

- Web Resources
- [FAQ](#)
- [Data](#)
- [Code](#)
- [Errata](#)
- [Lectures](#)
- [Cheatsheet](#)
- [References](#)
- [Online Course](#)
- [Programming Assignments](#)



1.4 Analysis of Algorithms

As people gain experience using computers, they use them to solve difficult problems or to process large amounts of data and are invariably led to questions like these:

- *How long will my program take?*
- *Why does my program run out of memory?*

Scientific method.

The very same approach that scientists use to understand the natural world is effective for studying the running time of programs:

- *Observe* some feature of the natural world, generally with precise measurements. 
- *Hypothesize* a model that is consistent with the observations.
- *Predict* events using the hypothesis.
- *Verify* the predictions by making further observations.
- *Validate* by repeating until the hypothesis and observations agree. 

The experiments we design must be *reproducible* and the hypotheses that we formulate must be *falsifiable*.

Observations.

Our first challenge is to determine how to make quantitative measurements of the running time of our programs. [Stopwatch.java](#) is a data type that measures the elapsed running time of a program.



定量

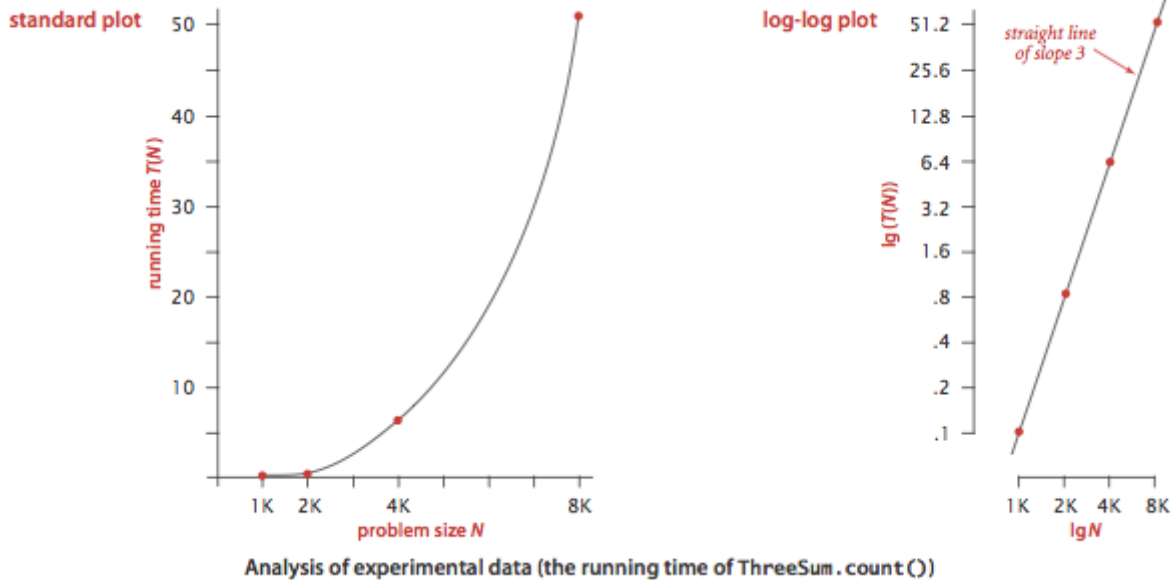
```
public class Stopwatch
```

```
    Stopwatch()      create a stopwatch
```

```
    double elapsedTime()  return elapsed time since creation
```

for $i=0, i < n, i++$
 for $j=i+1, j < n, j++$
 for $k=j+1, k < n, k++$

[ThreeSum.java](#) counts the number of triples in a file of N integers that sums to 0 (ignoring integer overflow).
[DoublingTest.java](#) generates a sequence of random input arrays, doubling the array size at each step, and prints the running times of `ThreeSum.count()` for each input size. [DoublingRatio.java](#) is similar but also output the ratios in running times from one size to the next.



Mathematical models.

The total running time of a program is determined by two primary factors: the cost of executing each statement and the frequency of execution of each statement.

- Tilde approximations. We use tilde approximations, where we throw away low-order terms that complicate formulas. We write $\sim f(N)$ to represent any function that when divided by $f(N)$ approaches 1 as N grows. We write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as N grows.

function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

\sim 近似.

- Order-of-growth classifications. Most often, we work with tilde approximations of the form $g(N) \sim a f(N)$ where $f(N) = N^b \log^c N$ and refer to $f(N)$ as the order of growth of $g(N)$. We use just a few structural primitives (statements, conditionals, loops, nesting, and method calls) to implement algorithms, so very often the order of growth of the cost is one of just a few functions of the problem size N .

description	order of growth	typical code framework	description	example
<i>constant</i>	1	<code>a = b + c;</code>	<i>statement</i>	<i>add two numbers</i>
<i>logarithmic</i>	$\log N$	[see page 47]	<i>divide in half</i>	<i>binary search</i>
<i>linear</i>	N	<pre>double max = a[0]; for (int i = 1; i < N; i++) if (a[i] > max) max = a[i];</pre>	<i>loop</i>	<i>find the maximum</i>
✓ <i>linearithmic</i>	$N \log N$	[see ALGORITHM 2.4]	<i>divide and conquer</i>	<i>mergesort</i>
<i>quadratic</i>	N^2	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) if (a[i] + a[j] == 0) cnt++;</pre>	<i>double loop</i>	<i>check all pairs</i>
<i>cubic</i>	N^3	<pre>for (int i = 0; i < N; i++) for (int j = i+1; j < N; j++) for (int k = j+1; k < N; k++) if (a[i] + a[j] + a[k] == 0) cnt++;</pre>	<i>triple loop</i>	<i>check all triples</i>
<i>exponential</i>	2^N	[see CHAPTER 6]	<i>exhaustive search</i>	<i>check all subsets</i>

- *Cost model.* We focus attention on properties of algorithms by articulating a *cost model* that defines the basic operations. For example, an appropriate cost model for the 3-sum problem is the number of times we access an array entry, for read or write.

Property. The order of growth of the running time of [ThreeSum.java](#) is N^3 .

Proposition. The brute-force 3-sum algorithm uses $\sim N^3 / 2$ array accesses to compute the number of triples that sum to 0 among N numbers.

Designing faster algorithms.

One of the primary reasons to study the order of growth of a program is to help design a faster algorithm to solve the same problem. Using mergesort and binary search, we develop faster algorithms for the 2-sum and 3-sum problems.

- 2-sum. The brute-force solution [TwoSum.java](#) takes time proportional to N^2 . [TwoSumFast.java](#) solves the 2-sum problem in time proportional to $N \log N$ time.
- 3-sum. [ThreeSumFast.java](#) solves the 3-sum problem in time proportional to $N^2 \log N$ time.

Coping with dependence on inputs.

For many problems, the running time can vary widely depending on the input.

- *Input models.* We can carefully model the kind of input to be processed. This approach is challenging because the model may be unrealistic.
- *Worst-case performance guarantees.* Running time of a program is less than a certain bound (as a function of the input size), no matter what the input. Such a conservative approach might be appropriate for the software that runs a nuclear reactor or a pacemaker or the brakes in your car.
- *Randomized algorithms.* One way to provide a performance guarantee is to introduce randomness, e.g., quicksort and hashing. Every time you run the algorithm, it will take a different amount of time. These guarantees are not absolute, but the chance that they are invalid is less than the chance your computer will be struck by lightning. Thus, such guarantees are as useful in practice as worst-case guarantees.
- *Amortized analysis.* For many applications, the algorithm input might be not just data, but the sequence of operations performed by the client. Amortized analysis provides a worst-case performance guarantee on a *sequence* of operations.

Proposition. In the linked-list implementation of Bag, Stack, and Queue, all operations take constant time in the worst case.

Proposition. In the resizing-array implementation of Bag, Stack, and Queue, starting from an empty data structure, any sequence of N operations takes time proportional to N in the worst case (amortized constant time per operation).

Memory usage.

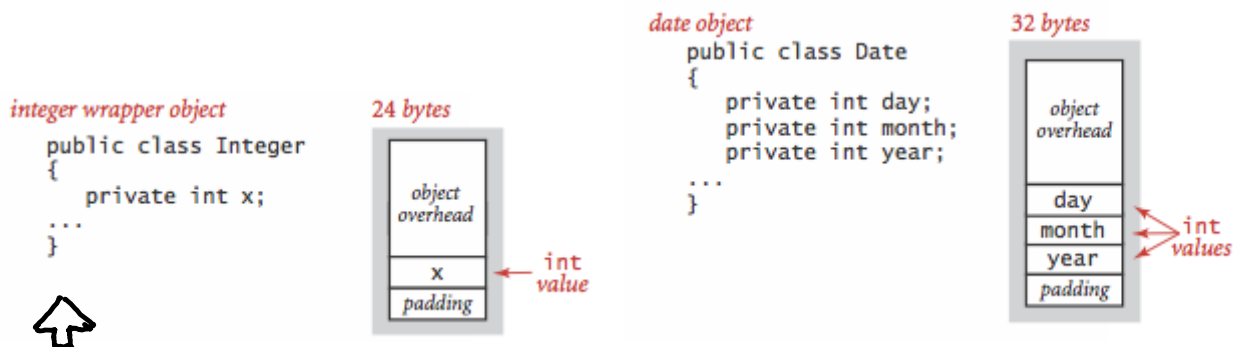
To estimate how much memory our program uses, we can count up the number of variables and weight them by the number of bytes according to their type. For a *typical* 64-bit machine,

- *Primitive types.* the following table gives the memory requirements for primitive types.

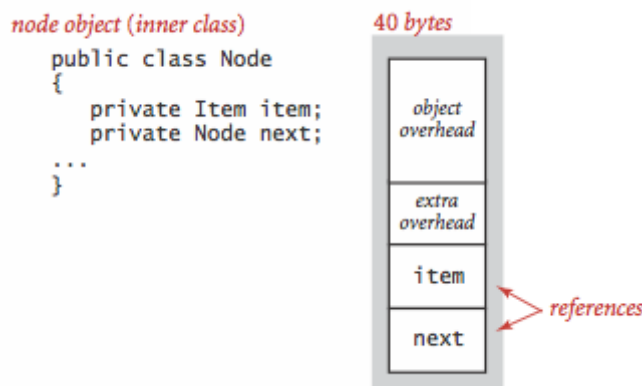
type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

Typical memory
requirements for
primitive types

- *Objects.* To determine the memory usage of an object, we add the amount of memory used by each instance variable to the overhead associated with each object, typically 16 bytes. Moreover, the memory usage is typically padded to be a multiple of 8 bytes (on a 64-bit machine).



- References. A reference to an object typically is a memory address and thus uses 8 bytes of memory (on a 64-bit machine).
- Linked lists. A nested non-static (inner) class such as our Node class requires an extra 8 bytes of overhead (for a reference to the enclosing instance).

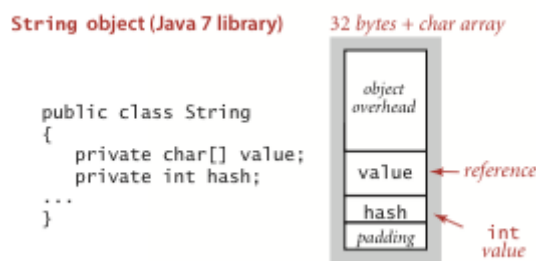


- Arrays. Arrays in Java are implemented as objects, typically with extra overhead for the length. An array of primitive-type values typically requires 24 bytes of header information (16 bytes of object overhead, 4 bytes for the length, and 4 bytes of padding) plus the memory needed to store the values.

type	bytes
int[]	~4N
double[]	~8N
Date[]	~40N
double[][]	~8NM

N: length.

- Strings. A Java 7 string of length N typically uses 32 bytes (for the String object) plus $24 + 2N$ bytes (for the array that contains the characters) for a total of $56 + 2N$ bytes.



Depending on context, we may or may not count the memory references by an object (recursively). For example, we count the memory for the `char[]` array in the memory for a `String` object because this memory is allocated when the string is created. But, we would not ordinarily count the memory for the `String` objects in a `StackOfStrings` object because the `String` objects are created by the client.

Q + A

Q. How do I increase the amount of memory and stack space that Java allocates?

A. You can increase the amount of memory allotted to Java by executing with `java -Xmx200m Hello` where 200m means 200 megabytes. The default setting is typically 64MB. You can increase the amount of stack space allotted to Java by executing with `java -Xss200k Hello` where 200k means 200 kilobytes. The default setting is typically 128KB. It's possible to increase both the amount of memory and stack space by executing with `java -Xmx200m -Xss200k Hello`.

Q. What is the purpose of padding? 填充

A. Padding makes all objects take space that is a multiple of 8 bytes. This can waste some memory but it speeds up memory access and garbage collection.

Q. I get inconsistent timing information in my computational experiments. Any advice?

A. Be sure that your computation is consuming enough CPU cycles so that you can measure it accurately. Generally, 1 second to 1 minute is reasonable. If you are using huge amounts of memory, that could be the bottleneck. Consider turning off the HotSpot compiler, using `java -Xint`, to ensure a more uniform testing environment. The downside is that you are no longer measuring exactly what you want to measure, i.e., actual running time.

Q. Does the linked-list implementation of a stack or queue really guarantee constant time per operation if we take into account garbage collection and other runtime processes?

A. Our analysis does not account for many system effects (such as caching, garbage collection, and just-in-time compilation)—in practice, such effects are important. In particular, the default Java garbage collector achieves only a constant amortized time per operation guarantee. However, there are *real-time* garbage collectors that guarantee constant time per operation in the worst case. [Real time Java](#) provides extensions to Java that provide worst-case performance guarantees for various runtime processes (such as garbage collection, class loading, Just-in-time compilation, and thread scheduling).

Exercises

6. Give the order of growth (as a function of N) of the running times of each of the following code fragments:

1.

```
int sum = 0;
for (int n = N; n > 0; n /= 2)
    for (int i = 0; i < n; i++)
        sum++;
```

2.

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
    for (int j = 0; j < i; j++)
        sum++;
```

3.

```
int sum = 0;
for (int i = 1; i < N; i *= 2)
    for (int j = 0; j < N; j++)
        sum++;
```

Answer: linear ($N + N/2 + N/4 + \dots$); linear ($1 + 2 + 4 + 8 + \dots$); linearithmic (the outer loop loops $\lg N$ times).

Creative Problems