武漢大學
WUHAN UNIVERSITY

数据结构与算法

算法分析

张晓平

武汉大学数学与统计学院

Table of contents

实验研究

实验研究

统计运行时间

```
from time import time
start_time = time()
run algorithm
end_time = time()
elapsed = end_time - start_time
```

实验分析的局限性

- 需要在相同的硬件和软件环境下做实验对比
- 只能做有限次数的测试实验，不可能穷尽所有可能性. 而验证一个算法的效率应考虑所有可能的情况，这显然是不可能做到的.

实验研究

实验分析之外

实验分析之外

欲分析一个算法的效率，我们希望

- 不依赖软硬件环境就能评估两个算法的相对效率；
- 即使未具体实现算法，也能对算法的性能有足够多的认识；
- 考虑所有可能的输入.

统计基本操作

定义一个基本操作的集合:

- 赋值操作
- 算术操作 (如加减乘除)
- 比较操作
- 利用索引访问 Python 列表的元素
- 函数调用
- 从函数返回

一个基本操作通常对应一个常数运行时间的低级指令.

统计基本操作

一般地，我们会利用基本操作执行的次数来衡量一个算法的运行时间，而没有必要确定每个基本操作的执行时间.

当然这里有一个隐式的假设，即不同基本操作的运行时间大致相当，这就意味着一个算法执行基本操作的次数正比于其实际运行时间.

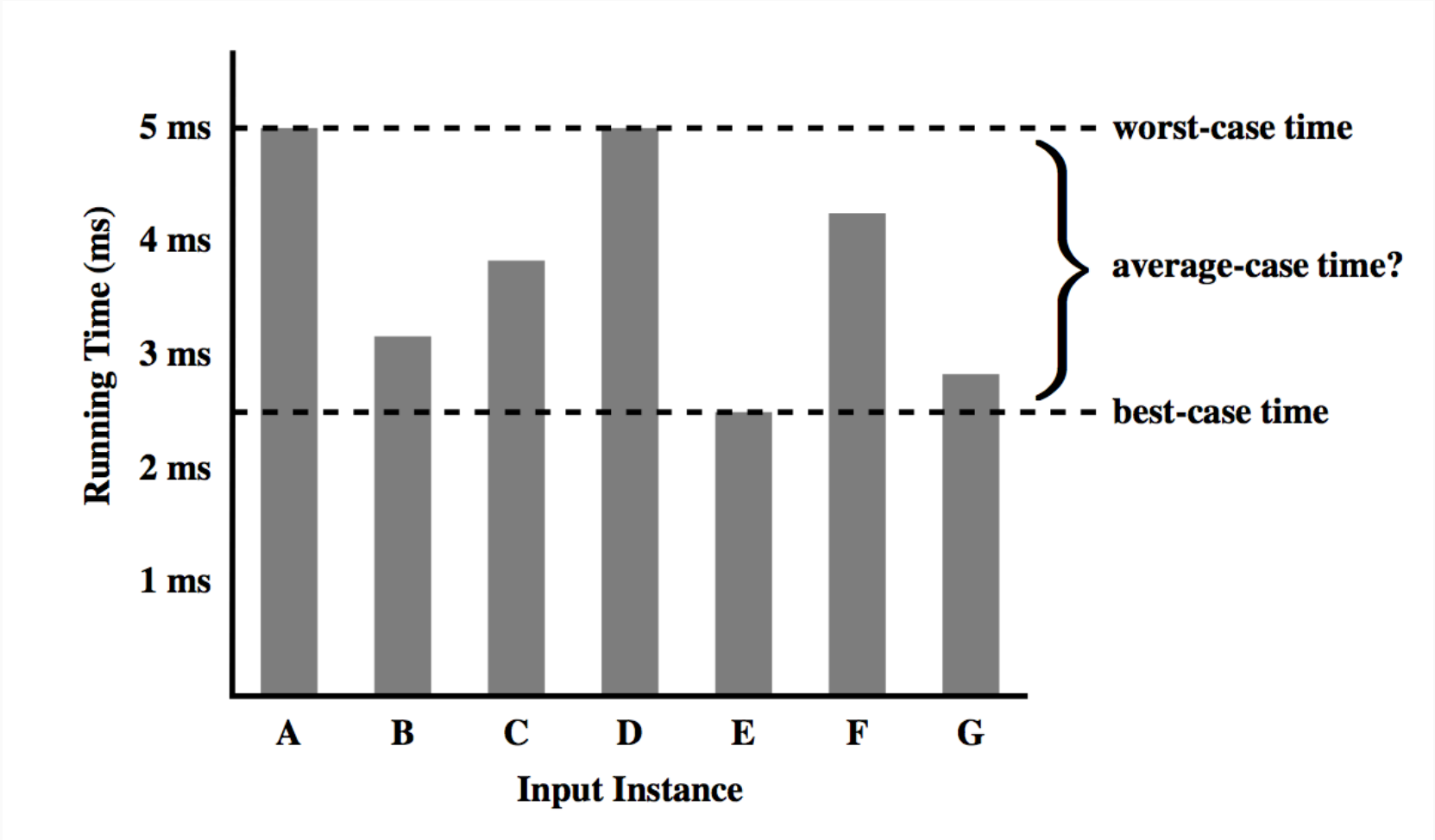
将操作看做是一个输入尺寸的函数

为衡量一个算法的运行时间，通常会定义一个函数 $f(n)$ 来描述基本操作的次数，它可看做是输入尺寸 n 的一个函数.

关注最坏情形

- 平均情况分析
  - 不容易做到，因为我们首先得定义输入集合的概率分布
- 最坏情况分析
  - 比较容易，它只需确定最坏情形的输入
  - 通常会得到更好的算法

关注最坏情形


5 ms
4 ms
3 ms
2 ms
1 ms
Running Time (ms)
worst-case time
average-case time?
best-case time
A
B
C
D
E
F
G
Input Instance

算法分析中常用的七个函数

算法分析中常用的七个函数

常函数	对数函数	线性函数	n-log-n	二次函数	三次函数	指数函数
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

算法分析中常用的七个函数

- 理想情况下，希望数据结构中每个操作的时间复杂度正比于常数函数或者对数函数，而算法的时间复杂度正比于线性函数或者 n-log-n 函数；
- 时间复杂度为二次或三次函数的算法不太实用
- 时间复杂度为指数函数的算法通常认为不可行，除非对一些小尺寸的输入

算法分析中常用的七个函数

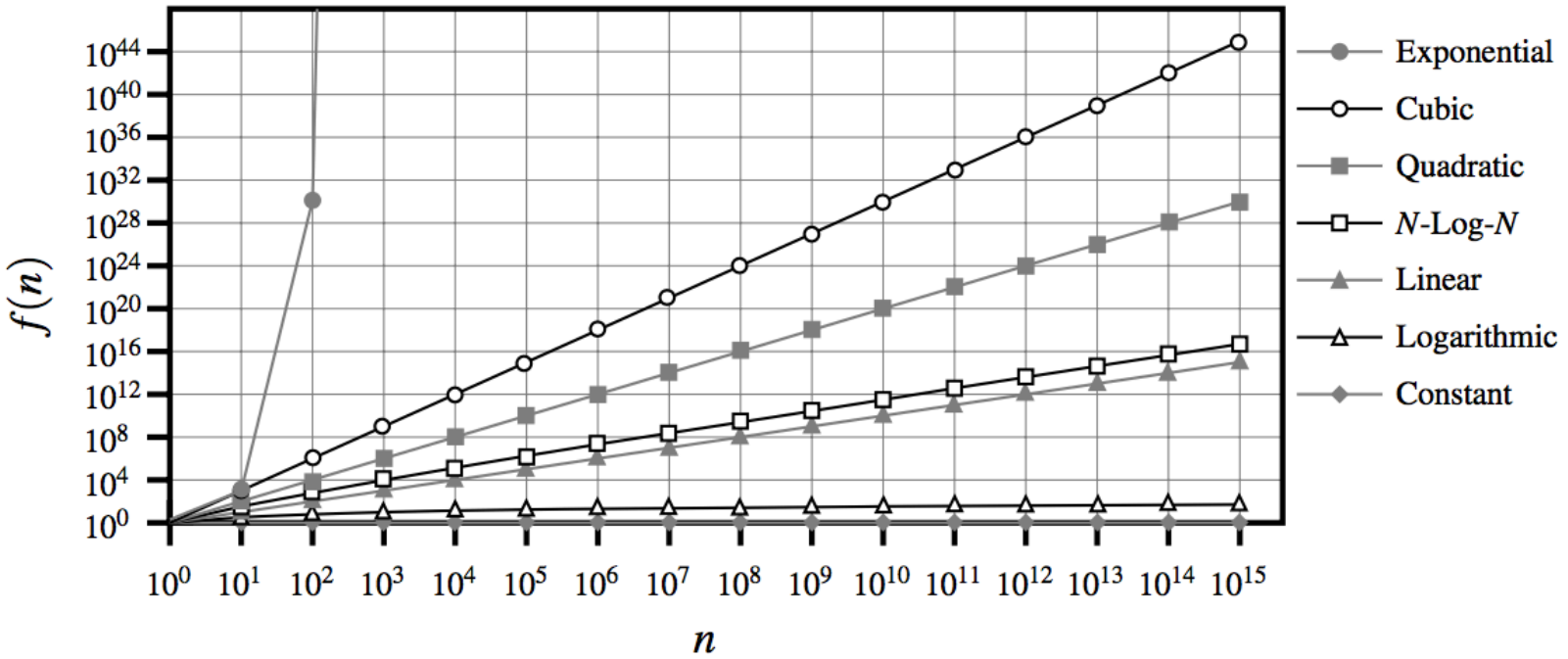


Figure 3.4: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

渐近分析

渐近分析

例

```
def find_max(data):
  biggest = data[0]   # O(1)
  for val in data:    # O(n^2)
    if val > biggest:
      biggest = val
  return biggest      # O(1)
```

渐近分析

例

```
def find_max(data):
  biggest = data[0]   # O(1)
  for val in data:    # O(n^2)
    if val > biggest:
      biggest = val
  return biggest      # O(1)
```

时间复杂度为 $O(n)$.

渐近分析

大 O 记号

大 O 记号

定义：大 O 记号

设 $f(n), g(n) : \mathbb{Z}^+ \to \mathbb{R}^+$ 为两个关于 n 的函数。称$f(n) = O(g(n))$，若存在一个实常数 $c > 0$ 和一个整常数 $n_0 \geq 1$ 使得

$$f(n) \leq cg(n), \quad n \geq n_0.$$

大 O 记号

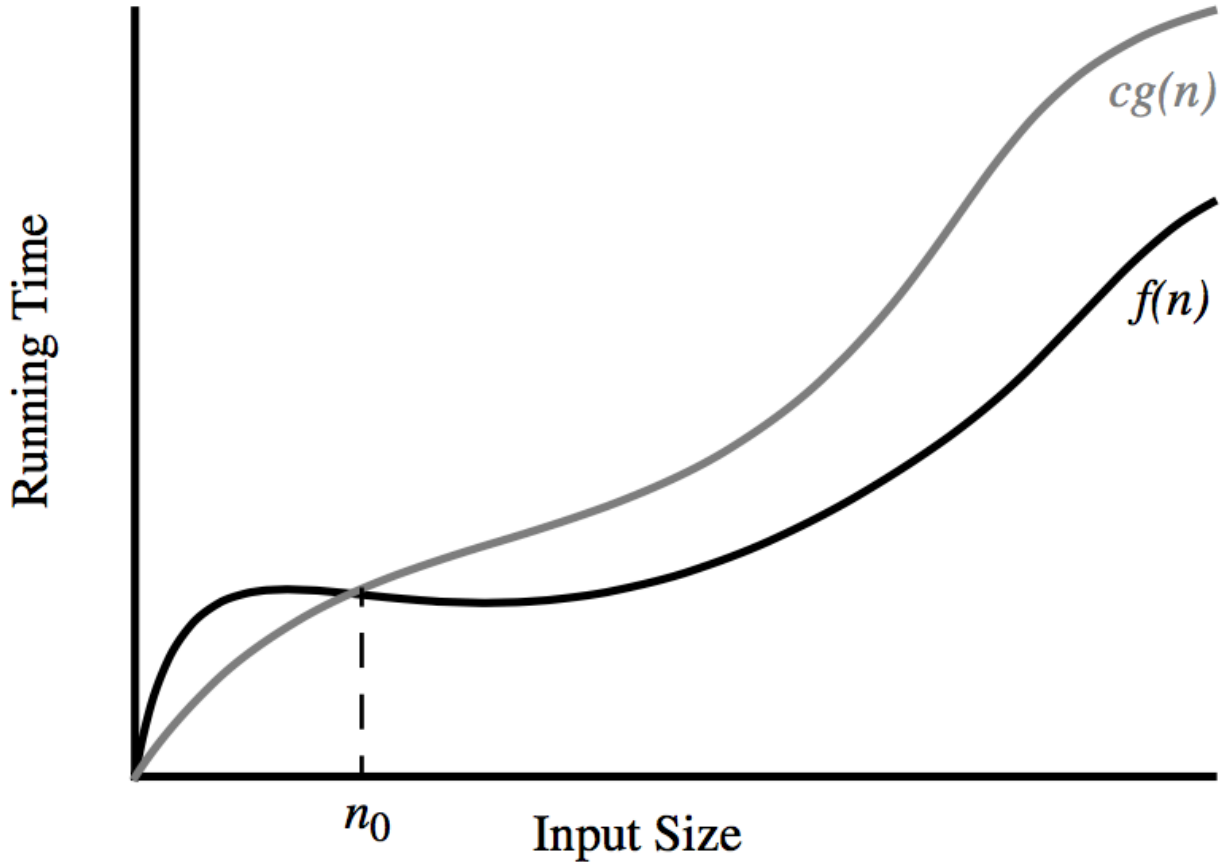


Figure 3.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, s
$f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

大 O 记号的一些性质

大 O 记号允许我们在分析算法复杂度时可以忽略一个函数的常数项和低阶项，而只关注其主部。

例

$5n^4 + 3n^3 + 2n^2 + 4n + 1 = O(n^4)$.

命题

若 $f(n)$ 为次数为 d 的多项式，即

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d, \quad a_d > 0,$$

则 $f(n) = O(n^d)$.

大 O 记号的一些性质

例

- $5n^2 + 3n\log n + 2n + 5 = O(n^2)$
- $20n^3 + 10n\log n + 5 = O(n^3)$
- $3\log n + 2 = O(\log n)$
- $2^{n+2} = O(2^n)$
- $2n + 100\log n = O(n)$

渐近分析

对比分析

对比分析

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 3.2: Selected values of fundamental functions in algorithm analysis.

对比分析

Running Time (μs)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 3.3: Maximum size of a problem that can be solved in 1 second, 1 minute and 1 hour, for various running times measured in microseconds.

渐近分析

算法分析的一些例子

常数时间操作

例

给定一个列表`data`，

- 函数调用：`len(data)`
- 访问元素：`data[j]`

时间复杂度均为 $O(1)$

线性时间操作

求一个序列的最大值

```
def find_max(data):
  biggest = data[0]   # O(1)
  for val in data:    # O(n^2)
    if val > biggest:
      biggest = val
  return biggest      # O(1)
```

前缀平均 (Prefix Average)

例：前缀平均

给定一个由 n 个数构成的序列 S，求另一个序列 A 使得其元素满足

$$A_j = \frac{\sum_{i=0}^{j} S_i}{j+1}, \quad j = 0, \cdots, n-1.$$

前缀平均: 平方时间算法

```
def prefix_average1(S):
  n = len(S)                        # O(1)
  A = [0] * n                       # O(n)
  for j in range(n):
    total = 0                       # O(n)
    for i in range(j + 1):          # 1+...+n=O(n^2)
      total += S[i]
    A[j] = total / (j+1)            # O(n)
  return A
```

时间复杂度为 $O(n^2)$.

前缀平均: 另一个平方时间算法

```
def prefix_average2(S):
  n = len(S)    # O(1)
  A = [0] * n   # O(n)
  for j in range(n):
    A[j] = sum(S[0:j+1])/(j+1) # 1+...+n=O(n^2)
  return A      # O(1)
```

时间复杂度为 $O(n^2)$.

前缀平均: 线性时间算法

```
def prefix_average3(S):
  n = len(S)    # O(1)
  A = [0] * n   # O(n)
  total = 0     # O(1)
  for j in range(n):
    total += S[j]          # O(n)
    A[j] = total / (j+1) # O(n)
  return A
```

时间复杂度为 $O(n)$.

三路集合不相交 (Three-Way Set Disjointness)

定义：三路集合不相交

给定三个数集 A, B, C，每个集合都不存在相同的元素。三路集合不相交问题是判定三个集合的交集是否为空，即是否不存在元素 x 使得 $x \in A, x \in B, x \in C$.

三路集合不相交

```
def disjoint1(A, B, C):
  for a in A:
    for b in B:
      for c in C:
        if a == b == c: # O(n^3)
          return False
  return True
```

很显然，三层嵌套，对于长度都为 n 的 A, B, C 来说，总的时间复杂度为 $O(n^3)$.

三路集合不相交

```
def disjoint2(A, B, C):
  for a in A:
    for b in B:
      if a == b:      # O(n^2)
        for c in C:
          if a == c:
            return False
  return True
```

三路集合不相交

```
def disjoint2(A, B, C):
  for a in A:
    for b in B:
      if a == b:          # O(n^2)
        for c in C:
          if a == c:
            return False
  return True
```

算法复杂度分析

在if a== b分支中，由于集合的互异性，一旦 $a == b$ 成立，则 a 不等于在 B 中除 b 以外的元素。而 $a == b$ 的次数最多为 n，故对最内层循环，其时间复杂度为 $O(n^2)$

元素唯一性（元素唯一性）

定义：元素唯一性

对一个长度为 n 的序列，判断其元素是否互异?

元素唯一性

```
def unique1(S):
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]: # (n-1)+...+1=O(n^2)
        return False
  return True
```

时间复杂度为 $O(n^2)$.

元素唯一性

```
def unique2(S):
  temp = sorted(S)  # O(n log n)
  for j in range(1, len(temp)):
    if S[j-1] == S[j]: # O(n)
      return False
  return True
```

时间复杂度为 $O(n \log n)$.