CSE1002 Lecture Notes

Program Analysis

Ryan Stansifer

Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

1 April 2024

What is CS?

A definition of computer science: The study of information, protocols and algorithms for idealized and real automata.

Short Definition
The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, "What can be (efficiently) automated?"

Peter J. Denning et al. (Jan. 1989). "Computing as a discipline". In: *Communications of the ACM* 32.1, pages 9–32, page 12

One important aspect of the study is *efficiency*. We wish to examine performance

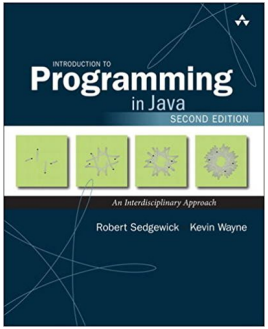
and introduce Big-Oh notation which is used to categorize computer programs quite usefully.

Worst Case (Aside)

"Usefully" does not mean that Big-Oh captures the whole story.
Merge sort $O(n \log n)$ comparisons, but Quick sort $O(n^2)$ comparisons is "better," and Tim's sort is even "better."
Adaptive sort takes advantage of the existing order of the input to try to achieve better times, so that the time taken by the algorithm to sort is a smoothly growing function of the size of the sequence *and* the disorder in the sequence. In other words, the more presorted the input is, the faster it should be sorted.

Textbook

Sedgewick and Wayne, Section 4.1 "Performance" in *Introduction to Programming in Java*.



Goodrich and Tamassia, Chapter 4 "Analysis Tools" in *Data Structures and Algorithms in Java*

Two approaches

1. Analytical. Static analysis of the program. Requires program source. (Mathematical guarantees.)
2. Empirical. Time experiments running the program with different inputs. (Scientific method.)

Profiling

Measuring the time a program takes is difficult. Many factors influence the time: processor, OS, multitasking, input data, resolution of the clock, etc. It is difficult to predict the performance of a program in general based on timing experiments.

Steps, Worse Case

It is plausible that the time it takes a program to execute is proportional to the number of instructions it executes.
This work that a program does can be approximated by the number of operations or steps it calls for–operations like assignment, IO, arithmetic operations and relational comparisons. The size of the steps–10 machine instructions, 100 machine instructions—does not matter in the long run.

When counting the steps of a program we always assume the worse. We assume that the program will "choose" the path that requires the most steps. This way we get an upper bound on the performance.

Input

Useful programs take different steps depending on the input. So, the number of steps a program takes for some particular input does not tell us how good the program is. A bad algorithm may take few steps for some small, simple input; and a good algorithm may take many steps for some large, complex input.

Input

Suppose we count the number of steps in terms of the *size of the input*, call it N. The number of steps is a function of N. For the program which reads N numbers in order to sum them, the number of steps might be $f(N) = 2N + 1$.
What is the size of the input? Many algorithms have a parameter that affects the running time most significantly. For example, the parameter might be the size of the file to be sorted or searched, the number of characters in a string.

The number of steps a programs takes is a *function* of the size of the input.

Asymptotic Notation

We wish to compare functions carefully by their growth. Unimportant information should be ignored, like "rounding" where

$$1,000,001 \approx 1,000,000$$

And we want the "big picture." This means that a function f may be smaller than a function g for some particular values, but "in the long run" it may be larger than g. Fortunately, a precise definition that captures our intuition (most of the time) is possible.

[Detour to pictures comparing.pdf]

Preliminaries

We want a precise, i.e., mathematical way to compare functions.
What kind of functions? Although the usual approach applies to functions $f : \mathbb{R} \to \mathbb{R}$, our context is more restrictive, so we simplify.

We consider our domain to be discrete "sizes," i.e., $\mathbb{N}$, and our domain to be discrete resource units "steps" or "bytes," i.e., $\mathbb{N}$.

Since the input to a program cannot have negative size, and the resources consumed by a program cannot be negative, we restrict ourselves to functions whose graphs are in the first quadrant.

Preliminaries

Let $f(n)$ and $g(n)$ be functions mapping from natural numbers $\mathbb{N}$ (non-negative integers) to $\mathbb{N}$.

$$f : \mathbb{N} \to \mathbb{N} \quad \text{and} \quad g : \mathbb{N} \to \mathbb{N}$$

We need to use traditional, real-valued functions like $f(n) = \log n$, but we can quietly think of them rounded up to the nearest integer, as in $f(n) = \lceil \log n \rceil$.

Big-Oh – $g(n)$ in $O(f(n))$

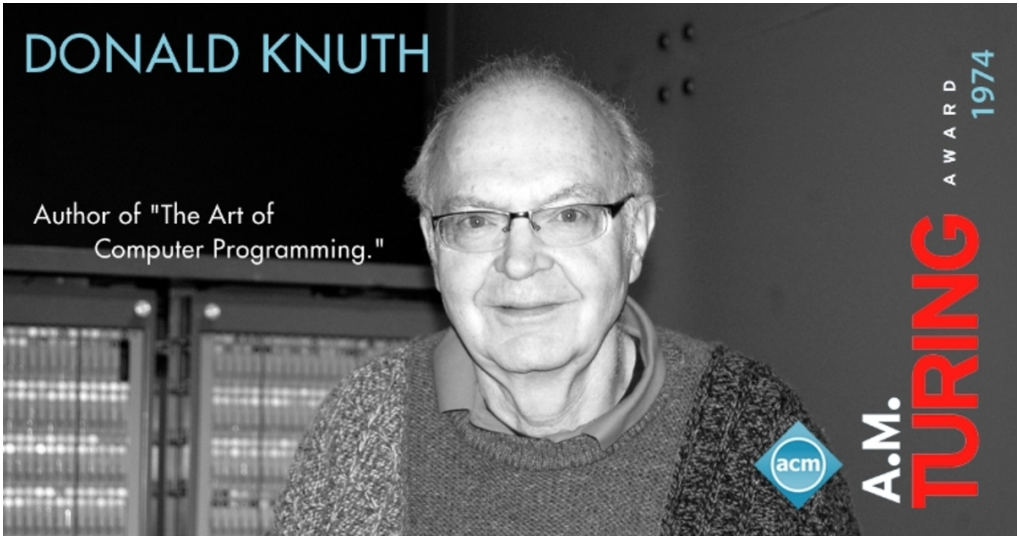
After discussing this problem with people for several years, I have come to the conclusion that the following definitions will prove to be most useful for computer scientists:

$O(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C and n_0 with $|g(n)| \leq Cf(n)$ for all $n \geq n_0$.

$\Omega(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C and n_0 with $g(n) \geq Cf(n)$ for all $n \geq n_0$.

$\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C, C', and n_0 with $Cf(n) \leq g(n) \leq C'f(n)$ for all $n \geq n_0$.

D. Knuth, SIGACT News, 1976.


DONALD KNUTH
Author of "The Art of Computer Programming."
acm
A.M. TURING AWARD
1974

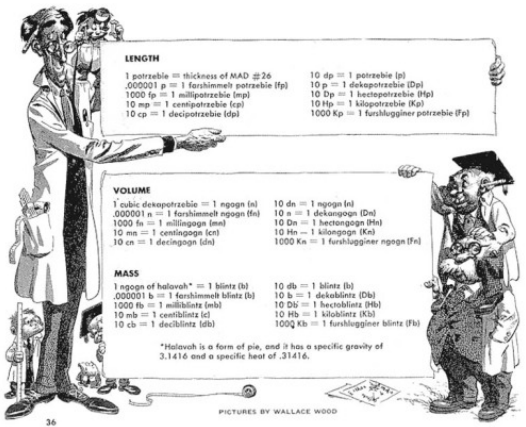
THE POTRZEBIE SYSTEM

OF WEIGHTS AND MEASURES

THE POTRZEBIE SYSTEM

This new system of measuring, which is destined to become the measuring system of the future, has decided improvements over the other systems now in use. It is based upon measurements taken 6-9-12 at the Physics Lab. of Milwaukee Lutheran High School, in Milwaukee, Wis., when the thickness of MAD Magazine #26 was determined to be 2.26334851-7438173216473 mm. This length is the basis for the entire system, and is called one potrzebie of length.

The Potrzebie has also been standardized at 3515.-3502 wave lengths of the red line in the spectrum of cadmium. A partial table of the Potrzebie System, the measuring system of the future, is given below.



LENGTH

1 potrzebie = thickness of MAD #26
.000001 p = 1 farshimmelt potrzebie (fp)
1000 fp = 1 millipotrzebie (mp)
10 mp = 1 centipotrzebie (cp)
10 cp = 1 decipotrzebie (dp)
10 dp = 1 potrzebie (p)
10 p = 1 dekapotrzebie (Dp)
10 Dp = 1 hectopotrzebie (Hp)
10 Hp = 1 kilopotrzebie (Kp)
1000 Kp = 1 furshlugginer potrzebie (Fp)

VOLUME

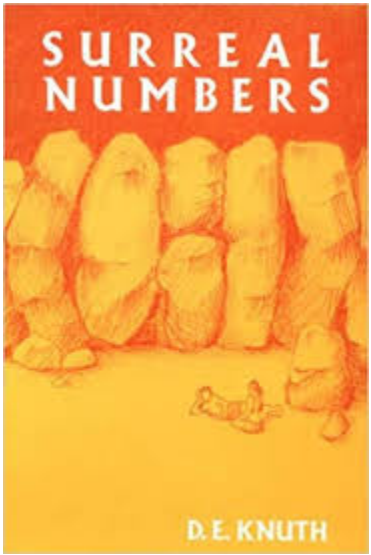
1 cubic dekapotrzebie = 1 ngogn (n)
.000001 n = 1 farshimmelt ngogn (fn)
1000 fn = 1 millingogn (mn)
10 mn = 1 centingogn (cn)
10 cn = 1 decingogn (dn)
10 dn = 1 ngogn (n)
10 n = 1 dekangogn (Dn)
10 Dn = 1 hectongogn (Hn)
10 Hn = 1 kilongogn (Kn)
1000 Kn = 1 furshlugginer ngogn (Fn)

MASS

1 ngogn of halavah* = 1 blintz (b)
.000001 b = 1 farshimmelt blintz (fb)
1000 fb = 1 millblintz (mb)
10 mb = 1 centiblintz (cb)
10 cb = 1 deciblintz (db)
10 db = 1 blintz (b)
10 b = 1 dekablintz (Db)
10 Db = 1 hectoblintz (Hb)
10 Hb = 1 kiloblintz (Kb)
1000 Kb = 1 furshlugginer blintz (Fb)

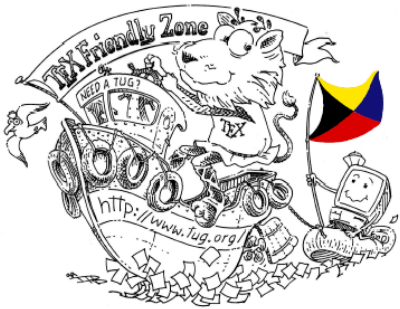
*Halavah is a form of pie, and it has a specific gravity of 3.1416 and a specific heat of .31416.

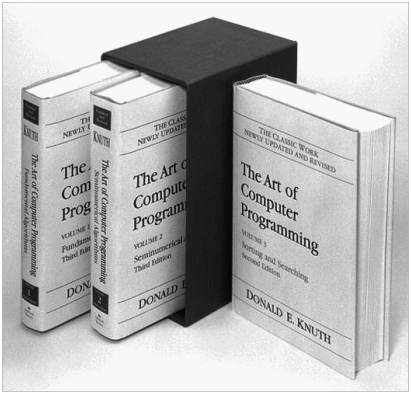
PICTURES BY WALLACE WOOD


SURREAL
NUMBERS
D. E. KNUTH

Donald E. Knuth (1938–)

Introduction to Knuth's organ composition YouTube [17 minutes]


TEX Friendly Zone
NEED A TUG?
http://www.tug.org/


The Art of Computer Programming
VOLUME 3
Sorting and Searching
Second Edition
DONALD E. KNUTH

Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it.

Knuth, Turing Award Lecture, 1974.

Science is what we understand well enough to explain to a computer. Art is everything else we do.

Knuth, 1995, foreword to the book $A = B$, page xi.

[Software] is harder than anything else I've ever had to do.

Knuth, *Notices of the AMS*, 49 (3), 2002, page 320, 2002

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Knuth, "Literate Programming," *The Computer Journal*, volume 27, 1984.

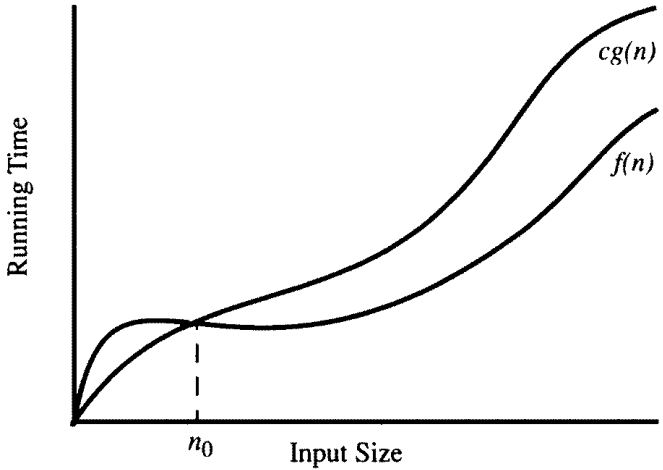
The point in my words: "Writing a computer program *or a proof* requires understanding the solution to a problem so well you can explain it to a mindless automaton, and yet express it so eloquently a fellow human can rapidly apprehend the method."

Biographies appear in:

- O'Regan, *Giants of Computing: A Compendium of Select, Pivotal Pioneers*, 2013
- Shasha and Lazere, *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists*, 1995
- Slater, *Portraits in Silicon*, 1987

To define the Big-Oh notation, we first give a diagram, then Knuth's original definition (in which the roles of f and g are swapped), and finally our definition.

Big-Oh – $f(n)$ is $O(g(n))$



Categorizing functions [f in g]

Let $f(n)$ and $g(n)$ be functions mapping non-negative numbers to non-negative numbers.

Big-Oh. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

Categorizing functions [f in g]

Let $f(n)$ and $g(n)$ be functions mapping non-negative numbers to non-negative numbers.

Big-Oh. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

Big-Omega. $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n_0$.

Big-Theta. $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$.

Little-Oh. $f(n)$ is $o(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

Little-Omega. $f(n)$ is $\Omega(g(n))$ if for any $c > 0$ there is $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for every number $n \geq n_0$.

$$
\begin{aligned}
f(n) \text{ is } O(g(n)) &\approx x \leq y \\
f(n) \text{ is } \Theta(g(n)) &\approx x = y \\
f(n) \text{ is } \Omega(g(n)) &\approx x \geq y \\
f(n) \text{ is } o(g(n)) &\approx x < y \\
f(n) \text{ is } \omega(g(n)) &\approx x > y
\end{aligned}
$$

The analogy is rough since some functions are not comparable, while any two real numbers are comparable.

Categorizing functions

There is a family or related notions, however, $O(n)$ is the only notion required at the moment.
You are asked to commit the definition to memory now. Eventually (e.g., in Algorithms and Data Structures), you will be expected to have a deeper understanding of these notions.

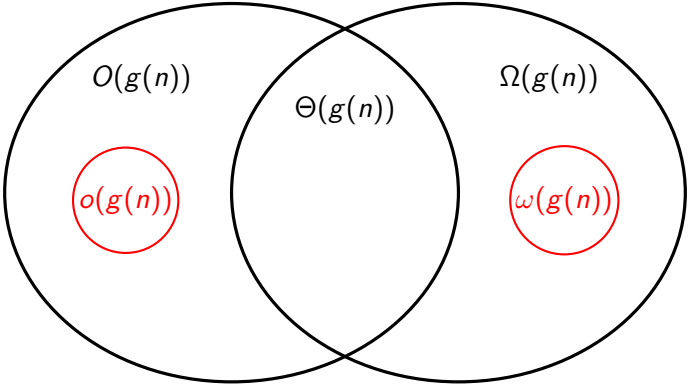
Big-Oh and Connection to Limits [f in g]

Let $f(n)$ and $g(n)$ be functions mapping non-negative real numbers to non-negative real numbers.

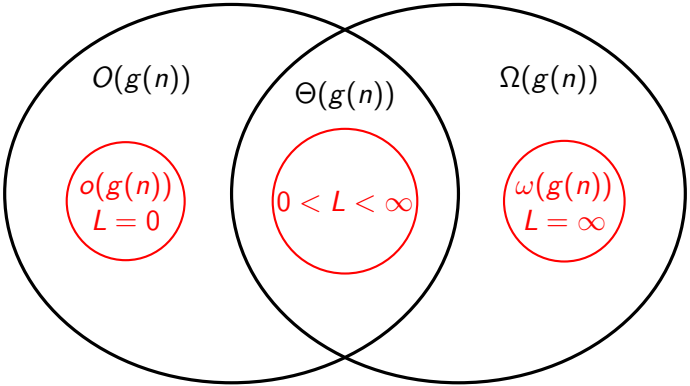
Big-Oh. $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every number $n \geq n_0$.

Lemma. $f(n)$ is $O(g(n))$ if (but not only if) $\lim_{n\to\infty} f(n)/g(n) = L$ where $0 < L < \infty$.
Lemma. $f(n)$ is $O(g(n))$ if, and only, if $\limsup_{n\to\infty} f(n)/g(n) = L$ where $0 < L < \infty$.

Relationships



Relationships



Here L denotes the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Polynomial

In mathematics, a polynomial is an expression consisting of indeterminates and coefficients, that involves only the operations of addition, subtraction, multiplication, and positive-integer powers of variables. An example of a polynomial of a single indeterminate x is $x^2 - 4x + 7$.

From Wikipedia

Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.

Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.

$f(n) = 4 \cdot n + 17$ is $O(n)$?

Example

The function $f(n) = 3 \cdot n + 17$ is $O(n)$. (Here $g(n) = n$.)
Proof. Take $c = 4$ and $n_0 = 17$. Then $f(n) = 3 \cdot n + 17 \leq c \cdot g(n)$ for every $n \geq n_0$. because $3 \cdot n + 17 \leq 4 \cdot n = 3 \cdot n + n$ for every $n \geq 17$.

$f(n) = 4 \cdot n + 17$ is $O(n)$?

$f(n) = 3 \cdot n + 88$ is $O(n)$?

Properties

In what follows, let $d(n)$, $f(n)$, $g(n)$, and $h(n)$ be functions mapping nonnegative integers to nonnegative integers.

Example: $5n^4 + 6n^3 + 7n^2 + 4n + 1$ is in $O(n^4)$. Because $5n^4 + 6n^3 + 7n^2 + 4n + 1 \leq (5 + 6 + 7 + 4 + 1)n^4 = 23n$; take $n \geq 1$ and $c = 23$.
In general, for all polynomials

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

$f(n)$ is $O(n^d)$. Because

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d \leq (a_0 + a_1 n + \cdots + a_d) n^d$$

take $n \geq 1$ and $c = a_0 + a_1 n + \cdots + a_d$.

Properties

In what follows, let $d(n)$, $f(n)$, $g(n)$, and $h(n)$ be functions mapping nonnegative integers to nonnegative integers.

Example, if $n+9$ is in $O(2n+1)$, and $2n+1$ is in $O(n^2/9)$, then $n+9$ is in $O(n^2/9)$
In general, if $d(n)$ is in $O(f(n))$, and $f(n)$ is in $O(g(n))$, then $d(n)$ is in $O(g(n))$.

Properties

- For all functions f, $f(n)$ is in $O(f(n))$.
- If $f(n) \leq g(n)$ for all $n_0 < n$, then $f(n)$ is in $O(g(n))$.
- If $d(n)$ is in $O(f(n))$, then $ad(n)$ is in $O(f(n)$, for any constant $a > 0$.
- If $d(n)$ is in $O(f(n))$, and $f(n)$ is in $O(g(n))$, then $d(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g(n))$ and $f_2(n)$ is in $O(g(n))$, then $f_1(n) + f_2(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(g_1(n) + g_2(n))$.
- n^x is in $O(a^n)$ for any constants $x > 0$ and $a > 1$.

Using the Big-Oh Notation

The notation is strange and even bad. It is difficult to use. [The language of mathematics has (and this is quite amazing) dealt very poorly with functions. Church's lambda notation is not widely used.]
The idea is simple: a function gives rise to a collection of functions containing that function and other functions.

It is best to write "$f(n)$ is $O(g(n))$" spoken f of n is in big-oh of g of n.

Some authors write $f(n) \in O(g(n))$, or even $f(n) = O(g(n))$, but I find this misleading.

Big-Oh Math

Lemma: If $d(n)$ is $O(f(n))$, then $a \times d(n)$ is $O(f(n))$, for any constant $a > 0$. Just take $c = a \times c_1$.

Another fact: If $f(n)$ and $h(n)$ are both $O(g(n))$, then $f(n) + h(n)$ is $O(g(n))$; just take $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$.

Finally: If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$.

For example, if $f(n) = an^2 + bn + c$, then it is $O(n^2)$.

Big-Oh Math

Lemma: n^d is in $O(n^{d+1})$

Big-Oh Math

Fact: $f(n) = n$ is $O(2^n)$ because, by induction, $n < 2^n$ for all n.

Another fact: $2^{n+4} = 2^4 \times 2^n < (2^4 + 1) \times 2^n$, so take $c = 2^4 + 1$ and therefore, 2^{n+4} is $O(2^n)$.

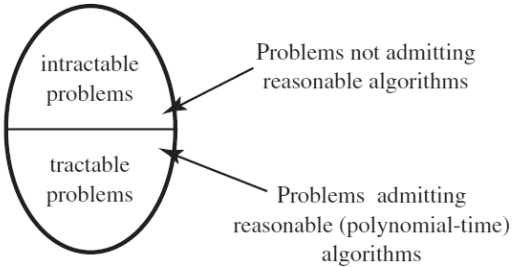
Important Categories of Functions

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	loglinear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Intractable Problems

Figure 7.6

The sphere of algorithmic problems: Version I.

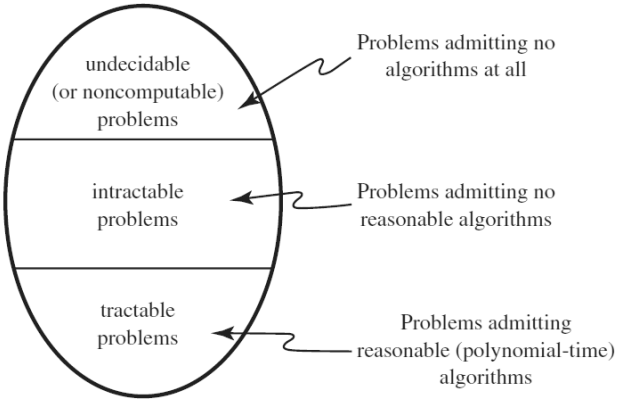


Harel 3rd

Unsolvable Problems

Figure 8.3

The sphere of algorithmic problems: Version II.



Harel 3rd

Intractable Problems

A problem is said to be intractable if the algorithm takes an impractical amount of time to find the solution.
Roughly speaking, we consider polynomial algorithms to be tractable and exponential algorithms to be impractical.

Many important problems (NP complete problems) are thought to be intractable no matter what algorithm is used.

- Important: traveling salesman, Boolean satisfiability, scheduling, packing
- One algorithm solves them all
- Great unsolved problems of mathematics. The Clay Mathematics Institute is offering a US$1 million reward to anyone who has a formal proof that P=NP or P$\neq$NP.

Choice of Algorithm

Observation 1: You cannot make an inefficient algorithm efficient by how you choose to implement it or what machine you choose to run it on.
Observation 2: It is virtually impossible to ruin the efficiency of an efficient algorithm by how you implement it or what machine you run it on.
So, the efficiency is determined by the algorithms and data structures used in your solution. Efficiency is not significantly affected by how well or how poorly you implement the code.

Fast Growing Functions

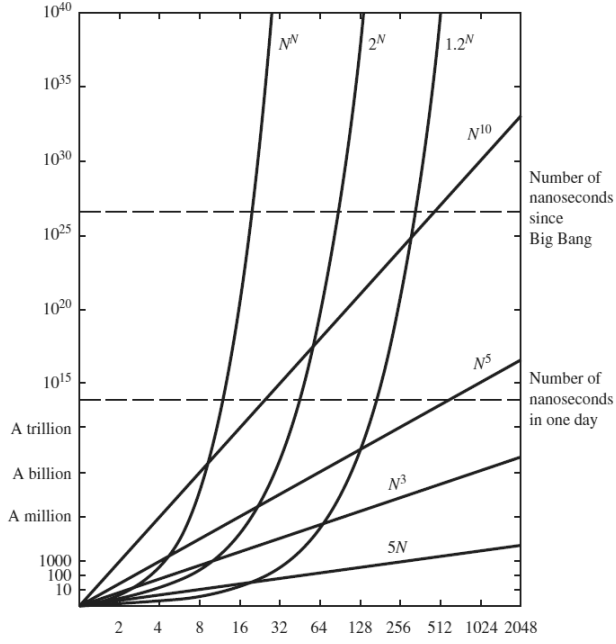
The order of an algorithm is generally more important than the speed of the processor.

Fast growing functions grow really fast. Their growth is stupefying. Don't be misled.

Goodrich and Tamassia, Table 3.2, page 120.

Figure 7.4

Growth rates of some functions.



Comparing Functions

In finding a name in phone book, suppose every comparison takes one millisecond (0.001 sec). How long does it take to find the name in the worse case?

city	pop	linear	binary
Port St. Lucie	164,603	2.8 min	0.017 sec
Fort Lauderdale	165,521	2.8 min	0.017 sec
Tallahassee	181,376	3.0 min	0.017 sec
Hialeah	224,669	3.7 min	0.018 sec
Orlando	238,300	4.0 min	0.018 sec
St. Petersburg	244,769	4.0 min	0.018 sec
Tampa	335,709	5.6 min	0.018 sec
Miami	399,457	6.7 min	0.019 sec
Jacksonville	821,784	13.7 min	0.020 sec

Comparing Functions

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec). How long does it take to find the name in the worse case?

city	pop	linear	binary
Dallas, TX	1,299,543	21.7 min	0.020 sec
San Diego, CA	1,306,301	21.8 min	0.020 sec
San Antonio, TX	1,373,668	22.9 min	0.020 sec
Philadelphia, PA	1,547,297	25.8 min	0.021 sec
Phoenix, AZ	1,601,587	26.7 min	0.021 sec
Houston, TX	2,257,926	37.6 min	0.021 sec
Chicago, IL	2,851,268	47.5 min	0.021 sec
Los Angeles, CA	3,831,868	63.9 min	0.022 sec
New York, NY	8,391,881	139.9 min	0.023 sec

Comparing Functions

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec). How long does it take to find the name in the worse case?

city	pop	linear	binary
Seoul	10,575,447	2.9 hr	0.023 sec
São Paulo	11,244,369	3.1 hr	0.023 sec
Moscow	11,551,930	3.2 hr	0.023 sec
Beijing	11,716,000	3.3 hr	0.023 sec
Mumbai	12,478,447	3.5 hr	0.024 sec
Delhi	12,565,901	3.5 hr	0.024 sec
Istanbul	12,946,730	3.6 hr	0.024 sec
Karachi	12,991,000	3.6 hr	0.024 sec
Shanghai	17,836,133	5.0 hr	0.024 sec

Fast Growing Functions

$\log n$	n	$n \log n$	n^2	n^3	2^n	
3	10	30	100	1,000	1,024	kilo
4	20	80	400	8,000	1,048,576	mega
4	30	120	900	27,000	1,073,741,824	giga
5	40	200	1,600	64,000	1,099,511,627,776	tera
5	50	250	2,500	125,000	1,125,899,906,842,624	peta
6	60	300	3,600	216,000	1.15×10^{18}	exa
6	70	420	4,900	343,000	1.18×10^{21}	zetta
6	80	480	6,400	512,000	1.21×10^{24}	yotta
6	90	540	8,100	729,000	1.24×10^{27}	
6	100	600	10,000	1,000,000	1.27×10^{30}	

Algorithms Have Changed the World

Performance is the key.

- QuickSort
- FFT
- Barnes-Hut

J. MacCormick, *Nine Algorithms That Changed The Future*. [I like the title, but not the list.]

What's the N?

```
while (STDIN.hasNext()) {
   final String token = STDIN.next();
}
```

The size N is the number of tokens in the input. So, the number of steps is $a(N+1)+bN$ where a is some arbitrary measure of the "cost" of executing `hasNext()` and b the "cost" of executing `next()`. Therefore, we say $O(N)$.

Two Independent Variables

```
for (int i=1; i<=N; i++) {
    x=x+1;
}
for (int j=1; j<=M; j++) {
    y=y+1;
}
```

Therefore, we say $O(N + M)$.

There are ways to reduce the number of independent variables. For example, ff $M \leq 10$, say, then $O(N + M) = O(N + 1) = O(N)$. If $M \approx N$, then $O(N + M) = O(N + N) = O(N)$.

Categorizing Programs

Compute $\sum_{i=1}^{n} i$

Algorithm 1 – $O(n)$

```
final int n = Integer.parseInt (args[0]);
int sum = 0;
for (int count=1; count<=n; counter++) {
   sum += count;
}
```

Algorithm 2 – $O(1)$

```
final int n = Integer.parseInt (args[0]);
int sum = (n*(n+1))/2;
```

Give a Big-Oh analysis in terms of N of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
     sum++;
}
```

Give a Big-Oh analysis in terms of N of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
    sum++;
}
```

$O(N)$

```
for (int i=1; i<N; i+=2) {
    sum++;
}
```

Give a Big-Oh analysis in terms of N of the running time for each of the following program fragments:

```
for (int i=1; i<N; i++) {
    sum++;
}
```

$O(N)$

```
for (int i=1; i<N; i+=2) {
    sum++;
}
```

$O(N/2) = O(N)$

```
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}
```

```
for (int i=1; i<N; i++) {
    for (int j=1; j<N; j++) {
        sum++;
    }
}
```

$O(N^2)$

```
for (int i=1; i<10; i++) {
    O(N) steps in loop
}
```

```
for (int i=1; i<10; i++) {
    O(N) steps in loop
}
```

$O(10N) = O(N)$

```
for (int i=1; i<10; i++) {
    O(N) steps in loop
}
```

```
for (int i=1; i<10; i++) {
    O(N) steps in loop
}
```

$O(10N) = O(N)$

```
if ( /**/ ) {
  for (int i=1; i<10; i++) {
    sum++
  }
} else {
  sum++
}
```

```
if ( /**/ ) {
  for (int i=1; i<10; i++) {
    sum++
  }
} else {
  sum++
}
```

O(1)

```
if ( /**/ ) {
  for (int i=1; i<N; i++) {
    sum++
  }
} else {
  sum++
}
```

```
if ( /**/ ) {
  for (int i=1; i<N; i++) {
    sum++
  }
} else {
  sum++
}
```

$$\max\left(O(N), O(1)\right) = O(N)$$

```
for (int i=1; i<=N; i++) {
   for (int j=1; j<=i j++) {
      sum++
   }
}
```

```
for (int i=1; i<=N; i++) {
   for (int j=1; j<=i j++) {
      sum++
   }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{i} 1 = \sum_{i=1}^{N} i == 1+2+3+\cdots+i = \frac{N\times(N+1)}{2} = \frac{N^2}{2}+\frac{N}{2} = O(N^2)$$

```
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N*N; j++) {
        for (int k=1; k<=j; k++) {
            sum++;
        }
    }
}
```

```
for (int i=1; i<=N; i++) {
    for (int j=1; j<=N*N; j++) {
        for (int k=1; k<=j; k++) {
            sum++;
        }
    }
}
```

$$\sum_{i=1}^{N}\sum_{j=1}^{N\times N}\sum_{k=1}^{j}1=\sum_{i=1}^{N}\sum_{j=1}^{N\times N}j=\sum_{i=1}^{N}\frac{N^2\times(N^2+1)}{2}=N\times\frac{N^2\times(N^2+1)}{2}$$

$$N\times\frac{N^2\times(N^2+1)}{2}=\frac{N^3\times(N^2+1)}{2}=\frac{N^5+N^3}{2}=O(N^5+N^3)=O(N^5)$$

```
for (int i=1; i<N; i*=2) {
    sum++;
}

while (N>1) {
    N = N/2;
    /* O(1) */
}
```

```
for (int i=1; i<N; i*=2) {
    sum++;
}

while (N>1) {
    N = N/2;
    /* O(1) */
}
```

$O(\log N)$

Categorizing Programs

Compute $\lceil \log n \rceil$

Algorithm 1 – $O(\log n)$

```
for (lgN=0; Math.pow(2,lgN)<n; lgN++);
```

Algorithm 2 – $O(\log n)$

```
for (lgN=0; n>0; lgN++, n/=2);
```

Algorithm 3 – $O(\log n)$

```
for (lgN=0,t=1; t<n; lgN++, t += t);
```

Some Recursive Patterns

```
public static void g (int N) {
  if (N==0) return;
  g (N/2); // half the amount work
}
```

Some Recursive Patterns

```
public static void g (int N) {
  if (N==0) return;
  g (N/2); // half the amount work
}
```

$O(\log N)$ as in binary search `RecursiveBinary.java` — tail recursive

— Bounded polymorphism (recursive) ??
`Binary.java` — iterative version
`GenericBinary.java` – Bounded polymorphism (iterative)
— Java program [???] to demonstrate working of
`Collections.binarySearch()` for `List<T>` with natural ordering

```java
public static void g (int N) {
  if (N==0) return;
  g (N/2);    // half the amount work
  g (N/2);    // not the same work
  /* O(N) */
}
```

```
public static void g (int N) {
  if (N==0) return;
  g (N/2);    // half the amount work
  g (N/2);    // not the same work
  /* O(N) */
}
```

$O(N \log N)$ as in merge sort. This pattern is associated with the divide-and-conquer strategy for problem solving. Merge.java

Quick Sort

Looks like the same pattern as merge sort, but it different. This is subtle and important in the study of sorting.

`Quick.java`

`GenericQuick.java`

Quick sort with Hungarian Folk Dancers

Select sort

```java
public static void f (int N) {
  if (N==0) return;
  f (N-1);
  f (N-1);
  /* O(1) */
}
```

```java
public static void f (int N) {
  if (N==0) return;
  f (N-1);
  f (N-1);
  /* O(1) */
}
```

$O(2^N)$

TowersOfHanoi.java Towers of Hanoi at Wiki

An example of a simple exponential problem: enumerating all the different kinds of pizzas with N possible pizza toppings.

Pizza.java

Problems

There is likely more than one algorithm to solve a problem.

Minimum Element in an Array. Given an array of N items, find the smallest item.
Closest Points in the Plane. Given N points in a plane, find the pair of points that are closest together.
Co-linear Points in the Plane. Given N points in a plane, determine if any three form a straight line.

Prefix Averages

Two algorithms to solve a simple problems.

`PrefixAverages.java` Java program

Maximum Contiguous Subsequence Sum

Maximum Contiguous Subsequence Sum Problem. Given (possibly negative) integers a_1, a_2, ..., a_n, find (and identify the sequence corresponding to) the maximum value of $\sum_{k=i}^{j} a_k$. The maximum contiguous subsequence sum is zero if all the integers are negative.

For example, if the input is $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, then the answer is 20 which corresponds to the contiguous subsequence encompassing elements 2 through 4.

Weiss, Section 5.3, page 153.

Maximum Contiguous Subsequence Sum

The obvious $O(n^3)$ algorithm: for every potential starting element of the subsequence, and for every potential ending element of the subsequence, find the one with the maximum sum.

Maximum Contiguous Subsequence Sum

Since $\sum_{k=i}^{j+1} a_k = (\sum_{k=i}^{j} a_k) + a_{j+1}$, the sum of the subsequence a_i, a_{i+1}, ..., a_{j+1} can be computed easily (without a loop) from the sum of a_i, a_{i+1}, ..., a_j.

Maximum Contiguous Subsequence Sum

Theorem. Let a_k for $i \leq k \leq j$ be any subsequence with $\sum_{k=i}^{j} a_k < 0$. If $q > j$, then a_k for $i \leq k \leq q$ is *not* a maximum contiguous subsequence.

Proof. The sum of the subsequence a_k for $j + 1 \leq k \leq q$ is larger.

Maximum Contiguous Subsequence Sum

`MaxSubsequenceSum.java` Java program

Dynamic Programming

See Sedgewick and Wayne.

Math Review

$$\log_b a = c \quad \text{if} \quad a = b^c$$

Nearly always we want the base to be 2.

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Lots of discrete steps

- $\lceil x \rceil$ the largest integer less than or equal to x.
- $\lfloor x \rfloor$ the smallest integer less than or equal to x.