Design and Analysis of Algorithms

L06: Performance Analysis

Dr. Ram P Rustagi
Sem IV (2019-H1)
Dept of CSE, KSIT
rprustagi@ksit.edu.in

Resources

- Text Book 1: Levitin
- Text Book: Horowitz

Amortized Cost

- Amortized cost:
  - An accounting artifact that often may not have any direct relationship to actual complexity of the operation.
- Requirement for amortized cost
  - *Sum of amortized complexities of all operations in any sequence of operations be greater than or equal to sum of the actual complexities, i.e*

$$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i) \qquad (1)$$

- Amortized cost can be viewed as the amount being charged for the operation rather than actual cost of the operation.

Examples: Amortized Cost

- Amortized cost of eating out (per day)
  - Consider that you eat dinner out every day and actuall cost of dinner typically is as follows.
    - Mon-Thu: Rs 50
    - Fri: Rs Rs 100
    - Sat-Sun: Rs 150
  - Consider that you need to have required money in the pocket before you proceed for dinner. The restaurant requires full payment in advance. You have fixed source of income per day (e.g. you can get only X amount per day from the source e.g. parents?).
  - What should be the value of X?
    - Rs 80?, Rs 100?
    - X is considered as amortized cost (potential func)

Amortized Cost : Potential Function

- To arrive at amortized cost relative to actual cost, define a Potential function `P(i)` for the `i`th operation

  `P(i)=amortized(i)-actual(i)+P(i-1)`.....(2)

- `i`th operation causes the potential function to change by the difference between the amortized and actual cost

$$\sum_{1 \le i \le n} P(i) = \sum_{1 \le i \le n} (amortized(i) - actual(i) + P(i-1))$$

$$\Rightarrow \sum_{1 \le i \le n} (P(i) - P(i-1)) = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

$$\Rightarrow P_n - P0 = \sum_{1 \le i \le n} (amortized(i) - actual(i))$$

$$\Rightarrow P_n - P_0 \ge 0 \qquad (3)$$

Amortized Cost : Potential Function

$$P(n) - P(0) \geq 0 \qquad (3)$$

- Potential function:
  - Assuming $P_0 \geq 0$, the potential `P(i)` represents the amount by which the first `i` operations have been overcharnged.
- Methods for computing amortized costs
  - Aggregate method
  - Accounting method
  - Potential method

Computing Amortized Costs

- Aggregate method:
  - Determine an upper bound for the sum of n operations.
  - Divide the amount by n to get amortized cost
  - Example: eating out dinner weekly limit of Rs `1000`
    - Amortized cost per day = `1000/7`= Rs.`142.85`...
- Accouting method:
  - Guess an amount show that P(i) satisfies eqn (2 & (3) i.e. `P(i)=amortized(i)-actual(i)+P(i-1)`, and `P(n) - P(0) ≥ 0`
- Potential method:
  - Guess a potential function that satisfied eqn (3), and
  - Compute amortized cost using eqn (2)

Computing Amortized Costs

- Aggregate method appears to be most comfortable
  - Hardest to use because
  - How to get upper bound on worst case behaviour
- Accouting method is intuitive to use
  - just need to verify eqn (2) and (3)
  - Often provides a tight bound on complexity of sequence of operations
- Potential method is again hardest to use
  - Guessing a proper potential function is difficult
  - Some applications this is the only way

Example: Subset Generation

- Problem: Given a set of `n` elements, generate all of its subsets i.e.
  - subset of set of `n` elements is defined by 2^n vectors `x[1:n]`, where each `x[i]` is either `0` or `1`.
  - `x[i]` is `1 iff` i^{th} element of the set is member of the subset.
  - Subsets of a set of 3 elements given by 8 vectors
    - `000,100,010,110,001,101,011,111`
- Enumeration strategy:
  - Start with subset `00...0`, and generate remaining subsets one at a time by invoking a fn *NextSubset*.
  - scans current subset from left to right, change every `1` to `0` till it encounters `0` which is changed to `1`, and return.
  - Stops when no `0` is encounted.

Subset Generation: Time Complexity

- Problem: Determine time complexity of generating first `m` subsets (excluding first subset `00...0`).
- Worst case method:
  - In any invocation of *NextSubset* function, max `n` bits can be changed. Thus, worst case cost is `n`. Since there are m subsets, the total aggregate cost is `m*n`.
- Aggregate method:
  - When function *NextSubset* is invoked `m` times, the vector `x[n]` changes `m` times, `x[n-1]` changes `m/2` times, `x[n-2]` changed $m/2^2$ times, `x[n-i]` changes $m/2^i$ times. Thus, the total cost is

$$\sum_{0 \le i \le \lfloor log_2 m \rfloor} \left\lfloor \frac{m}{2^i} \right\rfloor < 2m$$

Subset Generation: Accounting Method

- Accounting method:
  - First guess the amortized complexity and then verify it
  - Intuitvely, we think on the average 2 vectors will change
  - Let us guess the amortized cost as 2.
    - To show that `P(n)-P(0)≥0`
  - Approach: use the credit method and distributed the excess charge when needed.
  - Initially, each `x[i]` is zero and a credit of `0`.
  - On first invocation of *NextSubset* `1` cost is used for changing `x[n]`, `1` cost goes to credit on changing vector `x[n]`.
  - On 2nd invocation of *NextSubset*, the credit of `x[n]` is used to change `1` to `0`, `1` cost (from amortized cost of 2) is used to change `x[n-1]`, 1 credit goes to `x[n-1]`

Subset Generation: Accounting Method

- Accounting method…
  - On 3rd invocation of *NextSubset*, the credit of `x[n]` is changed to `1` from `0` using `1` cost (from amortized cost of 2), `1` credit is assigned to `x[n]`, `1` previous credit remains with `x[n-1]`
  - On 4`th` invocation, `1` credit of `x[n]` is used to change it to `0`, `1` credit of `x[n-1]` is used to change it `0`, I credit from amortized cost (of 2) is used to change `x[n-2]` and `1` credit is assigned to `x[n-2]`
  - Continuing this way, each `x[i]` that is `1`, has a credit of `1` unit with it. This credit is used to pay for changing it from `1` to `0`. I unit cost from amortized cost of `2` is used to change last `x[i]` and `1` credit is assigned to `x[i]`
  - The credit on each `x[i]` which is `0` it zero.

Subset Generation: Accounting Method

- Accounting method…
  - Thus, on any j^{th} invocation, `P(j)-P(0)` equals number of `x[i]`s that are `1`. This number is always non-negative.
  - Thus equation `P(m)-P(0)` ≥ 0 holds true for all `m`.
  - Thus, the complexity of Accounting method is `2m`, since amortized cost is taken as `2` units.

Subset Generation: Potential Method

- Approach for Potential method:
  - Guess (postulate) a potential function that satisfies equation $P(n) - P(0) \geq 0$, and then obtain amortized costs.
  - Define `P(i)` as equal to number of `x[i]`'s that are in i^{th} subset.
  - By definition, then P(0) = 0
  - There for $P(i) - P(0)$ is always ≥ 0
  - Computing amortized cost
  - Consider any subset `x[1:n]`. Let q be the number of `1`s at the left end of `x[]`.
    - On next invocation potential changes by $1-q$, since q vectors change from `1` to `0`, and one `0` is replaced by `1`.
    - Actual cost is $q+1$, since $q+1$ vectors are changed.

Subset Generation: Potential Method

- Thus, the amortized cost is given by (eqn (2))

```
amortized(i) = actual(i)+P(i)-P(i-1)
             = actual(i) + change in potential
             = q+1+1-q
             = 2
```

Asymptotic Notation

- Focus of analysis framework
  - Order of growth of time complexity function
- Notation
  - C(n): Count of basic operations of an algorithm
  - g(n): Some simple function for comparison purpose
    - A non-negative function
  - t(n): running time of algorithm indicated by C(n)
    - A non-negative function

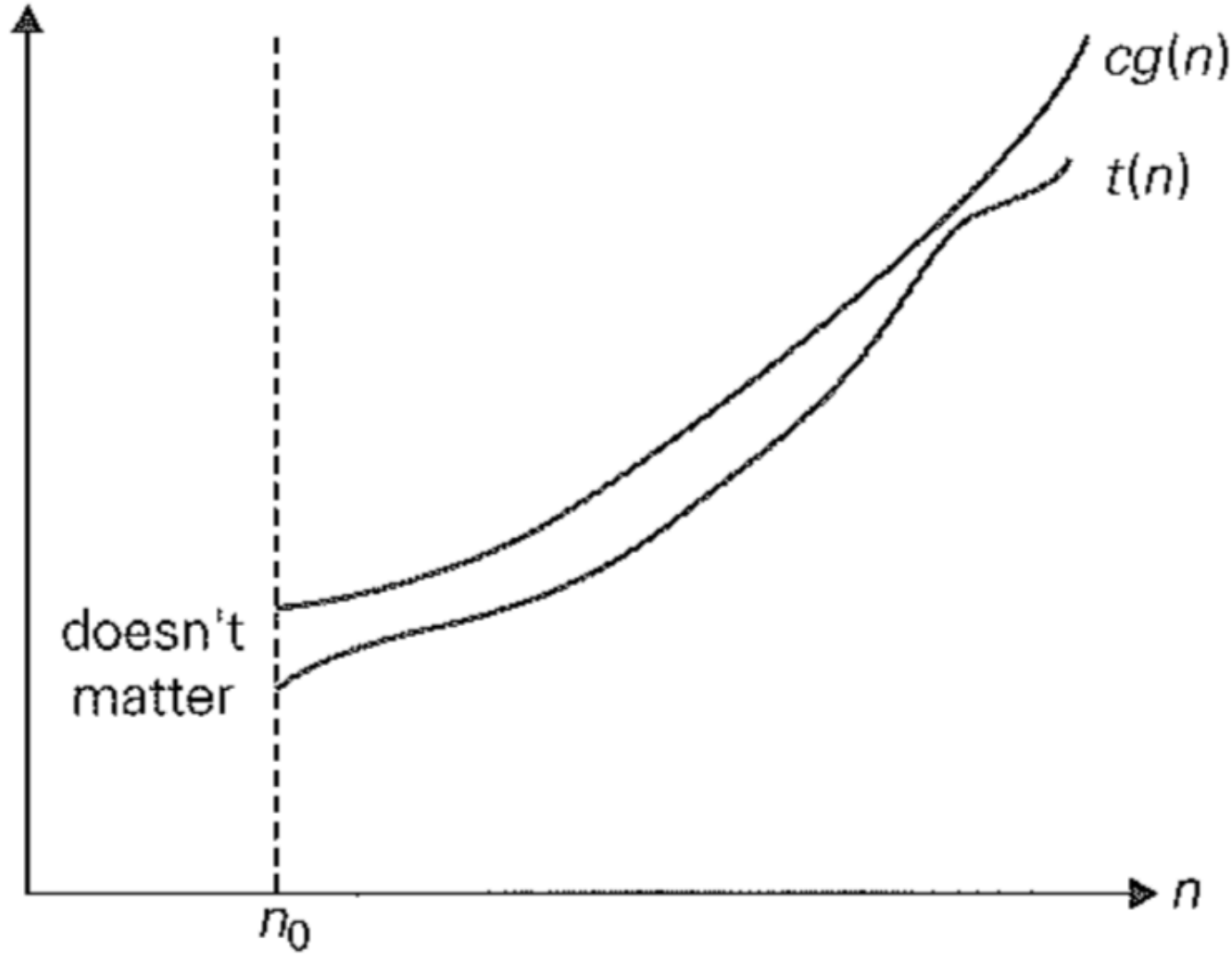
Asymptotic Notation

- Asymptotic order of growth
  - A way of comparing functions that ignores constant factors and small input sizes
  - $O(g(n))$: class of functions $f(n)$ that grow **no faster** than $g(n)$
  - $\Theta(g(n))$: class of functions $f(n)$ that grow **at same rate** as $g(n)$
  - $\Omega(g(n))$: class of functions $f(n)$ that grow **at least as fast** as $g(n)$

Asymptotic Notation: Big-Oh

- $O(g(n))$: set of all functions with a smaller or same order of growth as $g(n)$
  - $n \in O(n^2)$
  - $100n+5 \in O(n^2)$
  - $n(n+1)/2 \in O(n^2)$
  - $n^3 \notin O(n^2)$
  - $0.000001n^3 \notin O(n^2)$
  - $n^4+n^2+c \notin O(n^2)$
- A function $t(n)$ is said to be in $O(g(n))$ if $t(n)$ is bounded above by some +ve constant multiple of $g(n)$ for large n, i.e. $t(n) \in O(g(n))$, if
  $t(n) \leq cg(n)$ for all $n \geq n_0$

Big-Oh

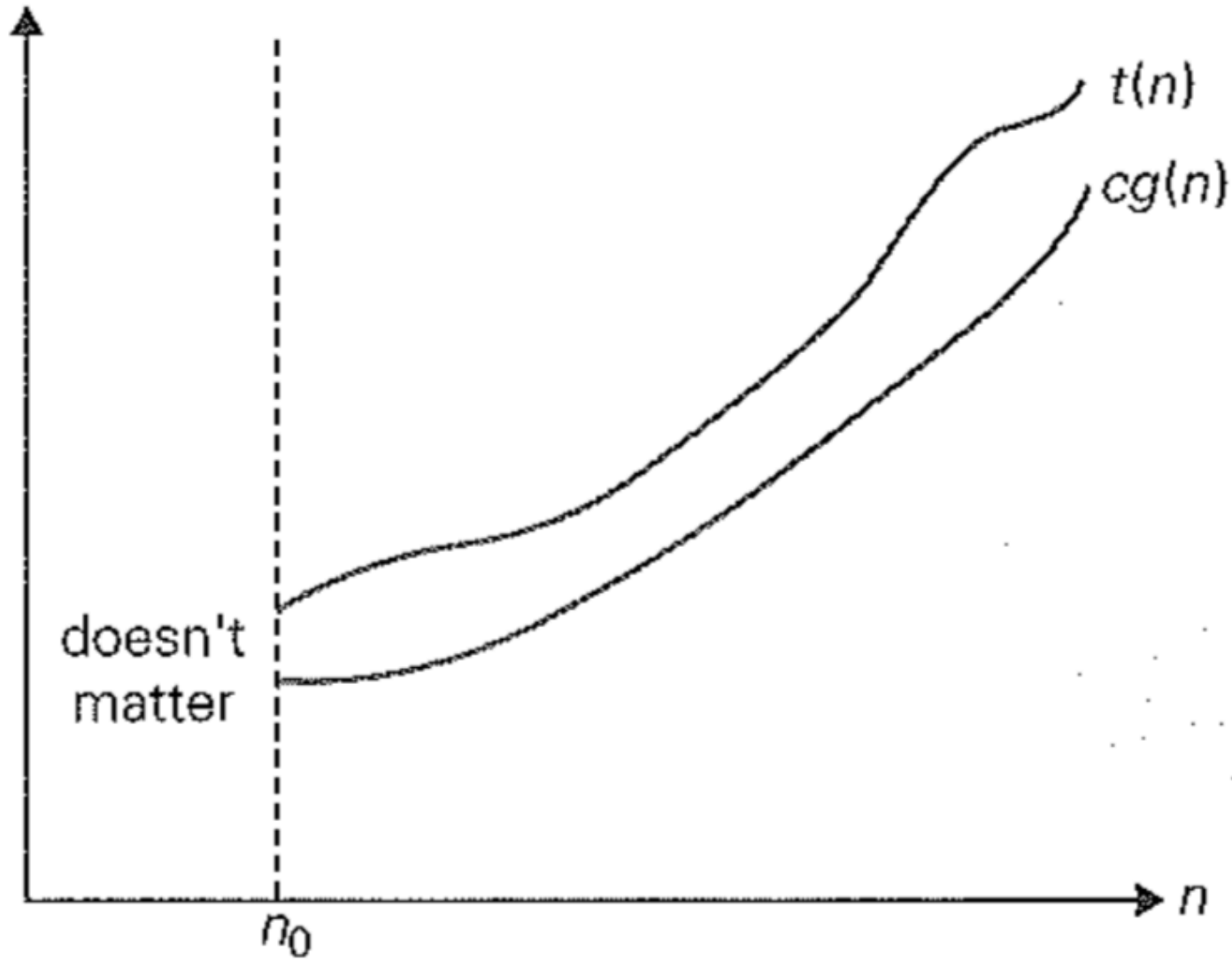


Big-Oh notation: `t(n)` $\in$ `O(`g`(`n`))`

Asymptotic Notation: Ω

- $\Omega(g(n))$: set of all functions with a larger or same order of growth as $g(n)$
  - $n \notin \Omega(n^2)$
  - $100n+5 \notin \Omega(n^2)$
  - $n(n+1)/2 \in \Omega(n^2)$
  - $n^3 \in \Omega(n^2)$
  - $0.000001n^3 \in \Omega(n^2)$
  - $n^4+n^2+c \in \Omega(n^2)$
- A function $t(n)$ is said to be in $\Omega(g(n))$ if $t(n)$ is bounded below by some +ve constant multiple of $g(n)$ for large n, i.e. $t(n) \in \Omega(g(n))$, if
  $t(n) \geq cg(n)$ for all $n \geq n_0$

Big Omega Notation



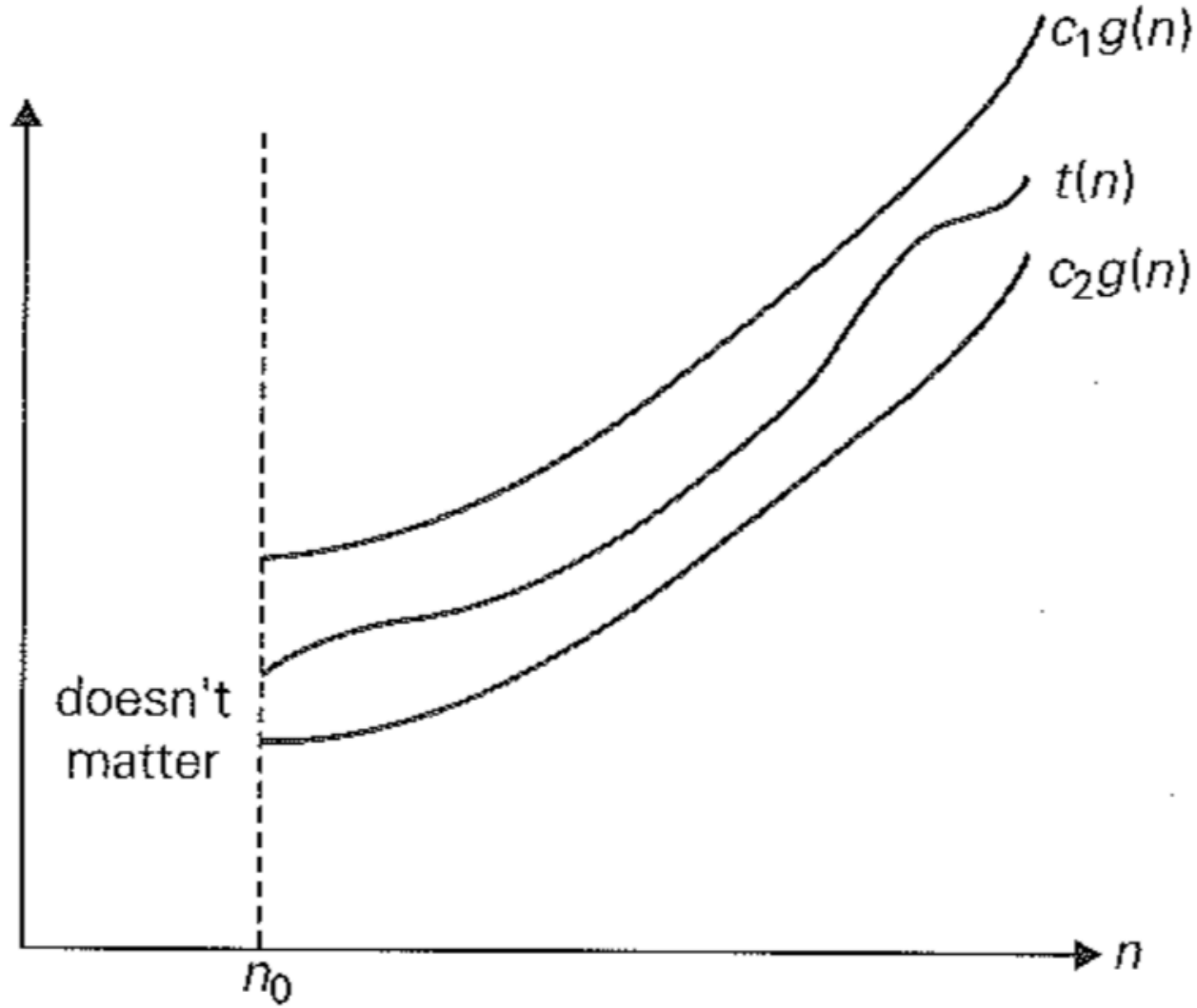
Big omega notation: $t(n) \in \Omega(g(n))$

Asymptotic Notation: Theta Θ

- $\Theta(g(n))$: set of all functions with a similar order of growth as $g(n)$ within a constant multiple
  - $n \notin \Theta(n^2)$
  - $100n+5 \notin \Theta(n^2)$
  - $n(n+1)/2 \in \Theta(n^2)$
  - $0.000001n^3 \notin \Theta(n^2)$
  - $n^4+n^2+c \notin \Theta(n^2)$
- A function $t(n)$ is said to be in $\Theta(g(n))$ if $t(n)$ is bounded both above and below by some +ve constant multiple of $g(n)$ for all large n,
  - i.e. $t(n) \in \Theta(g(n))$, if

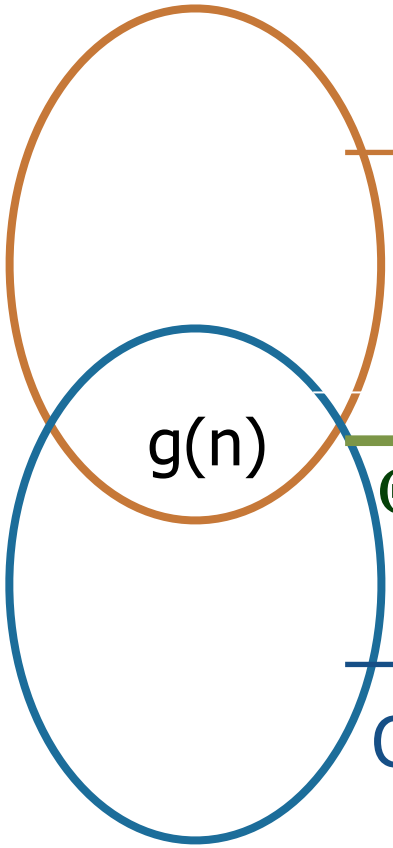
    $c_2 g(n) \leq t(n) \leq c_1 g(n)$ for all $n \geq n_0$

Big Theta Notation



Big theta notation: `t(n)` $\in \Theta(g(n))$

Asymptotic Notation View



>=

Ω(g(n)), functions that grow at least as fast as g(n)

=

Θ(g(n)), functions that grow at the same rate as g(n)

<=

O(g(n)), functions that grow no faster than g(n)

Theorem

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.
  - The analogous assertions are true for the Ω-notation and Θ-notation.
- Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, e.g.
  - $5n^2 + 3n\log n \in O(n^2)$
- Some properties
  - $f(n) \in O(f(n))$
  - $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
  - *If* $f(n) \in O(g(n))$ *and* $g(n) \in O(h(n))$, *then* $f(n) \in O(h(n))$
  - $\Sigma_{1 \le i \le n} \Theta(f(i)) = \Theta(\Sigma_{1 \le i \le n} f(i))$

Exercises

- Compare the order of growth of
  1. n^2 and $n(n-1)/2$
  2. $\log_2 n$ and $\sqrt{n}$
  3. $n!$ and 2^n

Basic Asymptotic Efficiency Classes

- `1` Constant
- `log n` Logarithmic
- `n` Linear
- `nlog n` n-log-n
- n^2 Quadratic
- n^3 Cubic
- 2^n Exponential
- `n!` Factorial

Useful Summation Formulas

- $\Sigma_{1 \le i \le n} 1 = 1+1+\ldots+1 = n, \Rightarrow n \in \Theta(n)$
- $\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$
- $\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$
- $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1}-1 \in \Theta(2^n)$
- $\Sigma_{0 \le i \le n} a^i = 1+a+\ldots+a^n = (a^{n+1}-1)/(a-1)$ **for any** $a \neq 1$
- $\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$
- $\Sigma c a_i = c \Sigma a_i$
- $\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Summary

- Order of growth
  - Big-Oh
  - Big-Omega
  - Big-Theta