Computer programming is the expression of computation as algorithm and data structures. This is mathematical problem solving except for:

- Finite bits versus numbers
- Performance

We wish to examine performance and we introduce Big-Oh notation which is used to categorize computer programs quite usefully.
“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, et al. *Data Structures and Algorithms in Java*
Something Related For Additional Reading

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.

There is a better way using functions.
Steps, Worse Case

The work a computer does can be measured in the number of individual instructions it executes. The work 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

In common with the empirical approach, we suppose it makes sense to talk about the size of the 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? Most 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, or some other abstract measure of the size on the data set being processed.

In the long run, little differences in the number of steps do not matter, so we group functions together in larger categories to more easily see significant difference.
Input Size?

Networks and graphs often take the “size” to be two numbers: the number of edges $E$ and the number of nodes $V$.

Different results are obtained if the size of input is measured differently. E.g., one integer of arbitrary magnitude, or $n$ bits representing an integer.

We will not concern ourselves here with such complications.
The number of steps a program 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]
We want a precise, i.e., mathematical way to compare functions. What kind of functions? Although the usual approach applies to functions \( f : \mathbb{R} \rightarrow \mathbb{R} \), 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.
Let $f(n)$ and $g(n)$ be functions mapping from natural numbers $\mathbb{N}$ (non-negative integers) to $\mathbb{N}$.

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

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 $f(n) = \lceil \log n \rceil$. 
Preliminaries

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 $O(g(n))$
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 $n \geq n_0$.

DONALD KNUTH

Author of "The Art of Computer Programming."
THE POTRZEbie SYSTEM

OF WEIGHTS AND MEASURES

THE POTRZEbie SYSTEM

The Potrzebie system is a new system of measuring, which is destined to become the measuring system of the future, based on improvements over the older systems now in use. It is based upon measurements taken 6-9-12 at the Physics Lab. of Milwaukee Luthorsen High School in Milwaukee, Wis., when the thickness of MAD Magazine was determined to be 2.2634451 - 7.4873216473 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 waves lengths of the red line in the spectrum of cesium. A partial table of the Potrzebie System, the measuring system of the future, is given below.

LENGTH

1 potrzebie = thickness of MAD Magazine
0.000001 = 1 farshimmelt potrzebie (fp)
1000 fp = 1 millopotrzebie (mp)
10 mp = 1 centipotrzebie (cp)
10 cp = 1 decipotrzebie (dp)
10 dp = 1 potrzebie (p)
100 p = 1 desapotrzebie (d)
100 dp = 1 hectaropotrzebie (h)
1000 h = 1 kilopotrzebie (kh)
1000 kh = 1 furshugginer potrzebie (fpf)

VOLUME

1 cubic decapotrzebie = 1 ngam (n)
0.000001 ngam = 1 farshimmelt ngam (fn)
1000 fn = 1 millangam (mm)
10 mm = 1 centangam (cm)
10 cm = 1 decangam (dm)
10 dm = 1 ngam (n)
10 n = 1 decangam (dn)
10 dm = 1 hectarangam (hm)
10 h = 1 kilangam (kh)
1000 kh = 1 furshugginer ngam (fn)

MASS

1 ngam of halavoh* = 1 biun (bl)
0.000001 biun = 1 farshimmelt biun (fb)
1000 fb = 1 millibiu (bb)
10 bb = 1 centibiu (cm)
10 cm = 1 decibiu (dm)
10 dm = 1 biun (bl)
10 bl = 1 decibiu (db)
10 db = 1 hectaribiu (hb)
10 hb = 1 kilobiu (kh)
1000 kh = 1 furshugginer biun (fb)

*Halavoh is a form of pie, and it has a specific gravity of 3.1416 and a specific heat of 3.1416.
Donald E. Knuth (1938–)

Introduction to Knuth’s organ composition ✝ YouTube [17 minutes]
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.

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


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.


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 rapidly apprehends the method.
Biographies appear in:

- Slater, *Portraits in Silicon*, 1987
Categorizing functions \( f \text{ in } g \)

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 \( n \geq n_0 \).
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 \( n \geq n_0 \).
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.
\[ f(n) \text{ is } O(g(n)) \approx x \leq y \]
\[ f(n) \text{ is } \Theta(g(n)) \approx x = y \]
\[ f(n) \text{ is } \Omega(g(n)) \approx x \geq y \]
\[ f(n) \text{ is } o(g(n)) \approx x < y \]
\[ f(n) \text{ is } \omega(g(n)) \approx x > y \]

The analogy is rough since some functions are not comparable, while any two real numbers are comparable.
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) \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = L \) where \( 0 < L < \infty \).

Lemma. \( f(n) \) is \( O(g(n)) \) if, and only, if \( \limsup_{n \to \infty} \frac{f(n)}{g(n)} = L \) where \( 0 < L < \infty \).
Relationships

\[ O(g(n)) \]
\[ \Theta(g(n)) \]
\[ \Omega(g(n)) \]
\[ o(g(n)) \]
\[ \omega(g(n)) \]
Here $L$ denotes the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$
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 $n \geq 17$. 
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 $O(n)$?
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 \( O(g(n)) \).

Some authors write \( f(n) \in O(g(n)) \), or even \( f(n) = O(g(n)) \), but I find that misleading.
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 \( O(n^2) \).
Lemma: \( n^d \) is in \( O(n^{d+1}) \)
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
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.
Comparing Functions

In finding a name in phone book, suppose every comparison takes one millisecond (0.001 sec).

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

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec).

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

In finding a name in phone book, suppose every comparison takes one microsecond (0.001 sec).

<table>
<thead>
<tr>
<th>city</th>
<th>pop</th>
<th>linear</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seoul</td>
<td>10,575,447</td>
<td>2.9 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>São Paulo</td>
<td>11,244,369</td>
<td>3.1 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Moscow</td>
<td>11,551,930</td>
<td>3.2 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Beijing</td>
<td>11,716,000</td>
<td>3.3 hr</td>
<td>0.023 sec</td>
</tr>
<tr>
<td>Mumbai</td>
<td>12,478,447</td>
<td>3.5 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Delhi</td>
<td>12,565,901</td>
<td>3.5 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Istanbul</td>
<td>12,946,730</td>
<td>3.6 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Karachi</td>
<td>12,991,000</td>
<td>3.6 hr</td>
<td>0.024 sec</td>
</tr>
<tr>
<td>Shanghai</td>
<td>17,836,133</td>
<td>5.0 hr</td>
<td>0.024 sec</td>
</tr>
</tbody>
</table>
## Fast Growing Functions

<table>
<thead>
<tr>
<th>log $n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>30</td>
<td>100</td>
<td>1,000</td>
<td>1,024</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>80</td>
<td>400</td>
<td>8,000</td>
<td>1,048,576</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>120</td>
<td>900</td>
<td>27,000</td>
<td>1,073,741,824</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>200</td>
<td>1,600</td>
<td>64,000</td>
<td>1,099,511,627,776</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>250</td>
<td>2,500</td>
<td>125,000</td>
<td>1,125,899,906,842,624</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>300</td>
<td>3,600</td>
<td>216,000</td>
<td>$1.15 \times 10^{18}$</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>420</td>
<td>4,900</td>
<td>343,000</td>
<td>$1.18 \times 10^{21}$</td>
</tr>
<tr>
<td>6</td>
<td>80</td>
<td>480</td>
<td>6,400</td>
<td>512,000</td>
<td>$1.21 \times 10^{24}$</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>540</td>
<td>8,100</td>
<td>729,000</td>
<td>$1.24 \times 10^{27}$</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>600</td>
<td>10,000</td>
<td>1,000,000</td>
<td>$1.27 \times 10^{30}$</td>
</tr>
</tbody>
</table>
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?

```java
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

```c
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, if $M \leq 10$, 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 – $O(n)$

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

Algorithm 2 – $O(1)$

```java
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:

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

$O(N)$

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

$O(N/2) = O(N)$
Give a Big-Oh analysis in terms of $N$ of the running time for each of the following program fragments:

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

```c
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:

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

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

\(O(10) = O(N)\)
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 (O(N), O(1)) = O(N)
for (int i=1; i<=N; i++) {
    for (int j=1; j<=i j++) {
        sum++
    }
}

\[
\sum_{i=1}^{N} \sum_{j=1}^{i} 1 = \sum_{i=1}^{N} i = 1 + 2 + 3 + \cdots + i = \frac{N(N+1)}{2} = \frac{N^2}{2} + \frac{N}{2} = O(N^2)
\]
for (int i=1; i<=N; i++) {
    for (int j=1; j<=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 ++;
        }
    }
}

\[
\sum_{i=1}^{N} \sum_{j=1}^{N\times N} \sum_{k=1}^{j} 1 = N \times N \times 1 = O(N^3)
\]
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 – $O(\log n)$

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

Algorithm 2 – $O(\log n)$

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

Algorithm 3 – $O(\log n)$

```java
for (lgN=0, t=1; t < n; lgN++, t += t);
```
public static void g(int N) {
    if (N==0) return;
    g(N/2); // half the amount work
}
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

Binary.java — iterative version

GenericBinary.java — (Bounded polymorphism)
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
public static void f (int N) {
    if (N==0) return;
    f (N-1);
    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 Problem. Given (possibly negative) integers $a_1, a_2, \ldots, a_n$, 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 $\{-2, 11, -4, 13, -5, 2\}$, then the answer is 20 which corresponds to the contiguous subsequence encompassing elements 2 through 4.

Weiss, Section 5.3, page 153.
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 $a_i, a_{i+1}, \ldots, a_{j+1}$ can be computed easily (without a loop) from the sum of $a_i, a_{i+1}, \ldots, a_j$. 
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

- \([x]\) the largest integer less than or equal to \(x\).
- \([x]\) the smallest integer less than or equal to \(x\).