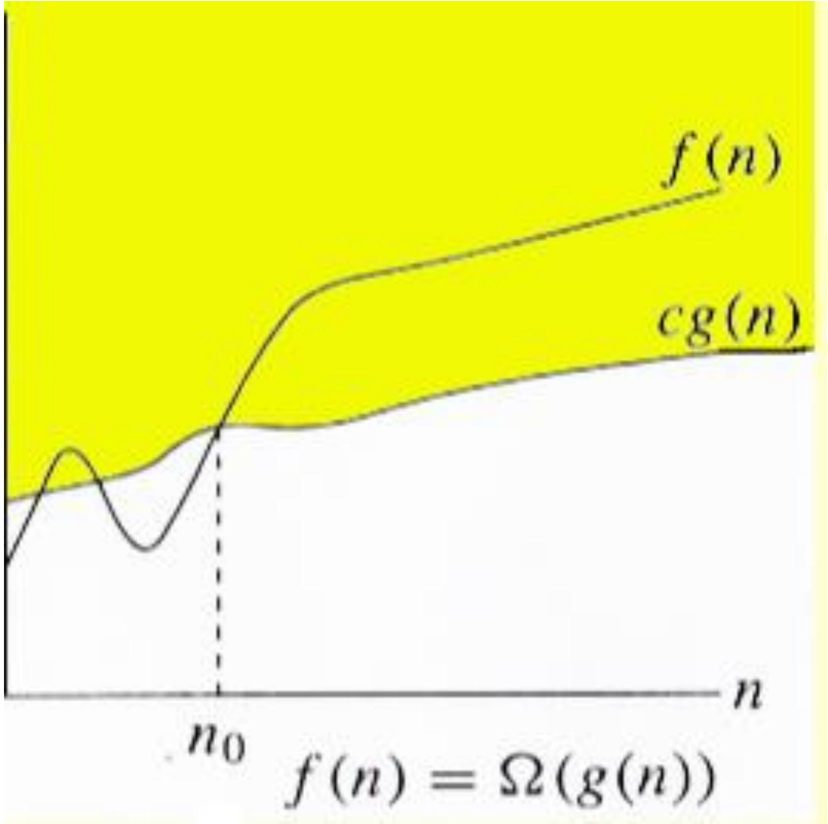
- $f(n) <= 8n^2$, $c = 8$ and $n_0 = 1$

- $f(n) = O(n^2)$

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n:

$\Omega(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Examle

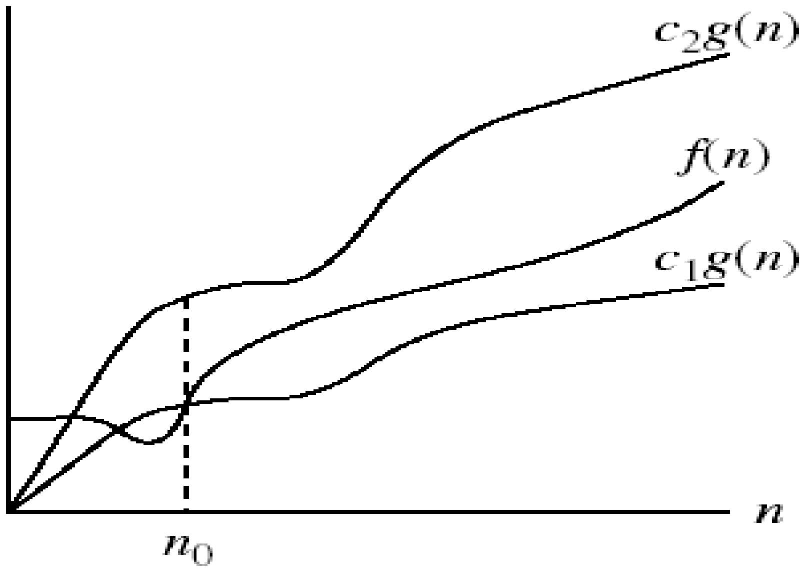
- $f(n) = 5n^2 + 2n + 1$
- $g(n) = n^2$

- $f(n) >= 5n^2$, $c = 5$ and $n_0 = 0$

- $f(n) = \Omega(n^2)$

Θ–notation

$$\Theta(g(n)) = \{ f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Examle

- $f(n) = 5n^2 + 2n + 1$
- $g(n) = n^2$

- $f(n) = n^2$, $c_1 = 5$, $c_2 = 8$ and $n_0 = 1$

- $f(n) = \Theta(n^2)$

General Rules

- We analyze time complexity for:
  1. Very large input size
  2. Worst case scenario

- Rules:
  1. Drop lower order terms
  2. Drop constants and multipliers

Examle

- $17n^4 + 3n^3 + 4n + 8$

- $16n + \log n$

O(1) Constant Time:

- An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

O(n) Linear Time

- An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases.

- Array: Linear Search, Traversing, Find minimum etc
- ArrayList: contains method
- Queue: contains method

O(log n) Logarithmic Time:

- An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size

- Binary Search

- the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess is too high or too low. Twenty questions game implies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as binary search

O(n log n) Logarithmic Time:

- Think of this as a combination of O(log(n)) and O(n). The nesting of the for loops help us obtain the O(n*log(n))

- QuickSort
- MergeSort
- HeapSort

O(n2) Quadratic Time

- An algorithm is said to run in quadratic time if its time execution is proportional to the square of the input size.

- Bubble Sort
- Selection Sort
- Insertion Sort

Quiz

- Next week
- Lectures 6-9 (including both)