1.4 Analysis of Algorithms

- Introduction
- Observations
- Mathematical models
- Order-of-growth classifications
- Memory

Cast of characters

Programmer needs to develop a working solution.

Client wants to solve problem efficiently.

Student (you) might play any or all of these roles someday.

Theoretician seeks to understand.

Running time

"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the shortest time?" — Charles Babbage (1864)
Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

Primary practical reason: avoid performance bugs.

An algorithmic success story

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

Discrete Fourier transform.
- Express signal as weighted sum of sines and cosines.
- Applications: DVD, JPEG, MRI, astrophysics, ....
- Brute force: $N^2$ steps.
- FFT algorithm: $N \log N$ steps, enables new technology.

The challenge

Q. Will my program be able to solve a large practical input?

Why is my program so slow? Why does it run out of memory?

Insight. [Knuth 1970s] Use scientific method to understand performance.
Scientific method applied to the analysis of algorithms

A framework for predicting performance and comparing algorithms.

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**.
- Hypotheses must be **falsifiable**.

**Feature of the natural world.** Computer itself.

---

**Example: 3-SUM**

**3-SUM.** Given $N$ distinct integers, how many triples sum to exactly zero?

<table>
<thead>
<tr>
<th>$a[i]$</th>
<th>$a[j]$</th>
<th>$a[k]$</th>
<th>$\text{sum}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>-40</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>-20</td>
<td>-10</td>
</tr>
<tr>
<td>3</td>
<td>-40</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-10</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

---

**Context.** Deeply related to problems in computational geometry.

---

**3-SUM: brute-force algorithm**

```java
public class ThreeSum {
    public static int count(int[] a) {
        int N = a.length;
        int count = 0;
        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)
                        count++;
        return count;
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        int[] a = in.readInts();
        StdOut.println(count(a));
    }
}
```

- check each triple
- for simplicity, ignore integer overflow
Measuring the running time

Q. How to time a program?
A. Manual.

Empirical analysis

Run the program for various input sizes and measure running time.

```
% java ThreeSum 1Kints.txt
70
% java ThreeSum 2Kints.txt
528
% java ThreeSum 4Kints.txt
4039
```

Empirical analysis

Run the program for various input sizes and measure running time.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>?</td>
</tr>
</tbody>
</table>

† on some particular machine
Data analysis

**Standard plot.** Plot running time $T(N)$ vs. input size $N$.

![Standard plot](image)

**Log-log plot.** Plot running time $T(N)$ vs. input size $N$ using log-log scale.

![Log-log plot](image)

**Regression.** Fit straight line through data points: $a N^b$, where $a = 2^{c}$

Regression equation:

$$
\text{lg}(T(N)) = b \text{ lg } N + c
$$

- $b = 2.999$
- $c = -33.2103$

**Prediction and validation**

**Hypothesis.** The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

**Predictions.**
- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

**Observations.**

<table>
<thead>
<tr>
<th>$N$</th>
<th>Time (seconds)</th>
<th>Ratio</th>
<th>lg Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
<td>4.8</td>
<td>2.3</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>6.9</td>
<td>2.8</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>7.7</td>
<td>2.9</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8.0</td>
<td>2.8</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

$\frac{T(N)}{T(N/2)} = \frac{a N^b}{a(N/2)^b} = 2^b$

- $\text{lg}(6.4 / 0.8) = 3.0$

**Doubling hypothesis**

**Doubling hypothesis.** Quick way to estimate $b$ in a power-law relationship.

Run program, doubling the size of the input.

**Hypothesis.** Running time is about $a N^b$ with $b = \text{lg} \text{ ratio}$.

**Caveat.** Cannot identify logarithmic factors with doubling hypothesis.
Doubling hypothesis

Quick way to estimate \( b \) in a power-law relationship.

Q. How to estimate \( a \) (assuming we know \( b \))?
A. Run the program (for a sufficient large value of \( N \)) and solve for \( a \).

\[
\begin{array}{|c|c|}
\hline
N & \text{time (seconds)} \uparrow \\
\hline
8,000 & 51.1 \\
8,000 & 51.0 \\
8,000 & 51.1 \\
\hline
\end{array}
\]

\( 51.1 = a \times 8000^b \)

\( \Rightarrow a = 0.998 \times 10^{-10} \)

Hypothesis. Running time is about \( 0.998 \times 10^{-10} \times N^3 \) seconds.

almost identical hypothesis
to one obtained via regression

Analysis of algorithms quiz 1

Estimate the running time to solve a problem of size \( N = 96,000 \).

<table>
<thead>
<tr>
<th>( N )</th>
<th>time (seconds) ( \uparrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.02</td>
</tr>
<tr>
<td>2000</td>
<td>0.05</td>
</tr>
<tr>
<td>4,000</td>
<td>0.20</td>
</tr>
<tr>
<td>8,000</td>
<td>0.81</td>
</tr>
<tr>
<td>16,000</td>
<td>3.25</td>
</tr>
<tr>
<td>32,000</td>
<td>13.00</td>
</tr>
</tbody>
</table>

A. 39 seconds.
B. 52 seconds.
C. 117 seconds.
D. 350 seconds.
E. I don’t know.

Experimental algorithmics

System independent effects.
• Algorithm.
• Input data. \( \begin{array}{|c|c|}
\hline
\text{determines exponent} \ b \\
\text{in power law} \ a \cdot N^b \\
\hline
\end{array} \)

System dependent effects.
• Hardware: CPU, memory, cache, ...
• Software: compiler, interpreter, garbage collector, ...
• System: operating system, network, other apps, ...

\( \begin{array}{|c|c|}
\hline
\text{determines constant} \ a \\
\text{in power law} \ a \cdot N^b \\
\hline
\end{array} \)

Bad news. Sometimes difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

An aside

Algorithmic experiments are virtually free by comparison with other sciences.

Bottom line. No excuse for not running experiments to understand costs.
1.4 ANALYSIS OF ALGORITHMS

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory

Total running time: sum of cost \times frequency for all operations.
- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.

Example: 1-SUM

Q. How many instructions as a function of input size $N$?

```java
int count = 0;
for (int i = 0; i < N; i++)
if (a[i] == 0) count++;
```

<table>
<thead>
<tr>
<th>operation</th>
<th>cost (ns)</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2/5</td>
<td>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>1/5</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/10</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/10</td>
<td>$N$</td>
</tr>
<tr>
<td>array access</td>
<td>1/10</td>
<td>N to 2N</td>
</tr>
<tr>
<td>increment</td>
<td>1/10</td>
<td></td>
</tr>
</tbody>
</table>

\[ T(N) = 0 + 1 + \ldots + (N-2) + (N-1) \]

Pf. [ Gauss ]

\[
T(N) = \frac{1}{2} N(N-1) = \frac{1}{2} \cdot \sum_{i=1}^{N} i = \frac{N^2 - N}{2}
\]

Example: 2-SUM

Q. How many instructions as a function of input size $N$?

```java
int count = 0;
for (int i = 0; i < N; i++)
for (int j = i+1; j < N; j++)
if (a[i] + a[j] == 0) count++;
```

\[
0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N(N-1)
\]

\[
T(N) = N(N-1)/2
\]
**Example: 2-Sum**

Q. How many instructions as a function of input size \( N \)?

```java
int count = 0;
for (int i = 0; i < N; i++)
  for (int j = i+1; j < N; j++)
    if (a[i] + a[j] == 0)
      count++;
```

1/5 \( N (N - 1) \)

**Simplification 1: cost model**

Cost model. Use some basic operation as a proxy for running time.

```java
int count = 0;
for (int i = 0; i < N; i++)
  for (int j = i+1; j < N; j++)
    if (a[i] + a[j] == 0)
      count++;
```

1/5 \( N (N+1) \)

<table>
<thead>
<tr>
<th>operation</th>
<th>cost (ns)</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2/5</td>
<td>( N + 2 )</td>
</tr>
<tr>
<td>assignment statement</td>
<td>1/5</td>
<td>( N + 2 )</td>
</tr>
<tr>
<td>less than compare</td>
<td>1/5</td>
<td>( \frac{1}{2} (N+1) (N+2) )</td>
</tr>
<tr>
<td>equal to compare</td>
<td>1/10</td>
<td>( \frac{1}{2} N (N-1) )</td>
</tr>
<tr>
<td>array access</td>
<td>1/10</td>
<td>( N(N-1) )</td>
</tr>
<tr>
<td>increment</td>
<td>1/10</td>
<td>( \frac{1}{2} N (N+1) ) to ( N^2 )</td>
</tr>
</tbody>
</table>

0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \left( \frac{N}{2} \right)

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size \( N \).
  - Ignore lower order terms.
  - when \( N \) is large, terms are negligible
  - when \( N \) is small, we don’t care

Ex 1. \( \frac{1}{6} N^3 + 20 N + 16 \) \( \sim \) \( \frac{1}{6} N^3 \)

Ex 2. \( \frac{1}{6} N^3 + 100 N^{4/3} + 56 \) \( \sim \) \( \frac{1}{6} N^3 \)

Ex 3. \( \frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \) \( \sim \) \( \frac{1}{6} N^3 \)

**Technical definition.** \( f(N) \sim g(N) \) means \( \lim_{N \to \infty} \frac{f(N)}{g(N)} = 1 \)

Simplifying the calculations

"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordins."

— Alan Turing

**Rounding-off errors in matrix processes**

By A. M. Turing

(Focal Physical Laboratory, Paddington, Willesden)

[Mon 1 November 1957]

A number of methods of solving sets of linear equations and linear matrices are discussed. The theory of the rounding-off errors involved is considered the source of the mistakes. In all cases examined, including the well known 'Gauss elimination process,' it is found that the errors are normally quite unknown, i.e., by the built-up round error.
Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don’t care

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>N + 2</td>
<td>~N</td>
</tr>
<tr>
<td>assignment statement</td>
<td>N + 2</td>
<td>~N</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N+1)(N+2)$</td>
<td>~$\frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N(N-1)$</td>
<td>~$\frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>N(N-1)</td>
<td>~N^2</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N(N+1)$ to $N^2$</td>
<td>~$\frac{1}{2} N^2$ to ~$N^2$</td>
</tr>
</tbody>
</table>

Example: 2-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

A. ~$N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify counts.

Example: 3-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
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)
                count++;
```

A. ~$\frac{1}{2} N^3$ array accesses.

```
= \frac{N(N-1)(N-2)}{3!}
~ \frac{1}{6} N^3
```

Bottom line. Use cost model and tilde notation to simplify counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take a discrete mathematics course (COS 340).
Estimating a discrete sum

**Q.** How to estimate a discrete sum?

**A2.** Replace the sum with an integral, and use calculus!

**Ex 1.** $1 + 2 + \ldots + N.$

\[ \sum_{i=1}^{N} i \sim \int_{x=1}^{N} x \, dx \sim \frac{1}{2} N^2 \]

**Ex 2.** $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}.$

\[ \sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} \, dx = \ln N \]

**Ex 3.** 3-sum triple loop.

\[ \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3 \]

**Ex 4.** $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$

\[ \int_{x=0}^{\infty} \left( \frac{1}{2} \right)^x \, dx = \frac{1}{\ln 2} \approx 1.4427 \]

\[ \sum_{i=0}^{\infty} \left( \frac{1}{2} \right)^i = 2 \]

**Mathematical models for running time**

**In principle,** accurate mathematical models are available.

**In practice,**

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

\[ T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E \]

- $A =$ array access
- $B =$ integer add
- $C =$ integer compare
- $D =$ increment
- $E =$ variable assignment

**Bottom line.** We use **approximate** models in this course: $T(N) \sim c \, N^3.$

**Estimating a discrete sum**

**Q.** How to estimate a discrete sum?

**A3.** Use Maple or Wolfram Alpha.

**Analysis of algorithms quiz 2**

**How many array accesses does the following code fragment make as a function of $N$?**

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = 1; k < N; k = k*2)
            if (a[i] + a[j] >= a[k])
                count++;
```

**A.** $N^2 \lg N$

**B.** $\sim \frac{3}{2} \, N^2 \, \lg N$

**C.** $\sim \frac{1}{2} \, N^3$

**D.** $\sim \frac{3}{2} \, N^3$

**E.** I don’t know.
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- memory

1.4 Analysis of Algorithms

Common order-of-growth classifications

Definition. If \( f(N) \sim c g(N) \) for some constant \( c > 0 \), then the order of growth of \( f(N) \) is \( g(N) \).

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is \( N^3 \).

```java
int count = 0;
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)
                count++;
```

Typical usage. Mathematical analysis of running times.

where leading coefficient depends on machine, compiler, JVM, ...

Common order-of-growth classifications

<table>
<thead>
<tr>
<th>order of growth</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
<th>( T(2N) / T(N) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 )</td>
<td>constant</td>
<td>( a = b + c; )</td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>( \log N )</td>
<td>logarithmic</td>
<td>while ( (N &gt; 1) ) { ( N = N/2; \ldots ) }</td>
<td>divide in half binary search</td>
<td>( \sim 1 )</td>
<td></td>
</tr>
<tr>
<td>( N )</td>
<td>linear</td>
<td>for ( (int i = 0; i &lt; N; i++) ) { ( \ldots ) }</td>
<td>single loop</td>
<td>find the maximum</td>
<td>2</td>
</tr>
<tr>
<td>( N \log N )</td>
<td>linearithmic</td>
<td>see mergesort lecture</td>
<td>divide and conquer mergesort</td>
<td>( \sim 2 )</td>
<td></td>
</tr>
<tr>
<td>( N^2 )</td>
<td>quadratic</td>
<td>for ( (int i = 0; i &lt; N; i++) ) { ( \ldots ) }</td>
<td>double loop</td>
<td>check all pairs</td>
<td>4</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>cubic</td>
<td>for ( (int i = 0; i &lt; N; i++) ) { ( \ldots ) }</td>
<td>triple loop</td>
<td>check all triples</td>
<td>8</td>
</tr>
<tr>
<td>( 2^N )</td>
<td>exponential</td>
<td>see combinatorial search lecture</td>
<td>exhaustive search</td>
<td>check all subsets</td>
<td>( T(N) )</td>
</tr>
</tbody>
</table>
Binary search

Goal. Given a sorted array and a key, find index of the key in the array.

Binary search. Compare key against middle entry.
- Too small, go left.
- Too big, go right.
- Equal, found.

Binary search: implementation

Trivial to implement?
- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java’s Arrays.binarySearch() discovered in 2006.

Binary search: Java implementation

Invariant. If key appears in array a[], then a[lo] ≤ key ≤ a[hi].

```java
public static int binarySearch(int[] a, int key) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

Binary search: mathematical analysis

Proposition. Binary search uses at most \(1 + \log N\) key compares to search in a sorted array of size \(N\).

Def. \(T(N) = \#\) key compares to binary search a sorted subarray of size \(\leq N\).

Binary search recurrence. \(T(N) \leq T(N/2) + 1\) for \(N > 1\), with \(T(1) = 1\).

Pf sketch. [assume \(N\) is a power of 2]

\[
T(N) \leq T(N/2) + 1 \quad \text{[ given ]}
\leq T(N/4) + 1 + 1 \quad \text{[ apply recurrence to first term ]}
\leq T(N/8) + 1 + 1 + 1 \quad \text{[ apply recurrence to first term ]}
\vdots
\leq T(N/N) + 1 + 1 + \ldots + 1 \quad \text{[ stop applying, } T(1) = 1]\]

\[
= 1 + \log N \quad \text{[ apply } \log \text{ ]}
\]

Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley’s first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful Programming Pearls (Addison-Wesley, 1989; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
The 3-Sum Problem

3-Sum. Given $N$ distinct integers, find three such that $a + b + c = 0$.

Version 0. $N^3$ time, $N$ space.
Version 1. $N^2 \log N$ time, $N$ space.

Note. For full credit, running time should be worst case.

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-Sum is significantly faster in practice than the brute-force $N^3$ algorithm.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>1,000</td>
<td>0.14</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>2,000</td>
<td>0.18</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>4,000</td>
<td>0.34</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8,000</td>
<td>0.96</td>
</tr>
<tr>
<td>16,000</td>
<td>3.67</td>
<td>16,000</td>
<td>3.67</td>
</tr>
<tr>
<td>32,000</td>
<td>14.88</td>
<td>32,000</td>
<td>14.88</td>
</tr>
<tr>
<td>64,000</td>
<td>59.16</td>
<td>64,000</td>
<td>59.16</td>
</tr>
</tbody>
</table>

ThreeSum.java

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth $\Rightarrow$ faster in practice.

Basics

Bit. 0 or 1. NIST most computer scientists
Byte. 8 bits.
Megabyte (MB). 1 million or $2^{20}$ bytes.
Gigabyte (GB). 1 billion or $2^{30}$ bytes.

64-bit machine. We assume a 64-bit machine with 8-byte pointers.

Some JVMs "compress" ordinary object pointers to 4 bytes to avoid this cost.
Typical memory usage for primitive types and arrays

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

one-dimensional arrays

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>(2N + 24)</td>
</tr>
<tr>
<td>int[]</td>
<td>(4N + 24)</td>
</tr>
<tr>
<td>double[]</td>
<td>(8N + 24)</td>
</tr>
</tbody>
</table>

two-dimensional arrays

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>(\sim 2MN)</td>
</tr>
<tr>
<td>int[][]</td>
<td>(\sim 4MN)</td>
</tr>
<tr>
<td>double[][]</td>
<td>(\sim 8MN)</td>
</tr>
</tbody>
</table>

Typical memory usage summary

Total memory usage for a data type value:
- Primitive type: 4 bytes for int, 8 bytes for double, ...
- Object reference: 16 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

Note. Depending on application, we may want to count memory for any referenced objects (recursively).

Typical memory usage for objects in Java

Object overhead. 16 bytes.
Reference. 8 bytes.
Padding. Each object uses a multiple of 8 bytes.

Ex 1. A Date object uses 32 bytes of memory.

```
public class Date {
    private int day;
    private int month;
    private int year;
    ...
}
```

16 bytes (object overhead)

```
16 bytes (padding)
```

~ 32 bytes

Analysis of algorithms quiz 3

How much memory does a WeightedQuickUnionUF use as a function of \(N\)?

A. \(~ 4N\) bytes
B. \(~ 8N\) bytes
C. \(~ 4N^2\) bytes
D. \(~ 8N^2\) bytes
E. I don’t know.

```
public class WeightedQuickUnionUF {
    private int[] parent;
    private int[] size;
    private int count;
    public WeightedQuickUnionUF(int N) {
        parent = new int[N];
        size = new int[N];
        count = 0;
        for (int i = 0; i < N; i++)
            parent[i] = i;
        for (int i = 0; i < N; i++)
            size[i] = 1;
    }
    ...
}
```
Turning the crank: summary

Empirical analysis.
- Execute program to perform experiments.
- Assume power law.
- Formulate a hypothesis for running time.
- Model enables us to make predictions.

Mathematical analysis.
- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

Scientific method.
- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.