

Analysis of Algorithms

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Introduction

- 🌐 The purpose of algorithm analysis is to **predict the behavior** (running time, space requirement, etc.) of an algorithm *without implementing it* on a specific computer. (Why?)
- 🌐 As the exact behavior of an algorithm is hard to predict, the analysis is usually an *approximation*:
 - ☀️ **Relative to the input size** (usually denoted by n): input possibilities too enormous to elaborate
 - ☀️ **Asymptotic**: should care more about larger inputs
 - ☀️ **Worst-Case**: easier to do, often representative (Why not average-case?)
- 🌐 Such an approximation is usually good enough for **comparing** different algorithms for the same problem.

Complexity

- 🌐 Theoretically, “complexity of an algorithm” is a more precise term for “approximate behavior of an algorithm”.
- 🌐 Two most important measures of complexity:
 - ☀️ **Time Complexity**
an upper bound on the number of steps that the algorithm performs.
 - ☀️ **Space Complexity**
an upper bound on the amount of temporary storage required for running the algorithm (excluding the input, the output, and the program itself).
- 🌐 We will focus on time complexity.

Comparing Running Times

- 🌐 How do we compare the following running times?
 1. $100n$
 2. $2n^2 + 50$
 3. $100n^{1.8}$
- 🌐 We will study an approach (the O notation) that allows us to ignore constant factors and concentrate on the behavior *as n goes to infinity*.
- 🌐 For most algorithms, the constants in the expressions of their running times tend to be small.


The O Notation

- 🌐 A function $g(n)$ is $O(f(n))$ for another function $f(n)$ if there exist constants c and N such that, for all $n \geq N$, $g(n) \leq cf(n)$.
- 🌐 The function $g(n)$ may be substantially less than $cf(n)$; the O notation bounds it *only from above*.
- 🌐 The O notation allows us to **ignore constants** conveniently.
- 🌐 Examples:
 - ☀ $5n^2 + 15 = O(n^2)$.
(cf. $5n^2 + 15 \leq O(n^2)$ or $5n^2 + 15 \in O(n^2)$)
 - ☀ $5n^2 + 15 = O(n^3)$.
(cf. $5n^2 + 15 \leq O(n^3)$ or $5n^2 + 15 \in O(n^3)$)
 - ☀ In an expression, $T(n) = 3n^2 + O(n)$.

The O Notation (cont.)


- 🌐 No need to specify the base of a logarithm.
 - ☀ $\log_2 n = \frac{\log_{10} n}{\log_{10} 2} = \frac{1}{\log_{10} 2} \log_{10} n$.
 - ☀ For example, we can just write $O(\log n)$.
- 🌐 $O(1)$ denotes a constant.

Properties of O

 We can add and multiply with O .

Lemma (3.2)

1. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then $f(n) + g(n) = O(s(n) + r(n))$.
2. If $f(n) = O(s(n))$ and $g(n) = O(r(n))$, then $f(n) \cdot g(n) = O(s(n) \cdot r(n))$.

 However, we cannot subtract or divide with O .

Polynomial vs. Exponential

- 🌍 A function $f(n)$ is *monotonically growing* if $n_1 \geq n_2$ implies that $f(n_1) \geq f(n_2)$.
- 🌍 An exponential function grows *at least* as fast as a polynomial function does.

Theorem (3.1)

For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$, $(f(n))^c = O(a^{f(n)})$.

🌍 Examples:

- ☀️ Take n as $f(n)$, $n^c = O(a^n)$.
- ☀️ Take $\log_a n$ as $f(n)$, $(\log_a n)^c = O(a^{\log_a n}) = O(n)$.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

Table 1.7 Function values

Source: E. Horowitz *et al.*, 1998.

Speed of Growth (cont.)

	<i>time 1</i>	<i>time 2</i>	<i>time 3</i>	<i>time 4</i>
running times	1000 steps/sec	2000 steps/sec	4000 steps/sec	8000 steps/sec
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Table 3.1 Running times (in seconds) under different assumptions ($n = 1000$)

Source: Manber 1989.

O , o , Ω , and Θ

- Let $T(n)$ be the number of steps required to solve a given problem for input size n .
- We say that $T(n) = \Omega(g(n))$ or the problem has a lower bound of $\Omega(g(n))$ if there exist constants c and N such that, for all $n \geq N$, $T(n) \geq cg(n)$.
- If a certain function $f(n)$ satisfies both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we say that $f(n) = \Theta(g(n))$.
- We say that $f(n) = o(g(n))$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Polynomial vs. Exponential (cont.)

- 🌐 An exponential function grows *faster* than a polynomial function does.

Theorem (3.3)

For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$, we have

$$(f(n))^c = o(a^{f(n)}).$$

- 🌐 Consider a previous example again:
Take $\log_a n$ as $f(n)$. For all $c > 0$ and $a > 1$,

$$(\log_a n)^c = o(a^{\log_a n}) = o(n).$$

Sums

- Techniques for summing expressions are essential for complexity analysis.
- For example, given that we know

$$S_0(n) = \sum_{i=1}^n 1 = n$$

and

$$S_1(n) = \sum_{i=1}^n i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2},$$

we want to compute the sum

$$S_2(n) = \sum_{i=1}^n i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2.$$

Sums (cont.)

From

$$(i + 1)^3 = i^3 + 3i^2 + 3i + 1,$$

we have

$$(i + 1)^3 - i^3 = 3i^2 + 3i + 1.$$

$$2^3 - 1^3 = 3 \times 1^2 + 3 \times 1 + 1$$

$$3^3 - 2^3 = 3 \times 2^2 + 3 \times 2 + 1$$

$$4^3 - 3^3 = 3 \times 3^2 + 3 \times 3 + 1$$

... ..

$$(n + 1)^3 - n^3 = 3 \times n^2 + 3 \times n + 1$$

$$(n + 1)^3 - 1 = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n)$$

$$(S_3(n + 1) - S_3(1)) - S_3(n) = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n)$$

Sums (cont.)

🌐 So, we have

$$(n + 1)^3 - 1 = 3 \times S_2(n) + 3 \times S_1(n) + S_0(n).$$

- 🌐 Given $S_0(n)$ and $S_1(n)$, the sum $S_2(n)$ can be computed by straightforward algebra.
- 🌐 Recall that the left-hand side $(n + 1)^3 - 1$ equals $(S_3(n + 1) - S_3(1)) - S_3(n)$, a result from “**shifting and canceling**” terms of two sums.
- 🌐 This generalizes to $S_k(n)$, for $k > 2$.
- 🌐 Similar shifting and canceling techniques apply to other kinds of sums.

Recurrence Relations

- 🌐 A *recurrence relation* is a way to define a function by an expression involving the same function.
- 🌐 The Fibonacci numbers can be defined as follows:

$$\begin{cases} F(1) = 1 \\ F(2) = 1 \\ F(n) = F(n-2) + F(n-1) \end{cases}$$

We would need $k - 2$ steps to compute $F(k)$.

- 🌐 It is more convenient to have an explicit (or **closed-form**) expression.
- 🌐 To obtain the explicit expression is called *solving* the recurrence relation.

Guessing and Proving an Upper Bound

🌐 Recurrence relation:
$$\begin{cases} T(2) = 1 \\ T(2n) \leq 2T(n) + 2n - 1 \end{cases}$$


🌐 Guess: $T(n) = O(n \log n)$.

🌐 Proof:

1. Base case: $T(2) \leq 2 \log 2$.




2. Inductive step:
$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1 \\ &\leq 2(n \log n) + 2n - 1 \\ &= 2n \log n + 2n \log 2 - 1 \\ &\leq 2n(\log n + \log 2) \\ &= 2n \log 2n \end{aligned}$$

Divide and Conquer Relations

-  The running time $T(n)$ of a divide-and-conquer algorithm satisfies

$$T(n) = aT(n/b) + cn^k$$

where

-  a is the number of subproblems,
-  n/b is the size of each subproblem, and
-  cn^k is the running time of the solutions-combining algorithm.

Divide and Conquer Relations (cont.)

Assume, for simplicity, $n = b^m$ ($\frac{n}{b^m} = 1$, $\frac{n}{b^{m-1}} = b$, etc.).

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + cn^k \\
&= a\left(aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^k\right) + cn^k \\
&= a\left(a\left(aT\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^k\right) + c\left(\frac{n}{b}\right)^k\right) + cn^k \\
&\dots \\
&= a\left(a\left(\dots\left(aT\left(\frac{n}{b^m}\right) + c\left(\frac{n}{b^{m-1}}\right)^k\right) + \dots\right) + c\left(\frac{n}{b}\right)^k\right) + cn^k
\end{aligned}$$

Assuming $T(1) = c$,

$$T(n) = c \sum_{i=0}^m a^{m-i} b^{ik} = ca^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i.$$

Three cases: $\frac{b^k}{a} < 1$, $\frac{b^k}{a} = 1$, and $\frac{b^k}{a} > 1$.

Divide and Conquer Relations (cont.)

Theorem (3.4)

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

Recurrent Relations with Full History

As a reading assignment ...

Useful Facts

Harmonic series

$$H_n = \sum_{k=1}^n \frac{1}{k} = \ln n + \gamma + O(1/n),$$

where $\gamma = 0.577\dots$ is Euler's constant.

Sum of logarithms

$$\begin{aligned} \sum_{i=1}^n \lfloor \log_2 i \rfloor &= (n+1) \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2 \\ &= \Theta(n \log n). \end{aligned}$$

Useful Facts (cont.)

- 🌐 Bounding a summation by an integral:
If $f(x)$ is monotonically increasing, then

$$\sum_{i=1}^n f(i) \leq \int_1^{n+1} f(x) dx.$$

- 🌐 Stirling's approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)).$$