02 Algorithm Analysis

College of Computer Science, CQU

Outline

- Introduction
- Best, Worst and Average Cases
- Asymptotic Analysis
- Space Bounds

Introduction

- How do you compare two algorithms for solving some problem in terms of efficiency?

- Solution 1:implement both algorithms as computer programs and then run them on a suitable range of inputs.

- Result: unsatisfactory

Asymptotic analysis

- Solution 2:using asymptotic analysis(渐进分析)
- Asymptotic analysis measures the efficiency of an algorithm as the input size becomes large.
- It is actually an estimating technique.
- However, asymptotic analysis has been proved useful.

Critical resource

- The critical resource for a program is most often its
  - running time.
  - space required to run the program.

- We have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer.

- The only alternative is to use some other measure as a surrogate for running time.

Estimating an algorithm's performance

- One primary consideration when estimating an algorithm's performance is the number of basic operations required by the algorithm to process an input of a certain size.

- Size is often the number of inputs processed.

- A basic operation must have the property that its time to complete does not depend on the particular values of its operands.

Example

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)   // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

- basic operations: to compare an integer's value to that of the largest value seen so far
- size: A.length

Growth rate

- The most important factor affecting running time is normally **size** of the input.
- For a given input size n we often express the time T to run the algorithm as a function of n, written as $T(n)$.
- Let us call *c* the amount of time required to compare two integers in function **largest.**
  - **$T(n) = cn$**
- This equation describes the growth rate for the running time of the largest-value sequential search algorithm

Growth rate

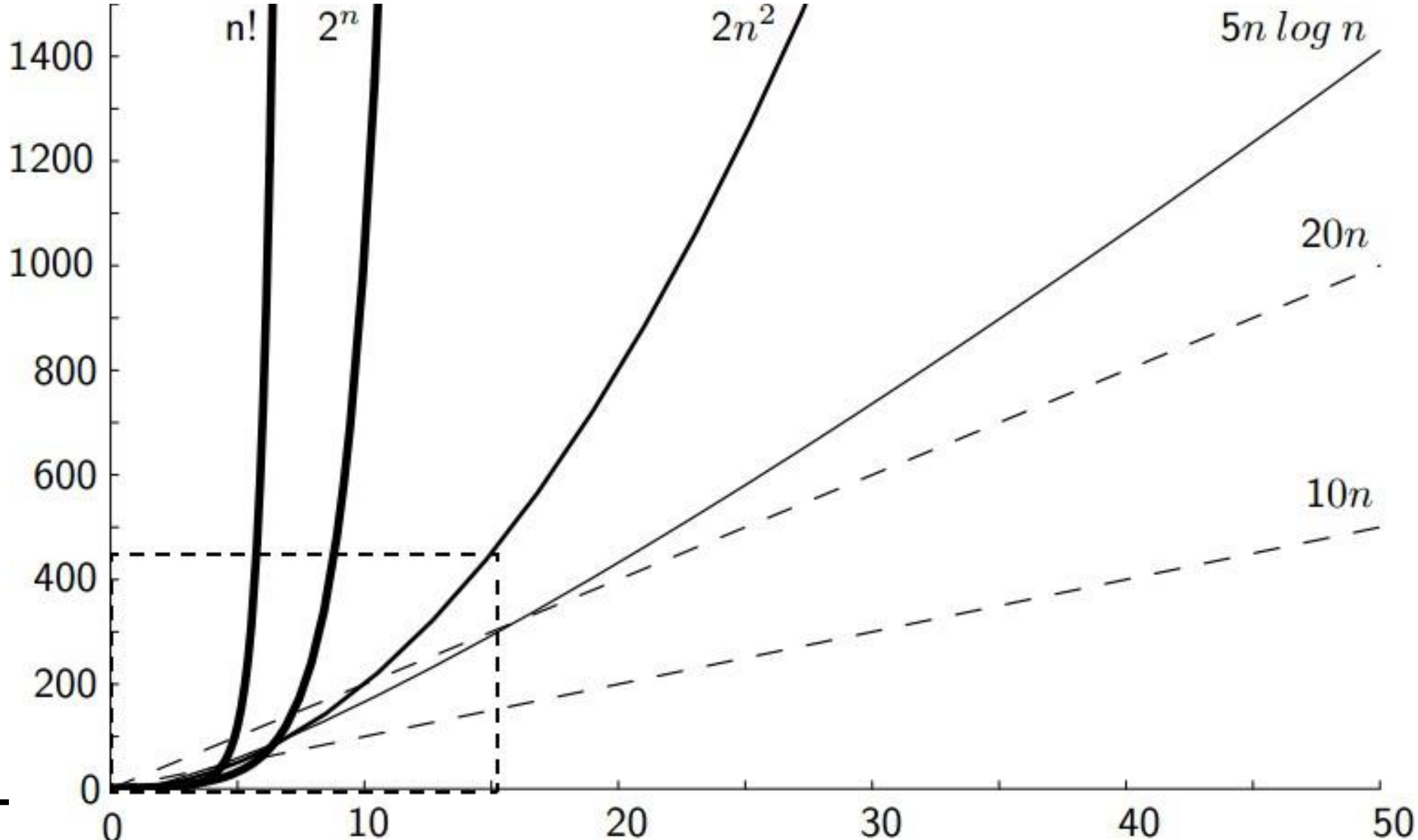
Example 3.3 Consider the following code:

```
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum++;
```

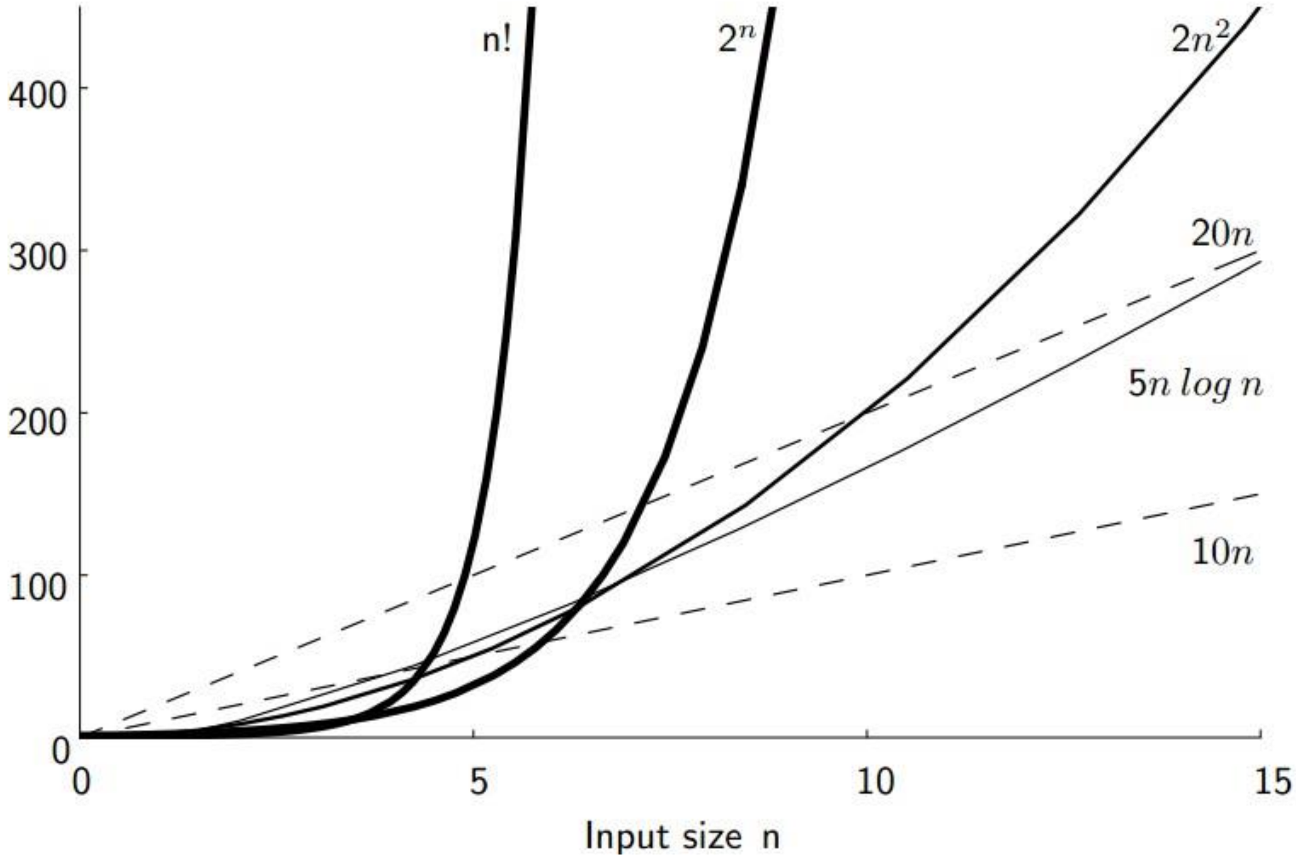
- The basic operation in this example is the increment operation for variable *sum*. We can assume that incrementing takes constant time; call this time c_2.
- The total number of increment operations is n^2.
- Thus, we say that the running time is $T(n) = c_2 n^2$

Growth rate

- The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.



Growth rate



Growth rate

n	log log n	log n	n	n log n	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Figure 3.2 Costs for growth rates representative of most computer algorithms.

Best, Worst, and Average Cases

- For some algorithms, different inputs of a given size require different amounts of time.
- Best case : Find “35”.Compare 1 value
- Worst case : Find “46”. Compare n values.
- Average case : Compare n/2 values.

35
16
78
85
43
29
33
21
54
46

Best, Worst, and Average Cases

- When analyzing an algorithm, should we study the best, worst, or average case?
- Normally the best case is too optimistic.
- For realtime applications we are likely to prefer a worst case analysis of an algorithm.
- Otherwise, we often desire an average-case analysis.

Faster Computer or Algorithm

- Assume that the new machine is ten times faster than the old.

f(n)	n	n′	Change	n′/n
$10n$	1000	10,000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

  - It would be much better off changing algorithms instead of buying a computer

Asymptotic Analysis

- *Asymptotic algorithm analysis(渐进算法分析).*
  - *Focus on* growth rate
  - *Ignore* the constants
  - Refer to the study of an algorithm as the input size "gets big" or reaches a limit

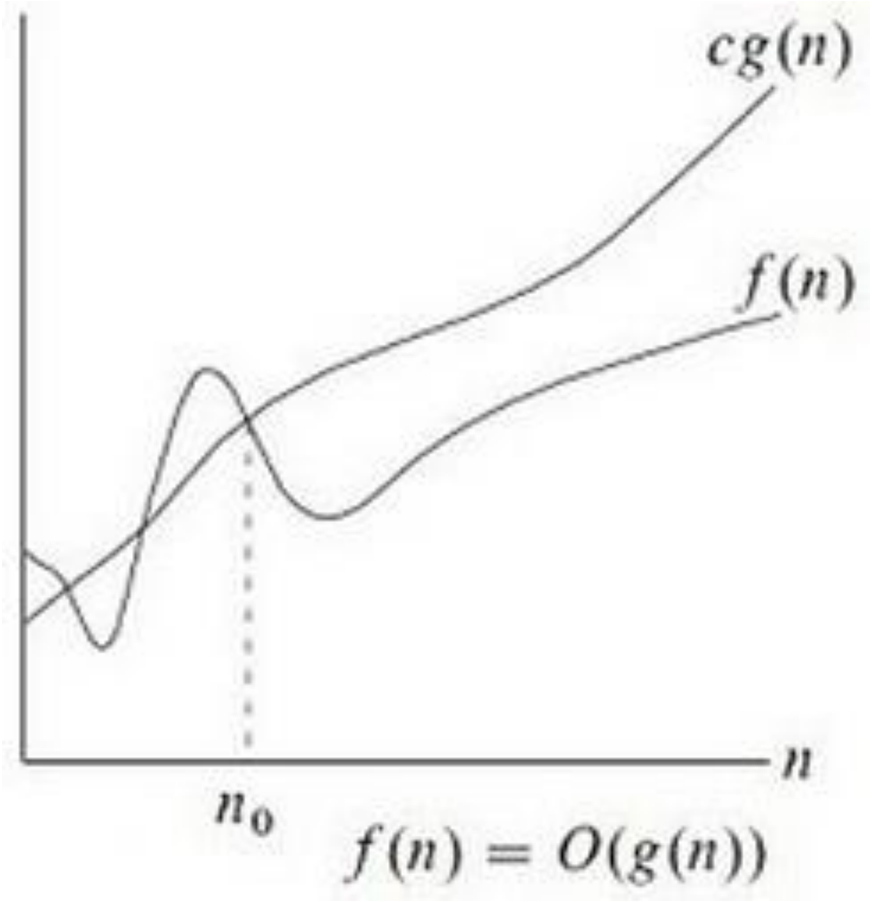
- Asymptotic analysis provides a simplified model of the running time or other resource needs of an algorithm.

Upper Bounds

- The *upper bound* for the growth of algorithm's running time indicates the upper or highest growth rate that the algorithm can have.
- The upper bound is defined by "Big-Oh".

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\mathrm{O}(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} \leq c$$


$cg(n)$
$f(n)$
n
n_0
$f(n) = O(g(n))$

Upper Bounds

- 意义:对于问题的所有输入,只要输入规模足够大(即n>n0),该算法总能在cf(n)步以内完成。
- 分别考虑最好、最坏、平均情况下的上限。
- 大O表示法给出了算法运行时间的上限，表明该算法可能有的最高增长率。-〉顶多坏到某种程度。

Example: 如果 $T(n) = 3n^2$, $\lim_{n\to\infty} \frac{3n^2}{n^2} = 3$，那么$T(n)$在$O(n^2)$中。

- 希望找找到最紧的上限。

当$T(n) = 3n^2$，我们可以说$T(n)$在$O(n^3)$中，但是更倾向于说$T(n)$在$O(n^2)$中。

Upper Bounds

Example 3.4 Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $\mathbf{T}(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $\mathbf{T}(n)$ is in $\mathrm{O}(n)$ for $n_0 = 1$ and $c = c_s$.

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2)n^2$ for all $n > 1$. So, $\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$ by the second definition.

Upper Bounds

Example 3.6 Assigning the value from the first position of an array to a variable takes constant time regardless of the size of the array. Thus, $\mathbf{T}(n) = c$ (for the best, worst, and average cases). We could say in this case that $\mathbf{T}(n)$ is in $O(c)$. However, it is traditional to say that an algorithm whose running time has a constant upper bound is in $O(1)$.

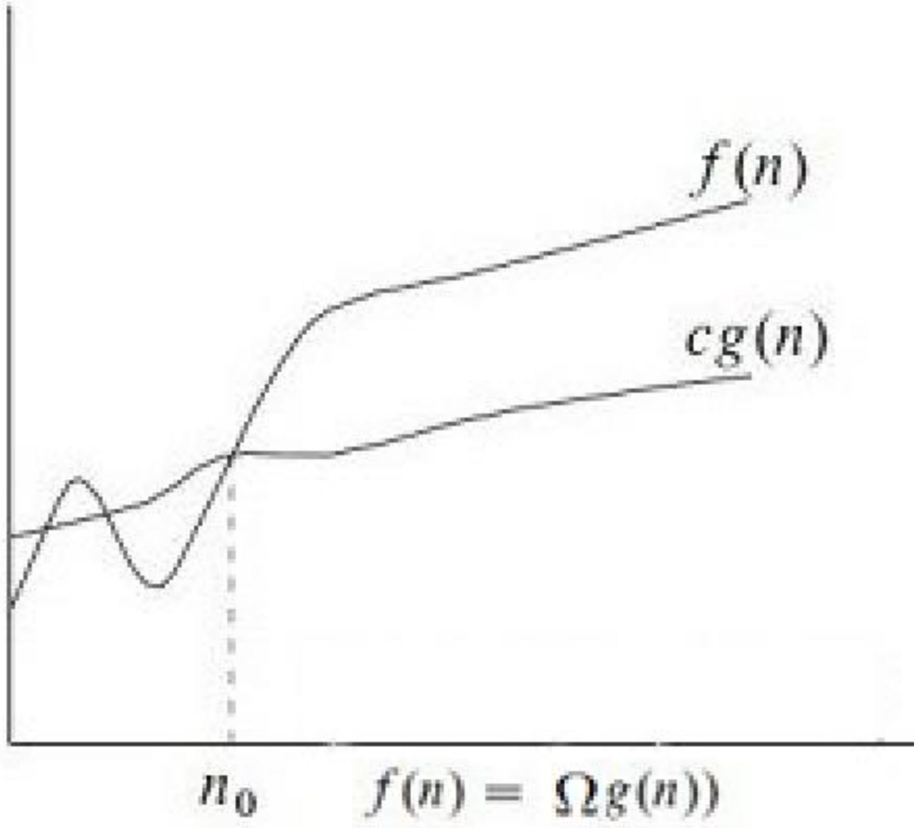
- 上限与最坏情况的区别：
  - 上限是用来确定运行时间的增长率，体现随着输入规模变化算法的代价变化
  - 最差情况是指：在一个给定的规模中，所有可能的输入中最糟糕的情况。

Lower Bounds

- The *lower bound* for the growth of algorithm's running time indicates the lower or lowest growth rate that the algorithm can have.
- The lower bound is defined by "Big-Omega".

For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.[1]

$$\lim_{n\to\infty}\frac{T(n)}{g(n)} \geq c$$



$f(n)$
$cg(n)$
n_0
$f(n) = \Omega g(n))$

Lower Bounds

- 意义:对于问题的所有输入,只要输入规模足够大(即n>n0),该算法至少需要cg(n)步以上才能 完成。
- 分别考虑最好、最坏、平均情况下的下限。
- 大Ω表示法给出了算法运行时间的下限，表明该算法可能有的最低增长率。

$$\lim_{n\to\infty}\frac{c_1 n^2 + c_2 n}{n^2} \geq \lim_{n\to\infty}\frac{c_1 n^2}{n^2} \geq c_1$$

$T(n) = c_1 n^2 + c_2 n.$

对于$n > 1$，$c_1 n^2 + c_2 n >= c_1 n^2$；

取$c = c_1$和$n_0 = 1$，有 $T(n) >= cn^2$；

因此，根据定义，$T(n)$在$\Omega(n^2)$中。

- 希望找找到最紧的下限。

Lower Bounds

Example 3.7 Assume $\mathbf{T}(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

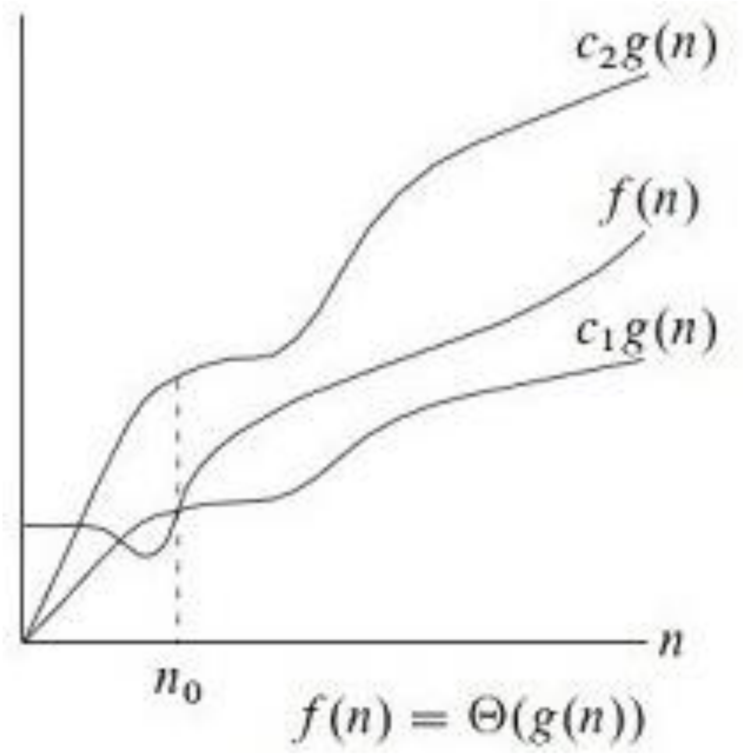
It is also true that the equation of Example 3.7 is in $\Omega(n)$. However, as with big-Oh notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n^2)$.

Θ Notation

- When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ(big-Theta) notation.
- An algorithm is said to be Θ(h(n)) if it is in O(h(n))and it is in Ω(h(n)).
- For an algorithm, the upper and lower bounds always meet.

Example: $T(n) = c_1 n^2$.

- Big-Oh:
  - $c_1 n^2 \leq c_1 n^2$ for all $n \geq 1$
  - Therefore, $T(n)$ is $O(n^2)$
- Big-Omega:
  - $c_1 n^2 \geq c_1 n^2$ for all $n \geq 1$
  - Therefore, $T(n)$ is $\Omega(n^2)$
- $T(n)$ is $O(n^2)$ and $\Omega(n^2)$, so $T(n)$ is $\Theta(n^2)$


$c_2g(n)$
$f(n)$
$c_1g(n)$
n
n_0
$f(n) = \Theta(g(n))$

Simplifying Rules

1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.

   → 上限的上限仍是上限

2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.

   → 常数因子可以忽略

3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n) + f_2(n)$ is in $\mathrm{O}(\max(g_1(n), g_2(n)))$.

   → 程序顺序给出的两部分，只考虑开销最大的那部分

4. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n)f_2(n)$ is in $\mathrm{O}(g_1(n)g_2(n))$.

   → 循环：总的代价为每次代价与循环次数的乘积

Calculating the Running Time for a Program

Example 3.9 We begin with an analysis of a simple assignment to an integer variable.

```
a = b;
```

Because the assignment statement takes constant time, it is $\Theta(1)$.

Example 3.10 Consider a simple **for** loop.

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

The first line is $\Theta(1)$. The **for** loop is repeated n times. The third line takes constant time so, by simplifying rule (4) of Section 3.4.4, the total cost for executing the two lines making up the **for** loop is $\Theta(n)$. By rule (3), the cost of the entire code fragment is also $\Theta(n)$.

Calculating the Running Time for a Program

Example 3.11 We now analyze a code fragment with several **for** loops, some of which are nested.

```
sum = 0;
for (i=1; i<=n; i++)       // First for loop
    for (j=1; j<=i; j++)   //   is a double loop
        sum++;
for (k=0; k<n; k++)        // Second for loop
    A[k] = k;
```

$\Theta(n^2)$.

Example 3.12 Compare the asymptotic analysis for the following two code fragments:

```
sum1 = 0;
for (i=1; i<=n; i++)       // First double loop
    for (j=1; j<=n; j++)   //   do n times
        sum1++;
```

$\Theta(n^2)$.

```
sum2 = 0;
for (i=1; i<=n; i++)       // Second double loop
    for (j=1; j<=i; j++)   //   do i times
        sum2++;
```

Calculating the Running Time for a Program

Example 3.13 Not all doubly nested **for** loops are $\Theta(n^2)$. The following pair of nested loops illustrates this fact.

```
sum1 = 0;
for (k=1; k<=n; k*=2)      // Do log n times
   for (j=1; j<=n; j++)    // Do n times
      sum1++;
```

$\Theta(n \log n)$

```
sum2 = 0;
for (k=1; k<=n; k*=2)      // Do log n times
   for (j=1; j<=k; j++)    // Do k times
      sum2++;
```

$\Theta(n)$

Typical Growth Rate

- There is a terminology for certain growth rate functions.

Function	Name
c	Constant
$\log n$	Logarithmic
$\log^2 n$	Log-squared
n	Linear
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Space Bounds

- Besides time, space is the other computing resource that is commonly of concern to programmers.
- The analysis techniques used to measure space requirements are similar to those used to measure time requirements.
- However, while time requirements are normally measured for an algorithm that manipulates a particular data structure, space requirements are normally determined for the data structure itself.
- The concepts of asymptotic analysis for growth rates on input size apply completely to measuring space requirements.

Space Bounds

Example 3.16 What are the space requirements for an array of n integers? If each integer requires c bytes, then the array requires cn bytes, which is $\Theta(n)$.

Example 3.17 Imagine that we want to keep track of friendships between n people. We can do this with an array of size $n \times n$. Each row of the array represents the friends of an individual, with the columns indicating who has that individual as a friend. For example, if person j is a friend of person i, then we place a mark in column j of row i in the array. Likewise, we should also place a mark in column i of row j if we assume that friendship works both ways. For n people, the total size of the array is $\Theta(n^2)$.

Space Bounds

- One important aspect of algorithm design is referred to as the *space/time tradeoff principle(空间时间权衡原理)*, which says that one can often achieve a reduction in time if one is willing to sacrifice space or vice versa.
- Many programs can be modified to reduce storage requirements by "packing" or encoding information. The resulting program uses less space but runs slower.
- Conversely, many programs can be modified to pre-store results or reorganize information to allow faster running time at the expense of greater storage requirements.

Homework

- P87, 3.8
- P88,3.12

Knowledge Points

- Chapter 3, pp.55-86