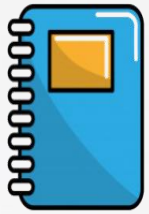
江西理工大学 Jiangxi University of Science and Technology
信息工程学院 School of information engineering


高级算法分析与设计

Advanced Algorithm Analysis and Design

Lecture 03:
Time complexity(PARTA)

Dr Ata Jahangir Moshayedi

EMAIL: ajm@jxust.edu.cn

Prof Associate ,
School of information engineering Jiangxi university of science and technology, China

LIVE Lecture series

Autumn _2022

江西理工大学 JIANGXI UNIVERSITY OF SCIENCE AND TECHNOLOGY

高级算法分析与设计

Advanced Algorithm Analysis and Design

LECTURE 03:

Time complexity_A

时间复杂度_A

Agenda

目录

CONTENTS

01

Why performance analysis?

为什么进行性能分析?

Why performance analysis?

为什么性能分析?

- performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!
- *performance == scale.*
- Imagine a text editor that can load 1000 pages but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it.
- If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

- 性能就像货币，通过它我们可以买到以上所有的东西。研究性能的另一个原因是——速度是有趣的!
- *性能= =标尺。*
- 想象一下，一个文本编辑器可以加载1000页，但每分钟可以检查一页的拼写，或者一个图像编辑器需要1小时将图像向左旋转90度，或者……
- 如果一个软件功能不能应付用户需要执行的大量任务，那么它就等于死了。

Given two algorithms for a task, how do we find out which one is better?

- One naive way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.

  一种简单的方法——实现这两种算法，并在计算机上为不同的输入运行这两个程序，看看哪个花的时间更少。

- There are many problems with this approach for the analysis of algorithms.
  - *It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.*
  - *It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.*

- 这种方法在分析算法时存在许多问题。
  - *有可能对于某些输入，第一种算法比第二种算法执行得更好。对于一些输入第二种执行得更好。*
  - *也有可能，对于某些输入，第一种算法在一台机器上执行得更好，而第二种算法在另一台机器上执行得更好。*

02

Performance Analysis

性能分析

Performance Analysis

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU.

Formally they are notified as complexities in terms of:

Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components:

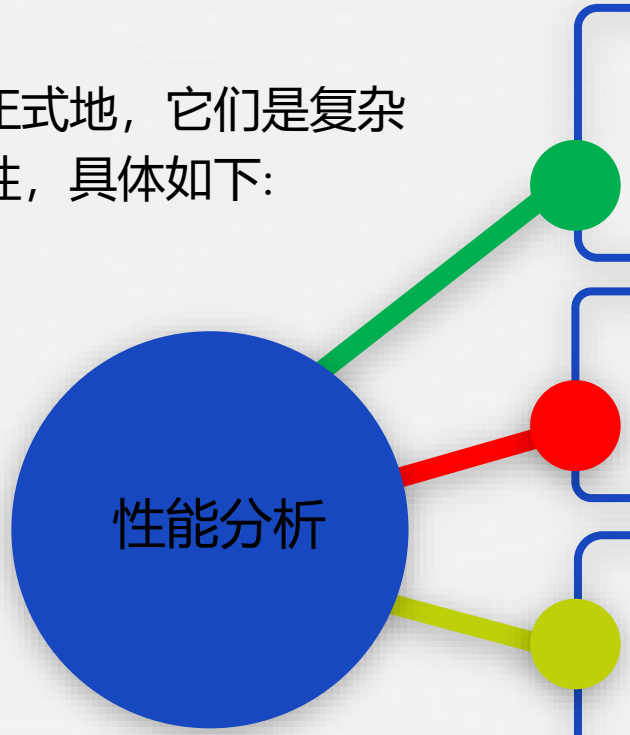
Fixed Component: This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables, and constants variables.

Variable Component: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

性能分析

算法的性能分析取决于两个因素，即内存使用量和CPU上消耗所用的计算时间。

正式地，它们是复杂性，具体如下:



一个算法的空间复杂度是指它从开始执行到结束所需要的内存量。任何算法所需的空间都是以下组成部分的和:

固定组件:这与输入和输出的特性无关。该部分包括:内存空间、简单变量空间、固定大小的组件变量和常量变量。

变量组件:这包括组件变量所需的空间，其大小取决于被解决的特定问题实例(输入/输出)，引用变量所需的空间和递归堆栈空间是最突出的组件之一。这也包括数据结构组件，如链表，堆，树，图等。

Performance Analysis

Therefore, the total space requirement of any algorithm 'A' can be provided as

- **Space(A) = Fixed Components(A) + Variable Components(A)**

Among both fixed and variable component, the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'. To identify the space complexity of any algorithm following steps can be followed:

1. Determine the variables which are instantiated by some default values.
2. Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
3. Generally, the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.
4. Sometimes more complex measures of the interrelationships among the data items can used.

性能分析

因此，任意算法A的总空间需求可表示为

- **空间(A) =固定分量(A) +可变分量(A)**

在固定分量和可变分量中，准确确定可变分量很重要，这样才能确定算法A的实际空间需求。要确定任何算法的空间复杂度，可以遵循以下步骤:

1. 确定由一些默认值实例化的变量。
2. 确定应该使用哪些实例特征来度量空间需求，这将是具体问题。
3. 一般来说，选择仅限于与算法的输入和输出的数量和量级相关的数量。
4. 有时可以使用数据项之间相互关系的更复杂的度量。

Example: Space Complexity

```
Algorithm Sum(number,size)
\\ procedure will produce sum of all numbers provided in 'number' list
{
      result=0.0;
      for count = 1 to size do
 \\will repeat from 1,2,3,4,....size times
            result= result + number[count];
      return result;
}
```

- In this example, when calculating the space complexity we will be looking for both fixed and variable components.
- ***here we have***
- ***Fixed components as 'result','count' and 'size' variable there for total space required is three(3) words.***
- Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n').
- because this will decide the size of 'number' list and will also drive the for loop.
- therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').
- therefore the space complexity can be written as **Space(Sum) = 3 + n;**

例如:空间复杂性

```
Algorithm Sum(number,size)
\\ procedure will produce sum of all numbers provided in 'number' list
{
      result=0.0;
      for count = 1 to size do
 \\will repeat from 1,2,3,4,....size times
            result= result + number[count];
      return result;
}
```

- 在本例中，当计算空间复杂度时，我们将寻找固定和可变分量。
- ***这里我们有***
- ***固定组件为'result'，'count'和'size'变量，总空间需要三(3)个字。***
- 变量组件被描述为存储在'size'变量中的值(假设存储在变量'size'中的值为'n ')。
- 因为这将决定'number'列表的大小，也将驱动for循环。
- 因此，如果size使用的空间是一个单词，那么'number'变量所需的总空间将是'n'(存储在变量'size'中的值)。
- 因此，空间复杂度可写成space (Sum) = 3 + n;

Time Complexity

时间复杂度

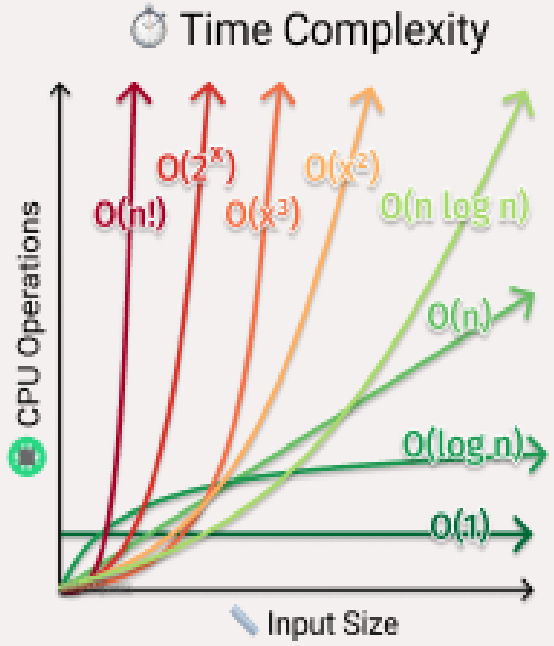
- **Time Complexity** of an algorithm(basically when converted to program) is the amount of computer time it needs to run to completion.
- The time taken by a program is the sum of the compile time and the run/execution time .
- The compile time is independent of the instance(problem specific) characteristics.
- Finding out the time complexity of your code can help you develop better programs that run faster. Some functions are easy to analyze, but when you have loops, and recursion might get a little trickier when you have recursion.

- 算法的时间复杂度(基本上是在转换为程序时)是指算法运行到完成所需的计算时间。
- 程序所花费的时间是编译时间和运行/执行时间的总和。
- 编译时间独立于实例(特定于问题的)特征。
- 找出代码的时间复杂度可以帮助您开发运行速度更快的更好程序。有些函数很容易分析，但当你有循环时，当你有递归时，递归可能会变得有点棘手。

Time Complexity

following factors effect the time complexity:

1. Characteristics of compiler used to compile the program.
2. Computer Machine on which the program is executed and physically clocked.
3. Multiuser execution system.
4. Number of program steps.



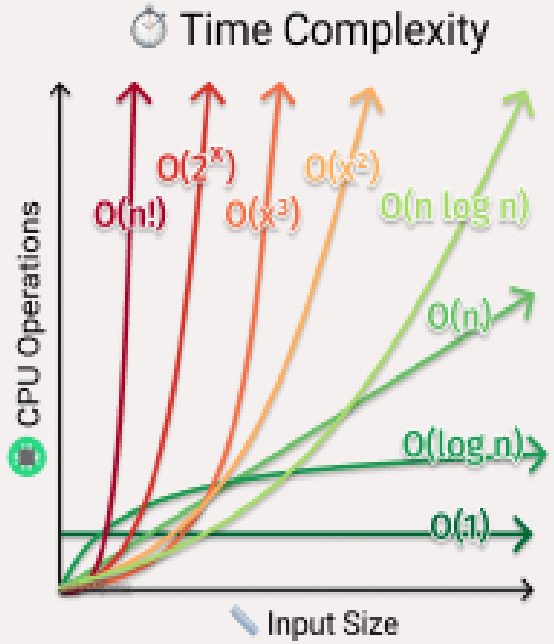
影响时间复杂度的因素有:

1. 用于编译程序的编译器的特征。
2. 计算机在其上执行程序并进行物理计时的机器。
3. 多用户执行系统。
4. 程序步骤数。

Time Complexity 时间复杂度

- Therefore, the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:

- **Time(A) = Fixed Time(A) + Instance Time(A)**



- 因此，时间复杂度再次由固定(仅因子1)和变量/实例(因子2,3和4)两部分组成，因此对于任何算法'A'，它被提供为:

- **时间(A) =固定时间(A) +实例时间(A)**

Time Complexity 时间复杂度

- Here the number of steps is the most prominent instance characteristics and the number of steps any program statement is assigned depends on the kind of statement like
  - comments count as zero steps,
  - an assignment statement which does not involve any calls to other algorithm is counted as one step,
  - for iterative statements we consider the steps count only for the control part of the statement etc.

- 在这里，步骤数是最突出的实例特征，而分配给任何程序语句的步骤数取决于语句的类型
  - 讨论算作开始，
  - 一个不涉及任何调用其他算法的赋值语句被视为一个步骤，
  - 对于迭代语句，我们认为步骤只计算语句的控制部分等等。

- Therefore, to calculate total number program of program steps we use following procedure.
- For this we build a table in which we list the total number of steps contributed by each statement.
- This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed.

- 因此，我们用下面的程序来计算程序的总步骤数。
- 为此，我们构建了一个表，其中列出了每个语句的总步骤数。
- 这通常是通过首先确定每次执行语句的步骤数和每个语句执行的频率来实现的。

Example: Time Complexity

- This procedure is explained using an example.

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	size+1	size + 1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

- In this example if you analyze carefully frequency of "for count = 1 to size do" it is 'size +1' this is because the statement will be executed one time more die to condition check for false situation of condition provided in for statement.
- Now once the total steps are calculated they will resemble the instance characteristics in time complexity of algorithm.
- Also, the repeated compile time of an algorithm will also be constant every time we compile the same set of instructions so we can consider this time as constant 'C'.
- Therefore, the time complexity can be expressed as: **Time(Sum) = C + (2size +3)**

例如:时间复杂度

- 通过一个例子来解释这个过程。

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	size+1	size + 1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

- 在这个例子中，如果你仔细分析“for count = 1 to size do”的频率，它是“size +1”，这是因为该语句将被执行一次，以检查for语句中提供的条件的错误情况。
- 现在计算出的总步长在时间复杂度上与算法实例特征相似。
- 此外，每当我们编译同一组指令时，算法的重复编译时间也将是常数，因此我们可以将此时间视为常数C。
- 因此，时间复杂度可以表示为:time (Sum) = C + (2size +3)

03

What do you mean by a good Algorithm?

Types of Algorithm Analysis

什么是好的算法?

算法分析类型

What do you mean by a good Algorithm?
什么是好的算法?

There are three asymptotic notations that are used to represent the time complexity of an algorithm.

有三种渐近符号用来表示算法的时间复杂度。

- Θ Notation (theta)
  Θ 符号(θ)
- Big O Notation
  大O符号
- Ω Notation
  Ω符号

- Before learning about these three asymptotic notation, we should learn about the best, average, and the worst case of an algorithm.

- 在学习这三种渐近表示法之前，我们应该学习算法的最佳、平均和最坏情况。

Types of Algorithm Analysis:

- Best case
- Average case
- Worst case

- in the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1)

- 最好的情况
- 平均情况
- 最差情况

- 在最佳情况分析中，我们计算算法运行时间的下界。我们必须知道导致执行的操作数量最少的情况。在线性搜索问题中，最好的情况发生在x出现在第一个位置时。最佳情况下的操作次数是常数(不依赖于n)。因此，最佳情况下的时间复杂度为Ω(1)。

Types of Algorithm Analysis:

算法分析类型:

- Best case
- Average case
- Worst case

- 最好的情况
- 平均情况
- 最差情况

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.
- Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases.
- For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array).
- So we sum all the cases and divide the sum by (n+1).
- Following is the value of average-case time complexity.

Average Case Time = \sum_{i=1}^{n}\frac{\theta (i)}{(n+1)} = \frac{\theta (\frac{(n+1)*(n+2)}{2})}{(n+1)} = \theta (n)

- 在平均案例分析中，我们取所有可能的输入并计算所有输入的计算时间。
- 将所有计算值相加，并除以输入总数。我们必须知道(或预测)病例的分布情况。
- 对于线性搜索问题，我们假设所有的情况都是均匀分布的(包括x不在数组中)。
- 我们把所有情况相加然后除以(n+1)
- 下面是平均案例时间复杂度的值。
- **平均情况下时间**= \ sum_ {i = 1} ^ {n} \压裂{\θ(i)} {(n + 1)} = \压裂{\θ(\压裂{(n + 1) * (n + 2)} {2})} {(n + 1)} = \θ(n)

Types of Algorithm Analysis:
算法分析类型:

- Best case
- Average case
- Worst case

- 最好的情况
- 平均情况
- 最差情况

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm.
- We must know the case that causes a maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched (x) is not present in the array.
- When x is not present, the search() function compares it with all the elements of arr[] one by one.
- Therefore, the worst-case time complexity of the linear search would be O(n).

- 在最坏情况分析中，我们计算算法运行时间的上限。
- 我们必须知道导致执行最大操作数的情况。
- 对于线性搜索，最坏的情况发生在要搜索的元素(x)不在数组中。
- 当x不存在时，search()函数将它与arr[]的所有元素逐一进行比较。
- 因此，线性搜索的最坏情况时间复杂度为O(n)。

Which Complexity analysis is generally used?

The ranked mention of complexity analysis notation based on popularity:

1. Worst Case Analysis:

- Most of the time, we do worst-case analyses to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

2. Average Case Analysis

- The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

3. Best Case Analysis

- The Best-Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

一般使用哪种复杂性分析?

基于流行度的复杂度分析符号提及排序:

1. 最坏情况分析:

- 大多数时候，我们做最坏情况分析来分析算法。在最坏分析中，我们保证算法的运行时间有一个上界，这是一个好的信息。
- **2.平均情况分析**
- 一般的案例分析在大多数实际案例中是不容易做的，很少有人这样做。在平均案例分析中，我们必须知道(或预测)所有可能输入的数学分布。
- **3. 最好的案例分析**
- 最佳情况分析是虚假的。保证一个算法的下界并不能提供任何信息，因为在最坏的情况下，一个算法可能需要数年才能运行。

Θ Notation (theta)

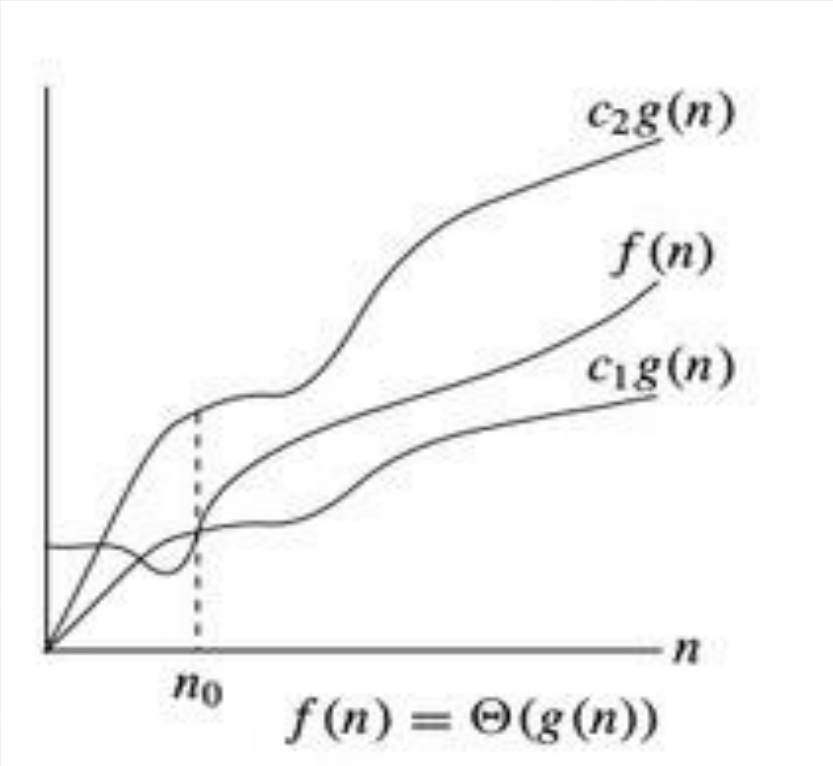
- The Θ Notation is used to find the average bound of an algorithm i.e. it defines an upper bound and a lower bound, and your algorithm will lie in between these levels.
- So, if a function is g(n), then the theta representation is shown as Θ(g(n)) and the relation is shown as:

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c1, c2 and n0 such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq n0 \}$

- The above expression can be read as theta of g(n) is defined as set of all the functions f(n) for which there exists some positive constants c1, c2, and n0 such that c1*g(n) is less than or equal to f(n) and f(n) is less than or equal to c2*g(n) for all n that is greater than or equal to n0.

- For example:

if $f(n) = 2n^2 + 3n + 1$
and $g(n) = n^2$
then for c1 = 2, c2 = 6, and n0 = 1, we can say that $f(n) = \Theta(n^2)$



Θ符号(θ)

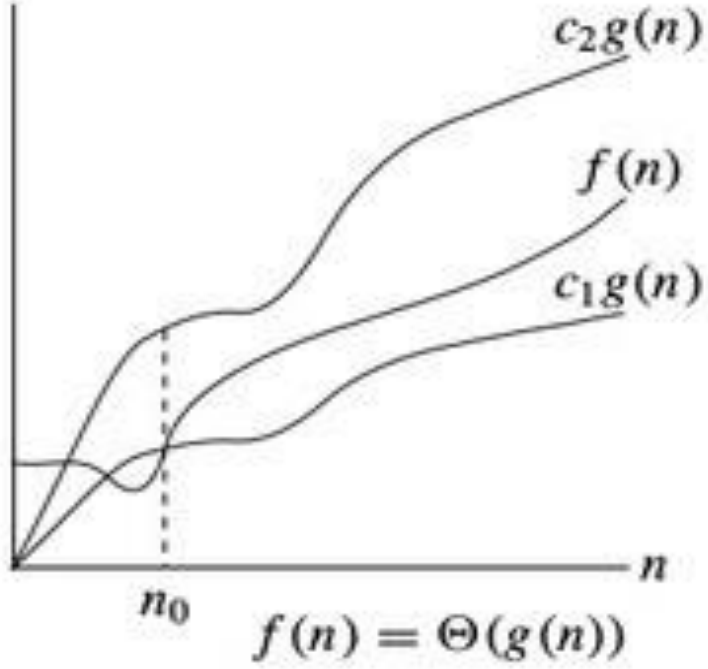
- Θ符号用于找到算法的平均边界，也就是说，它定义了一个上界和下界，你的算法将位于这些级别之间。
- 因此，如果函数为g(n)，则表示为Θ(g(n))，其关系表示为:

$\Theta(g(n)) = \{f(n)$:存在正常数c1, c2, n0令所有$n \geq n0\}$时，$0 \leq c1g(n) \leq f(n) \leq c2g(n)$

- 上面的表达式可以理解为(g(n))被定义为所有函数f(n)的集合，对于这些函数f(n)存在一些正常数c1, c2，和n0，使得c1*g(n)小于等于f(n)且f(n)小于等于c2*g(n)对于所有大于等于n0的n。

- 例如:

如果$f(n) = 2n^2 + 3n + 1$
$g(n) = n^2$
对于c1 = 2 c2 = 6 n0 = 1，我们可以说$f(n) = \Theta(n^2)$



Ω Notation

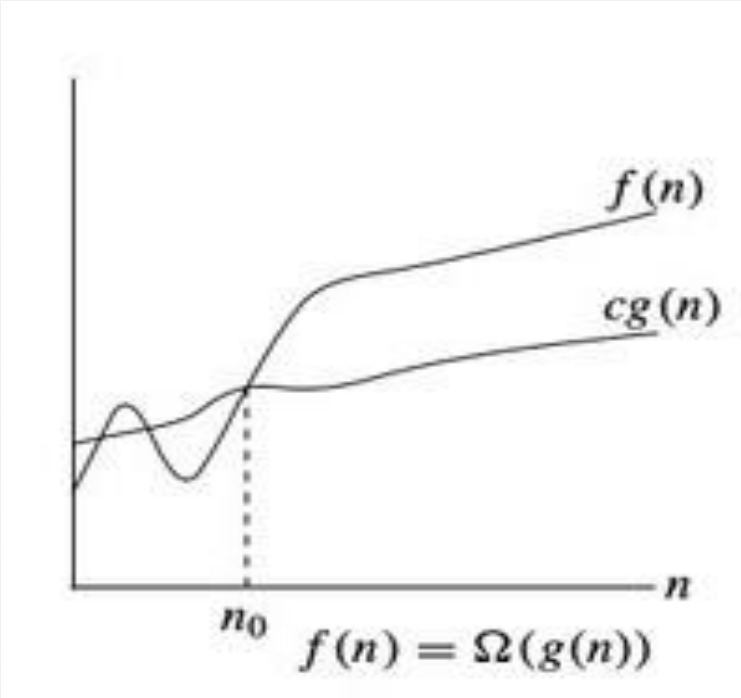
- The Ω notation denotes the lower bound of an algorithm i.e. the time taken by the algorithm can't be lower than this.
- In other words, this is the fastest time in which the algorithm will return a result.
- Its the time taken by the algorithm when provided with its best-case input.
- So, if a function is g(n), then the omega representation is shown as Ω(g(n)) and the relation is shown as:

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0 \}$

The above expression can be read as omega of g(n) is defined as set of all the functions f(n) for which there exist some constants c and n0 such that c*g(n) is less than or equal to f(n), for all n greater than or equal to n0.

- For example:

if $f(n) = 2n^2 + 3n + 1$
and $g(n) = n^2$
then for c = 2 and n0 = 1, we can say that $f(n) = \Omega(n^2)$



Ω符号

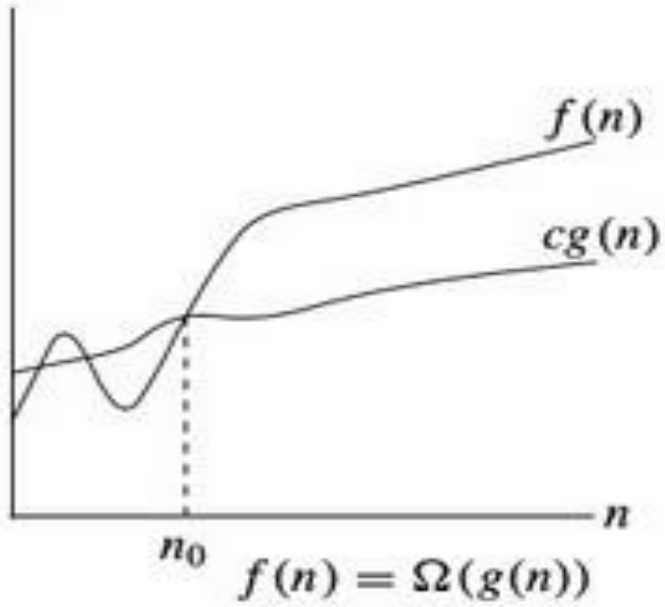
- Ω表示算法的下界，即算法所花费的时间不能低于此值。
- 换句话说，这是算法返回结果的最快时间。
- 它是算法在提供最佳情况输入时所花费的时间。
- 因此，如果函数为g(n)，则表示为Ω(g(n))，其关系表示为:

$\Omega(g(n)) = \{f(n)$:存在正常数c和n0令$0 \le cg(n) \le f(n)$对于所有$n \ge n0\}$

上面的表达式可以理解为(g(n))被定义为所有函数f(n)的集合，f(n)中存在一些常数c和n0，使得c*g(n)小于或等于f(n)，对于所有n大于或等于n0。

- 例如:

如果$f(n) = 2n^2 + 3n + 1$
$g(n) = n^2$
对于c = 2和n0 = 1，我们可以说$f(n) = \Omega(n^2)$



Big O Notation

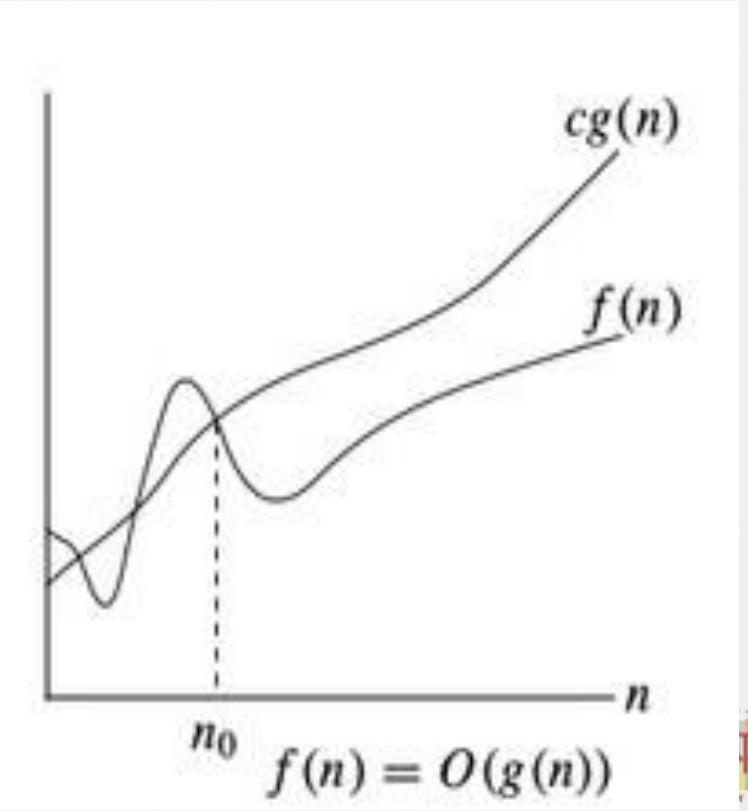
- The Big O notation defines the upper bound of any algorithm i.e. your algorithm can't take more time than this time. In other words, we can say that the big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm.
- So, big O notation is the most used notation for the time complexity of an algorithm. So, if a function is g(n), then the big O representation of g(n) is shown as O(g(n)) and the relation is shown as:

$$O(g(n)) = \{ f(n): \text{there exist positive constants c and n0 such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n0 \}$$

- The above expression can be read as Big O of g(n) is defined as a set of functions f(n) for which there exist some constants c and n0 such that f(n) is greater than or equal to 0 and f(n) is smaller than or equal to c*g(n) for all n greater than or equal to n0.

- For example:

if $f(n) = 2n^2 + 3n + 1$
and $g(n) = n^2$ then for $c = 6$
and $n0 = 1$, we can say that $f(n) = O(n^2)$



大O符号

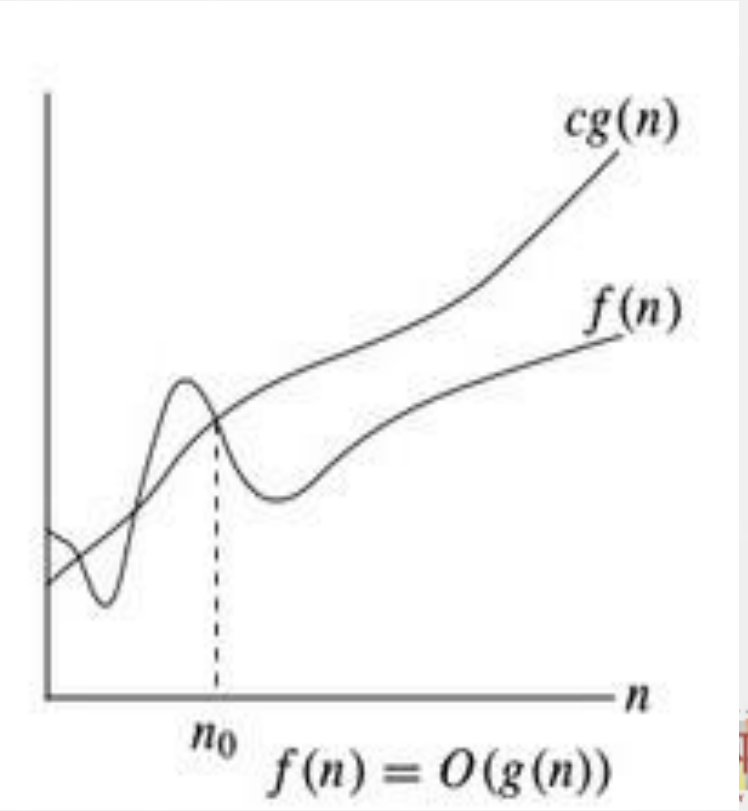
- 大O符号定义了任何算法的上限，即你的算法不能比这个时间花费更多的时间。换句话说，我们可以说，大O符号表示一个算法所花费的最大时间，或者一个算法的最坏情况时间复杂度。
- 大O符号是算法时间复杂度最常用的符号。所以，如果一个函数是g(n)，那么g(n)的大O表示表示为O(g(n))，其关系表示为:

$O(g(n)) = \{f(n)$:存在正常数c和n0令$0 \le f(n) \le cg(n)$对于所有$n \ge n0\}$

- 上面的表达式可以理解为，Big O (g(n))被定义为一组函数f(n)，其中存在一些常数c和n0，使得对于所有n大于等于n0的函数f(n)大于等于0且f(n)小于等于c*g(n)。

- 例如:

如果$f(n) = 2n^2 + 3n + 1$
当$c = 6$时$g(n) = n^2$
当$n0 = 1$时，我们可以说$f(n) = O(n^2)$



Typical Big O Functions – "Grades"

Function	Common Name
N!	Factorial 阶乘
2^N	Exponential 指数
N^d, d > 3	Polynomial 多项式
N^3	Cubic 立方
N^2	Quadratic 平方
$\sqrt{N}$	N Square root N N 平方根
N log N	N log N
N	Linear 线性
$\sqrt[n]{N}$	Root - n
log N	Logarithmic
1	Constant

Running time grows 'quickly' with more input.

随着输入的增加，运行时间会“快速”增长。

随着输入的增加，运行时间增长“缓慢”。

Running time grows 'slowly' with more input.

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

- log n
- n(linear)
- n log n
- n^2
- n^3

多项式时间 (简单或容易处理)

polynomial time (easy or tractable)

指数时间 (硬或棘手的)

2^n

n!

exponential time (hard or intractable)

notation	name
O(1)	constant
O(n)	linear
O(logn)	logarithmic
O(n^2)	quadratic
O(n^c)	polynomial
O(c^n)	exponential
O(n!)	factorial

$$O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n) < O(n!) < O(n^n)$$

Grades…….

N更多的 时间更多	最好的之一 当n多的时候并不多	快速排序	冒泡排序		不太好
N more time more	One of best When n is more time Is not more	Fast sorting	Bubble sorting		Not good

N	log 2N	N log 2N	N^2	N^3	2^N	N!
1	0	0	1	1	2	1
10	3.3	33	100	1000	1024	3628800
50	6.5	282	2500	125000	64digit number	64 digit number
100	6.6	664	10000	1000000	30 digit number	157 digit number

If N more then the time will increase

如果N多，那么时间就会增加

Grades.....

Assume that we have a machine that can execute 1,000,000 required operations per sec

假设我们有一台每秒钟可以执行1,000,000个所需操作的机器

	Algorithm 1 **算法1**	**Algorithm 2** **算法2**	**Algorithm 3** **算法3**	**Algorithm 4** **算法4**	**Algorithm 5** **算法5**
Frequency **频率** **Count** **数**	33n	6nlogn	$13n^2$	$3.4n^3$	2^n
n=10 **n=10,000**	< 1 sec < 1 sec	< 1 sec < 1 sec	< 1 sec 22 min	< 1 sec 39 days	< 1 sec many many centuries

Execution time for algorithms with the given time complexities

给定时间复杂度的算法的执行时间

Reference

- https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm

江西理工大学
Jiangxi University of Science and Technology
信息工程学院
School of information engineering

高级算法分析与设计

Advanced Algorithm Analysis and Design

THANK YOU 谢谢你

"The beauty of research is that you never know where it's going to lead."

RICHARD ROBERTS
Nobel Prize in Physiology or Medicine 1993

"BE HUMBLE. BE HUNGRY.
AND ALWAYS BE THE
HARDEST WORKER
IN THE ROOM."