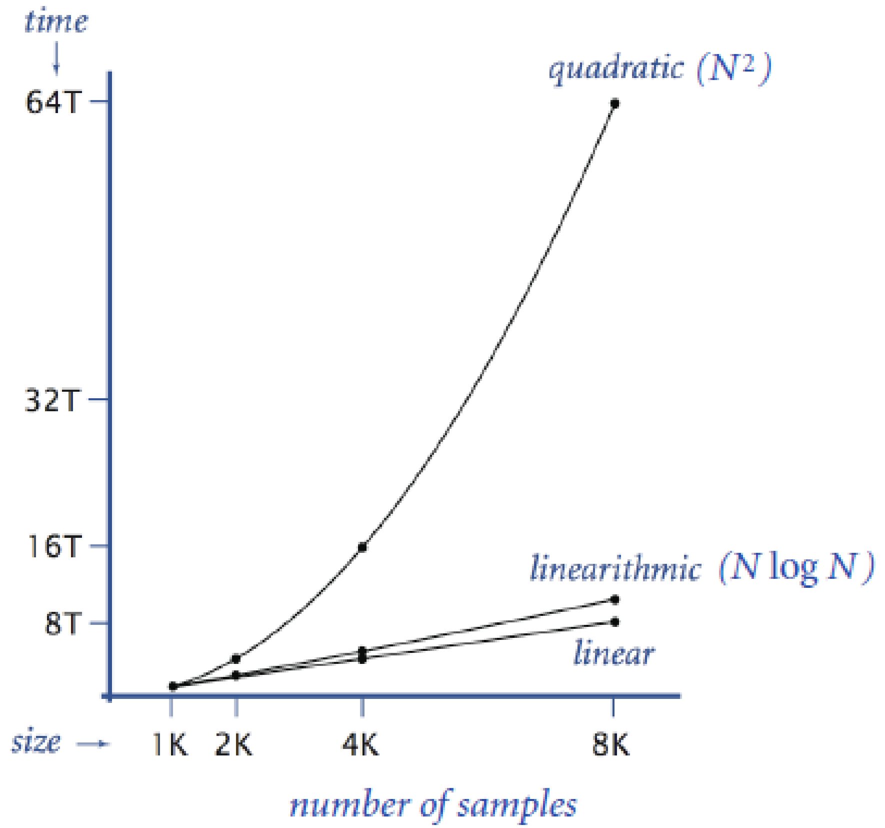
Performance

http://www.flickr.com/photos/exoticcarlife/3270764550/

http://www.flickr.com/photos/roel1943/5436844655/

Overview

- Performance analysis
  - Why we care
  - What we measure and how
  - How functions grow
- Empirical analysis
  - The doubling hypothesis
  - Order of growth



The Challenge

Q: Will my program be able to solve a large practical problem?

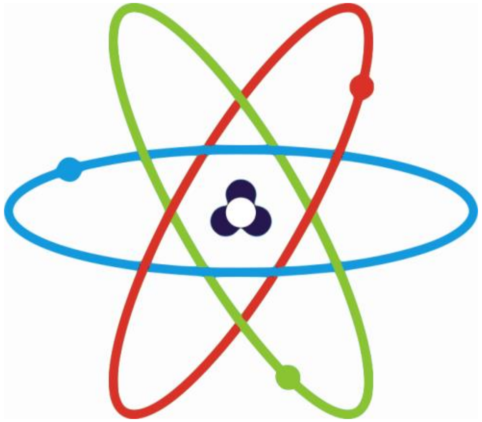
compile debug solve problems in practice

Key insight. [Knuth 1970s]

Use the scientific method to understand performance.

Scientific Method

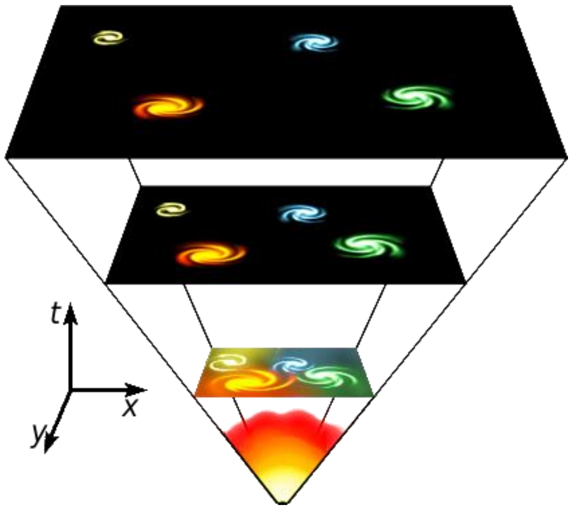
- Scientific method
  - Observe some feature of the natural world
  - Hypothesize a model that is consistent with the observations
  - Predict events using the hypothesis
  - Verify the predictions by making further observations
  - Validate by repeating until the hypothesis and observations agree



- Principles
  - Experiments must be reproducible
  - Hypothesis must be falsifiable

Why performance analysis

- Predicting performance
  - *When* will my program finish?
  - *Will* my program finish?

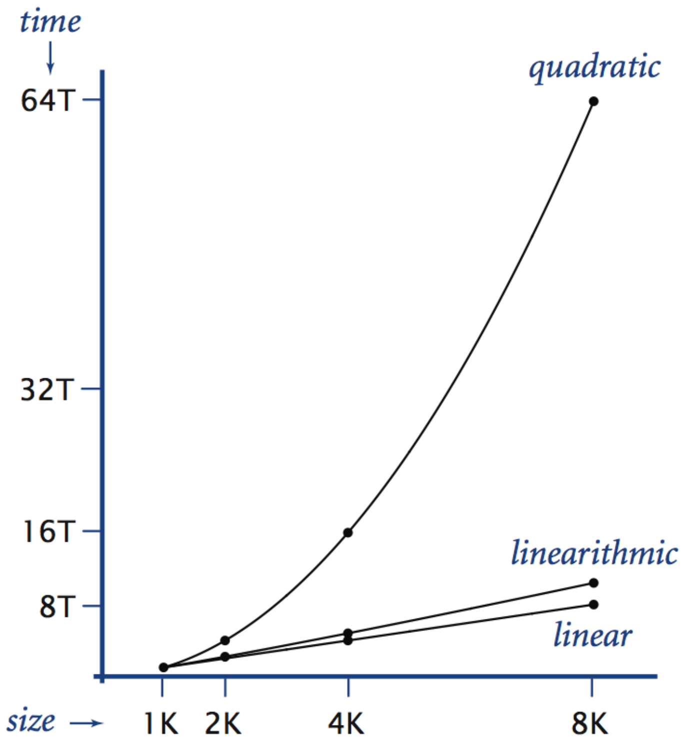


- Compare algorithms
  - Should I change to a more complicated algorithm?
  - Will it be worth the trouble?
- Basis for inventing new ways to solve problems
  - Enables new technology
  - Enables new research

Algorithmic successes

John von Neumann (1945)

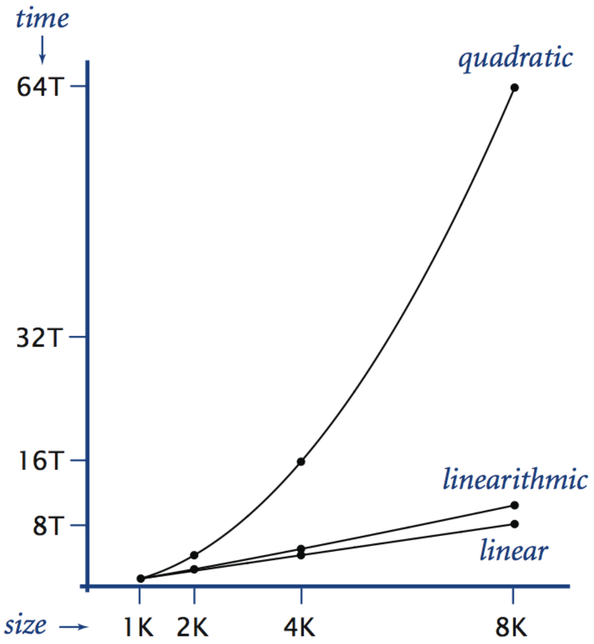
- Sorting
  - Rearrange array of N item in ascending order
  - Applications: databases, scheduling, statistics, genomics, ...
  - Brute force: N^2 steps
  - Mergesort: $N \log N$ steps, enables new technology



Algorithmic successes

Freidrich Gauss
(1805)

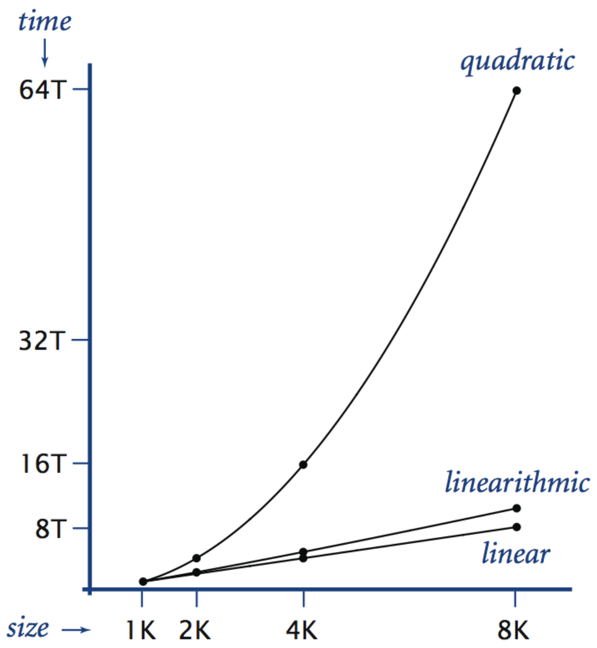
- Discrete Fourier transform
  - Break down waveform of N samples into periodic components
  - Applications: DVD, JPEG, MRI, astrophysics, ...
  - Brute force: N^2 steps
  - FFT algorithm: $N \log N$ steps, enables new technology



Algorithmic successes

Andrew Appel
PU '81

- N-body Simulation
  - Simulate gravitational interactions among N bodies
  - Application: cosmology, semiconductors, fluid dynamics, …
  - Brute force: N^2 steps
  - Barnes-Hut algorithm: $N \log N$ steps, enables new research



Performance metrics

- What do we care about?
  - Time, how long do I have to wait?
    - Measure with a stop watch (real or virtual)
    - Run in a performance profiler
      - Often part of an IDE (e.g. Microsoft Visual Studio)
      - Sometimes standalone (e.g. gprof)
      - Helps you determine bottleneck in your code

```
long   start       = System.currentTimeMillis();
// Do the stuff you want to time
long   now         = System.currentTimeMillis();
double elapsedSecs = (now - start) / 1000.0;
```

Measuring how long some code takes.

Performance metrics

- What do we care about?
  - Space, do I have the resources to solve it?
    - Usually we care about physical memory
      - 8 GB = 8.6 billion places to store a byte (byte = 256 possibilities)
      - Java double, 64-bits = 8 bytes
      - 8 GB / 8 bytes = over 1 million doubles!
    - Can swap to disk for some extra space
      - But much much slower

`Stats.java` class provides measurement of time and memory usage.

A "simple" problem

- Three-sum problem
  - Given N integers, find all triples that sum to 0

```
% more 8ints.txt
8
30 -30 -20 -10 40 0 10 5

% java ThreeSum < 8ints.txt
30 -30 0
30 -20 -10
-30 -10 40
-10 0 10
```

Brute force algorithm:
Try all possible triples and see if they sum to 0.

Three sums: brute-force

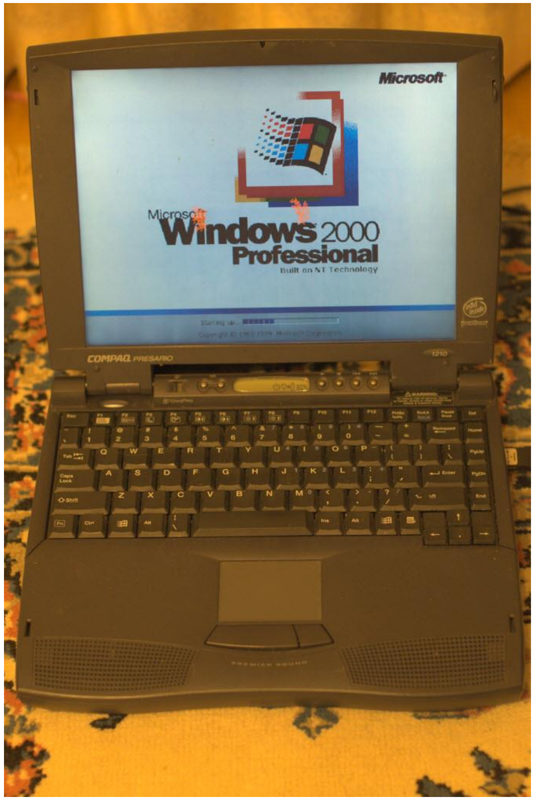
```
public class ThreeSum
{
   public static void main(String [] args)
   {
      int N = StdIn.readInt();
      int [] nums = new int[N];
      for (int i = 0; i < N; i++)
         nums[i] = StdIn.readInt();

      for (int i = 0; i < N; i++)
         for (int j = i + 1; j < N; j++)
            for (int k = j + 1; k < N; k++)
               if (nums[i] + nums[j] + nums[k] == 0)
                  System.out.println(nums[i] + " " +
                                     nums[j] + " " +
                                     nums[k]);
   }
}
```

All possible triples i < j < k from the set of integers.

Empirical analysis: three sum

- Run program for various input sizes, 2 machines:
  - My first laptop: Pentium 1, 150Mhz, 80MB RAM
  - My desktop: Phenom II, 3.2Ghz (3.6Ghz turbo), 32GB RAM

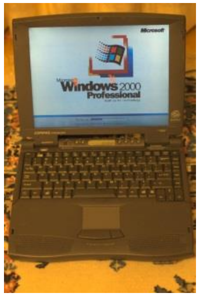


VS.

Empirical analysis: three sum

- Run program for various input sizes, 2 machines:
  - My first laptop: Pentium 1, 150Mhz, 80MB RAM
  - My desktop: Phenom II, 3.2Ghz (3.6Ghz turbo), 32GB RAM

N	ancient laptop	modern desktop
100	0.33	0.01
200	2.04	0.04
400	11.23	0.16
800	94.96	0.63
1600	734.03	4.33
3200	5815.30	33.69
6400	47311.43	263.82



Doubling hypothesis

- Cheap and cheerful analysis
  - Time program for input size N
  - Time program for input size 2N
  - Time program for input size 4N
  - …
  - Ratio T(2N) / T(N) approaches a constant
  - Constant tells you the exponent in $T = aN^b$

N	T(N)	ratio
400	0.16	-
800	0.63	3.94
1600	4.33	6.87
3200	33.69	7.78
6400	263.82	7.83

Desktop data

Constant from ratio	Hypothesis	Order of growth
2	$T = a N$	linear, $O(N)$
4	$T = a N^2$	quadratic, $O(N^2)$
8	$T = a N^3$	cubic, $O(N^3)$
16	$T = a N^4$	$O(N^4)$

Estimating constant, making predictions

N	T(N)	ratio
400	0.16	-
800	0.63	3.94
1600	4.33	6.87
3200	33.69	7.78
6400	263.82	7.83

Desktop data

$$T = a\,N^3$$

$$263.82 = a\,(6400)^3$$
$$a = 1.01 \times 10^{-09}$$

N	T(N)	ratio
400	11.23	-
800	94.96	8.45
1600	734.03	7.72
3200	5815.30	7.92
6400	47311.43	8.14

Laptop data

$$T = a\,N^3$$

$$47311.43 = a\,(6400)^3$$
$$a=1.80 \times 10^{-07}$$

How long for desktop to solve a 100,000 integer problem?

$1.01 \times 10^{-09}\,(100000)^3 = 1006393$ secs
$= 280$ hours

How long for laptop to solve a 100,000 integer problem?

$1.80 \times 10^{-07}\,(100000)^3 = 1.80 \times 10^{08}$ secs
$= 50133$ hours

Bottom line

- My three sum algorithm sucks
  - Does not scale to large problems → an algorithm problem
  - 15 years of computer progress didn't help much
  - My algorithm: $O(N^3)$
  - A slightly more complicated algorithm: $O(N^2 \log N)$

Using the better algorithm, how long does it take the desktop to solve a 100,000 integer problem?

$1.01 \times 10^{-09} (100000)^2 \log(100000) = 168$ secs

Using the better algorithm, how long does it take the ancient laptop to solve a 100,000 integer problem?

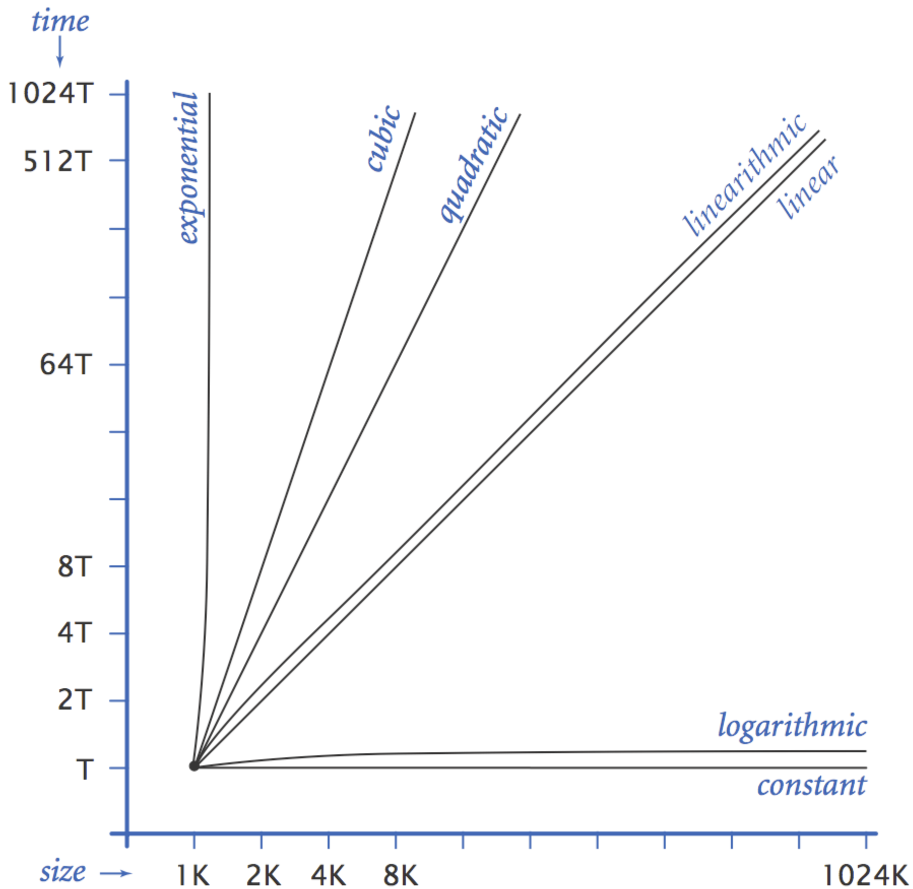
$1.80 \times 10^{-07} (100000)^2 \log(100000) = 29897$ secs

This assumes the same constant. Really should do the doubling experiment again with the new algorithm.

Constant in the time equation

- What influences the constant a?
  - e.g. $T = a N^2$
  - Speed of computer (CPU, memory, cache, …)
  - Implementation of algorithm
    - Body inside the nested for-loops may use more or less instructions
  - Software
    - Operating system
    - Compiler
    - Garbage collector
  - System
    - Other applications
    - Network (e.g. Windows update)

Order of growth



Orders of growth (log-log plot)

Doubling hypothesis ratio	Hypothesis	Order of growth
1	T = a	constant
1	T = a log N	logarithmic
2	T = a N	linear
2	T = a N log N	linearithmic
4	T = a N^2	quadratic
8	T = a N^3	cubic
2^N	T = a 2^N	exponential

Order of Growth: Consequences

order of growth	*predicted running time if problem size is increased by a factor of 100*
linear	a few minutes
linearithmic	a few minutes
quadratic	several hours
cubic	a few weeks
exponential	forever

Effect of increasing problem size for a program that runs for a few seconds

order of growth	*predicted factor of problem size increase if computer speed is increased by a factor of 10*
linear	10
linearithmic	10
quadratic	3-4
cubic	2-3
exponential	1

Effect of increasing computer speed on problem size that can be solved in a fixed amount of time

Order of growth

A small number of functions describe the running time of many fundamental algorithms!

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

log N

```
for (int i = 0; i < N; i++)
   ...
```

N

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      ...
```

N^2

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         ...
```

N^3

```
public static void g(int N)
{
   if (N == 0) return;
   g(N / 2);
   g(N / 2);
   for (int i = 0; i < N; i++)
      ...
}
```

N log N

```
public static void f(int N)
{
   if (N == 0) return;
   f(N - 1);
   f(N - 1);
   ...
}
```

2^N

Growth of nested loops

- Nested loops
  - A good clue to order of growth
  - But each loop must execute "on the order of" N times
  - If loop not a linear function of N, loop doesn't cause order to grow

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         count++;
```

N^3

N	T(N)	ratio
5000	6.85	-
10000	53.48	7.8
20000	425.97	8.0

$425.97 = a\ (20000^3)$

$a = 1.06 \times 10^{-6}$

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < 10000; k++)
         count++;
```

N^2

N	T(N)	ratio
5000	13.40	-
10000	53.20	3.97
20000	212.49	3.99

$212.49 = a\ (20000^2)$

$a = 5.31 \times 10^{-7}$

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         count++;
```

N^3

N	T(N)	ratio
5000	6.85	-
10000	53.48	7.8
20000	425.97	8.0

$425.97 = a\ (20000^3)$

$a = 1.06 \times 10^{-6}$

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < 10000; k++)
         count++;
```

N^2

N	T(N)	ratio
5000	13.40	-
10000	53.20	3.97
20000	212.49	3.99

$212.49 = a\ (20000^2)$

$a = 5.31 \times 10^{-7}$

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N / 5; k++)
         count++;
```

N^3

N	T(N)	ratio
5000	1.59	-
10000	11.08	6.97
20000	86.36	7.79

$86.36 = a\ (20000^3)$

$a = 2.16 \times 10^{-7}$

```
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < 10; k++)
         count++;
```

N^2

N	T(N)	ratio
5000	0.11	-
10000	0.37	3.36
20000	1.47	3.97

$1.47 = a\ (20000^2)$

$a = 3.68 \times 10^{-9}$

String processing example

- Goal: Strip all numbers 0-9 from a `String`
  - Go one char at a time, dropping any that are 0-9

```
private String stripNums(String text)
{
   String result = "";
   for (int i = 0; i < text.length(); i++)
   {
      char ch = text.charAt(i);
      if ((ch < '0') || (ch > '9'))
         result += ch;
   }
   return result;
}
```

As a function of the length of the string text, what order of growth is this method?

String processing, doubling hypothesis

- Read file with String of different lengths (N)
- Time how long it takes to run `stripNums()`

N	T(N)	ratio
8k	0.056	-
16k	0.150	2.7
32k	0.520	3.5
64k	1.932	3.7
128k	8.104	4.2
256k	36.267	4.5
512k	180.275	5.0

Order of growth:
Looks like N^2

WTH?

Trouble in String city

- Problem: `String` objects in Java are immutable
  - Once created, they can't be changed in any way
  - Java has to create a new object, copy the text into it
  - The old string gets garbage collected (eventually)

```java
private String stripNums(String text)
{
   String result = "";
   for (int i = 0; i < text.length(); i++)
   {
      char ch = text.charAt(i);
      if ((ch < '0') || (ch > '9'))
         result += ch;
   }
   return result;
}
```

This line is a hidden for-loop that copies all characters in the current result string into the newly created one.

A better stripping method

- Solution: Use a `StringBuilder` object
  - Can efficiently append characters to a string
  - Convert to a normal `String` once the loop is done

```
private static String stripNumsFast(String text)
{
   StringBuilder result = new StringBuilder();
   for (int i = 0; i < text.length(); i++)
   {
      char ch = text.charAt(i);
      if ((ch < '0') || (ch > '9'))
         result.append(ch);
   }
   return result.toString();
}
```

Need to call a method to append instead of the + operator.

Convert the contents of the buffer object to a normal Java `String`.

String processing performance

N	T(N)	ratio
8k	0.056	-
16k	0.150	2.7
32k	0.520	3.5
64k	1.932	3.7
128k	8.104	4.2
256k	36.267	4.5
512k	180.275	5.0

Original `stripNums()` appending to a `String` object. **Order of growth: N^2**

N	T(N)	ratio
8k	0.0000	-
16k	0.0100	-
32k	0.0000	-
64k	0.0100	-
128k	0.0100	-
256k	0.0100	-
512k	0.0100	-
1024k	0.0100	-
2048k	0.0200	2.0
4096k	0.0500	2.5
8192k	0.1100	2.2

New `stripNumsFast()` appending to a `StringBuffer` object.
Order of growth: N

Summary

- Introduction to Analysis of Algorithms
  - Today: simple empirical estimation
  - Next year: an entire semester course
- The algorithm matters
  - Faster computer only buys you out of trouble temporarily
  - Better algorithms enable new technology!
- The data structure matters
  - `String` vs. `StringBuilder`
- Doubling hypothesis
  - Measure time ratio as you double the input size
  - If the ratio = 2^b, runtime of algorithm $T(N) = a N^b$