08 Fundamentals of Algorithm Analysis

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

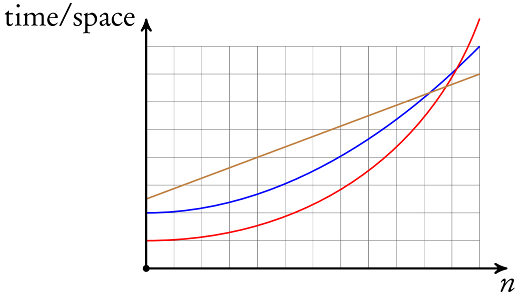
Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

February 15, 2019

Outline

Complexity

- The complexity of an algorithm indicates how costly to apply the algorithm, in terms of *time* and *space*.
- Most algorithms transform input objects into output objects. The cost of an algorithm typically grows with the input size.
- We characterize the complexities as functions of the input size n.



Theoretical Analysis

- The absolute running time depends on computing and processing power of hardware, not only the algorithm.
- We count the number of overall primitive operations for time measurement.
  - Evaluating an arithmetic or logic expression
  - Assigning a value to a variable
  - Indexing into an array-based list
  - Entering a function or method
  - Returning from a function or method
- We count the number of primitive data variables and reference variables for space measurement.
- We take account all possible inputs.

Counting Primitive Operations

By inspecting the code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

```
def list_max(a, n):           # operations
    m = a[0]                  2
    i = 1                     1
    while i < n:              n
        if a[i] > m:          2(n-1)
            m = a[i]          2(n-1)
        i = i+1               2(n-1)
    return m                  1
                        total 7n-2
```

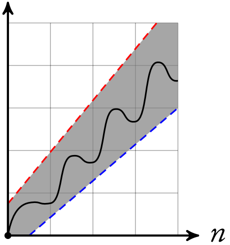
Estimating Running Time

- Algorithm *list_max* executes $7n-2$ primitive operations in the worst case. We define:

$$a = \text{time taken by the fastest primitive operation}$$
$$b = \text{time taken by the slowest primitive operation}$$

- Let $T(n)$ be worst-case time of *list_max*. Then

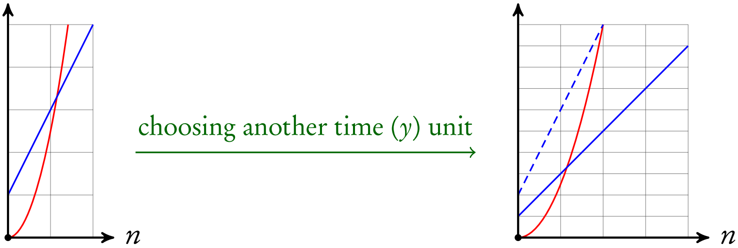
$$a(7n-2) \leqslant T(n) \leqslant b(7n-2).$$



- Hence, the running time $T(n)$ is bounded by two linear functions.
- Changing the hardware/software environment affects $T(n)$ by a constant factor, but does not alter the ***growth rate*** of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *list_max*, it is not affected by *constant factors* or *lower-order terms*.

The Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\mathcal{O}(g(n))$ if there are positive constants c and n_0 such that
$$f(n) \leqslant c \cdot g(n) \quad \text{for } n \geqslant n_0.$$
- Example: $2n+10$ is $\mathcal{O}(n)$. Because $2n+10 \leqslant c \cdot n \iff (c-2)n \geqslant 10 \iff n \geqslant \frac{10}{c-2}$. So, $2n+10 \leqslant 3n$ for $n \geqslant 10$.



Big-Oh Examples

- The function n^2 is not $\mathcal{O}(n)$.

$$n^2 \leqslant c \cdot n \iff n \leqslant c \quad \text{when } n > 0.$$

  The above inequality cannot be satisfied since c must be a constant.

- $7n - 2$ is $\mathcal{O}(n)$.

$$7n - 2 \leqslant 7n \quad \text{for } n \geqslant 1.$$

- $3n^3 + 20n^2 + 5$ is $\mathcal{O}(n^3)$.

$$3n^3 + 20n^2 + 5 \leqslant 4n^3 \quad \text{for } n \geqslant 21.$$

- $3\log n + 5$ is $\mathcal{O}(\log n)$.

$$3\log n + 5 \leqslant 8\log n \quad \text{for } n \geqslant 2.$$

Big-Oh Rules

- The big-Oh notation gives an *upper bound* on the growth rate of a function.
- The statement "$f(n)$ is $\mathcal{O}(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $\mathcal{O}(n^d)$, i.e., we drop lower-order terms and constant factors.
- We use the smallest possible class of functions, say "$2n$ is $\mathcal{O}(n)$" instead of "$2n$ is $\mathcal{O}(n^2)$".
- We use the simplest expression of the class, say "$3n+5$ is $\mathcal{O}(n)$" instead of "$3n+5$ is $\mathcal{O}(3n)$".

Seven Important Functions

Seven functions that often appear in algorithm analysis as growth rates.

Contant	1	
Logarithmic	$\log n$	
Linear	n	
Linearithmic (N-log-N)	$n \log n$	
Quadratic	n^2	
Cubic	n^3	(tractable)
Exponential	2^n	

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
- To perform the asymptotic analysis:
    - We find the worst-case number of primitive operations executed as a function of the input size.
    - We express this function with big-Oh notation.
- Example:
    - We determine that algorithm *list_max* executes at most $7n - 2$ primitive operations.
    - We say that algorithm *list_max* "runs in $\mathcal{O}(n)$ time".
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations and focus on repeated operations.

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The i-th prefix average of a list v is average of the first $i+1$ elements of v:

$$a[i] = \frac{v[0] + v[1] + \cdots + v[i]}{i+1}$$

- Computing the list a of prefix averages of another list v has applications to financial analysis

Prefix Averages — Quadratic

The following algorithm computes prefix averages in quadratic time by applying the definition.

```
def prefix_average_qua(v):                 # operations
    n = len(v)                             1
    a = [None]*n                           n
    for i in range(n):                     n
        s = v[0]                           n
        for j in range(1, i+1):            1 + 2 + ··· + (n − 1)
            s += v[j]                      1 + 2 + ··· + (n − 1)
        a[i] = s/(i+1)                     n
    return a                               1
```

The time complexity of *prefix_averages_qua* is $\mathcal{O}(1+2+\cdots+n)$, i.e., $\mathcal{O}(n^2)$.

Prefix Averages — Linear

The following algorithm computes prefix averages in linear time by keeping a running sum.

```
def prefix_average_lin(v):          # operations
    n = len(v)                      1
    a = [None]*n                    n
    s = 0                           1
    for i in range(n):              n
        s += v[i]                   n
        a[i] = s/(i+1)              n
    return a                        1
```

The time complexity of *prefix_averages_lin* is $\mathcal{O}(n)$.

Relatives of Big-Oh

- Big-Omega: $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geqslant 1$ such that
$$f(n) \geqslant c \cdot g(n),$$
for $n \geqslant n_0$.
- Big-Theta: $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geqslant 1$ such that
$$c' \cdot g(n) \leqslant f(n) \leqslant c'' \cdot g(n),$$
for $n \geqslant n_0$.
- Intuition for asymptotic notations:
  - $f(n)$ is $\mathcal{O}(g(n))$ if $f(n)$ is asymptotically **less than or equal to** $g(n)$.
  - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically **greater than or equal to** $g(n)$.
  - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically **equal to** $g(n)$.