


Design and Analysis of Computer Algorithms
Growth of Functions

Growth of Functions

- As algorithm runs, the ***running time*** grows in terms of the input size. The running time
  - varies for different inputs of the same size
  - is affected by the hardware and software environment
  - increases with the input size
  - can be studied by experiments
- Changing the hardware/ software environment
  - affects the running time of algorithm by a constant factor, but
  - does NOT alter the growth rate of the running time.
- We are interested in characterizing the ***growth rate*** of running time as ***a function of the input size***.
- The characterized function is the ***time complexity*** of the algorithm.

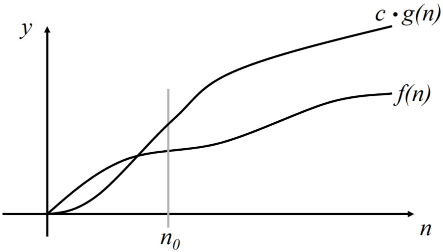
Asymptotic Notations - Big "oh"

- Suppose that f and g are two nonnegative functions.

Definition (*Big "oh"*)

$f(n) = O(g(n))$ iff. there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Examples
  - $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for $n \geq 2$
  - $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for $n \geq 5$



Big "oh"

- Some classes of functions
  - O(1): ____________constant
  - O(n): ____________linear
  - O(n^2): ____________quadratic
  - O(n^3): ____________cubic
  - O(2^n): ____________exponential
  - O($\log n$): ____________logarithmic
- $f(n) = O(g(n))$ states that $g(n)$ is an ***asymptotic upper bound*** of $f(n)$, so, $n = O(n^2)$.
- $g(n)$ should be as small as one can come up with.
- Check by yourself

Theorem

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

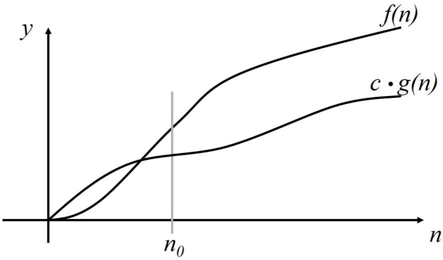
Asymptotic Notations - Big Omega

- Suppose that f and g are two nonnegative functions.

Definition (*Big-Omega*)

$f(n) = \Omega(g(n))$ iff. there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

- Examples
  - $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$
  - $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq 10n^2$ for $n \geq 1$



Big Omega

- Some classes of functions
  - $\Omega(1)$: __________constant
  - $\Omega(n)$: __________linear
  - $\Omega(n^2)$: __________quadratic
  - $\Omega(n^3)$: __________cubic
  - $\Omega(2^n)$: __________exponential
  - $\Omega(\log n)$: __________logarithmic
- $f(n) = \Omega(g(n))$ states that $g(n)$ is an ***asymptotic lower bound*** on $f(n)$, so, $n^2 = \Omega(n)$.
- $g(n)$ should be as large as one can come up with.
- Check by yourself

Theorem

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = \Omega(n^m)$.

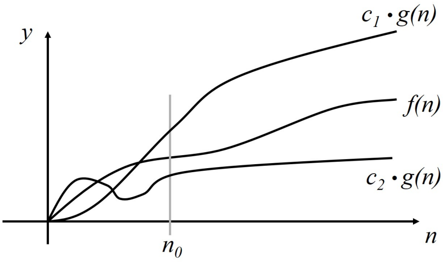
Asymptotic Notations - Big Theta

- Suppose that f and g are two nonnegative functions.

Definition (*Big-Theta*)

$f(n) = \Theta(g(n))$ iff. there exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

- Examples
  - $3n + 2 = \Theta(n)$ as $3n \leq 3n + 2 \leq 4n$ for $n \geq 2$
  - $10 \log n + 4 = \Theta(\log n)$



Big Theta

- Some classes of functions
  - $\Theta(1)$: __________constant
  - $\Theta(n)$: __________linear
  - $\Theta(n^2)$: __________quadratic
  - $\Theta(n^3)$: __________cubic
  - $\Theta(2^n)$: __________exponential
  - $\Theta(\log n)$: __________logarithmic
- $g(n)$ should have the same growth rate with $f(n)$ and vice versa.
- Check by yourself

Theorem

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = \Theta(n^m)$, where $a_m > 0$.

Asymptotic Notations - Little "oh"

- Suppose that f and g are two nonnegative functions.

Definition (*Little "oh"*)

$f(n) = o(g(n))$ iff.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Definition (alternative way)

$f(n) = o(g(n))$ iff. for any positive constant c, there exists a constant n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$.

- Examples
  - $3n + 2 = o(n^2)$ as $\lim_{n \to \infty} \frac{3n+2}{n^2} = 0$.
  - $3n + 2 = o(n \log n)$ as $\lim_{n \to \infty} \frac{3n+2}{n \log n} = 0$.

Asymptotic Notations - Little omega

- Suppose that f and g are two nonnegative functions.

Definition (***Little "omega"***)

$f(n) = \omega(g(n))$ iff.

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.$$

Definition (alternative way)

$f(n) = \omega(g(n))$ iff. for any positive constant c, there exists a constant $n_0 > 0$ such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$.

About Asymptotic Notations

- Note that the coefficients in all the $g(n)$'s should be 1.
  - Discussing the growth of functions
  - Notations representing the rate of growth
  - Comparing functions (algorithms)
- Note:
  - ***Worst case*** analysis using O() and related to the upper bound
  - ***Best case*** analysis using $\Omega()$ and related to the lower bound
- ***Asymptotic complexity*** can be determined quite easily without determining the exact step count.
- Need practice and experience to quickly determine the asymptotic complexity for an algorithm.

Analysis of Insertion Sort

INSERTION-SORT(A)	time	asym.
for $j = 2$ **to** $A.length$	n	$O(n)$
$key = A[j]$	$n-1$	$O(n)$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	$n-1$	$O(n)$
$i = j - 1$	$n-1$	$O(n)$
while $i > 0$ and $A[i] > key$	$\sum_{j=2}^{n} t_j$	$O(n^2)$
$A[i+1] = A[i]$	$\sum_{j=2}^{n}(t_j - 1)$	$O(n^2)$
$i = i - 1$	$\sum_{j=2}^{n}(t_j - 1)$	$O(n^2)$
$A[i+1] = key$	$n-1$	$O(n)$
overall		$O(n^2)$

- The best-case time complexity is $O(n)$.
- Time complexity for the worst-case is $O(n^2)$.

Asymptotic Notations - Properties

- **Transitivity**
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$
  - applied to all the notations
- **Reflexivity**
  - $f(n) = \Theta(f(n))$
  - $f(n) = O(f(n))$
  - $f(n) = \Omega(f(n))$
- **Symmetry**
  - $f(n) = \Theta(g(n))$ iff. $g(n) = \Theta(f(n))$
- **Transpose symmetry**
  - $f(n) = O(g(n))$ iff. $g(n) = \Omega(f(n))$
  - $f(n) = o(g(n))$ iff. $g(n) = \omega(f(n))$

Example - Permutation

```
PERMUTATION(A, k, n)
if k == n // Output permutation
    for i = 1 to n
        Print A[i]
else // A[k..n] has more than one permutation.
    // Generate these recursively.
    for i = k to n
        t = A[k]; A[k] = A[i]; A[i] = t;
        // All permutations of A[k+1..n]
        PERMUTATION(A, k+1, n);
        t = A[k]; A[k] = A[i]; A[i] = t;
```

Deriving Running Time for Permutation

- Let $T(k, n)$ denote the running time.
- Line 1-3 takes $\Theta(n)$.
- In Line 4-8, since there are $n - k + 1$ iterations, and each iteration take $T(k + 1, n) + 1$ time, the running time is

$$\begin{aligned} T(k, n) &= \Theta((n - k + 1) \times (T(k + 1, n) + 1)) \\ &= \Theta((n - k + 1) \times T(k + 1, n)) \end{aligned}$$

- The overall running time for n elements as $k = 1$ is

$$\begin{aligned} T(1, n) &= n \times T(2, n) \\ &= n \times ((n - 1) \times T(3, n)) \\ &= n \times (n - 1) \times \cdots \times 2 \times T(n, n) \\ &= n \times (n - 1) \times \cdots \times 2 \times n \\ &= n! \times n \end{aligned}$$

Practical Complexities (1)

- The *complexity function* can be used to compare two programs P and Q which perform the same task.
- Suppose P is $\Theta(n)$ and Q is $\Theta(n^2)$, P is faster than Q asymptotically.
- Be aware of the "sufficient large n" in the definitions of asymptotic functions.
- For P and Q above, when n is small, Q might be faster than P since
  - The input size n is small
  - P has large low order terms

Practical Complexities (2)

- It would be better to look the complexity function in "class fashion". *i.e.* linear order as one class, constant order as a class, etc.
- The utility of programs with exponential complexity is limited to small n.
- Programs having a complexity of high-degree polynomial are also of limited utility