1.4 Analysis of Algorithms

- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

Reference: Intro to Programming in Java, Section 4.1

Running time

“As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?” — Charles Babbage

Cast of characters

Programmer needs to develop a working solution.

Client wants problem solved efficiently.

Theoretician wants to understand.

Student might play any or all of these roles someday.

Basic blocking and tackling is sometimes necessary.

Primary practical reason: avoid performance bugs.

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

client gets poor performance because programmer did not understand performance characteristics
Some algorithmic successes

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.

---

Scientific analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.
- Observe some feature of the universe.
- Hypothesize a model that is consistent with observation.
- 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.

Universe = computer itself.
Experimental algorithmics

Every time you run a program you are doing an experiment!

First step. Debug your program!
Second step. Choose input model for experiments.
Third step. Run and time the program for problems of increasing size.

Example: 3-sum

3-sum. Given $N$ integers, find all triples that sum to exactly zero.

```
% more input8.txt
8
30 -30 -20 -10 40 0 10 5
% java ThreeSum < input8.txt
4
30 -30   0
30 -20 -10
-30 -10  40
-10   0  10
```

Empirical analysis

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

```
ThreeSum.java

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.26</td>
</tr>
<tr>
<td>2000</td>
<td>2.16</td>
</tr>
<tr>
<td>4000</td>
<td>17.18</td>
</tr>
<tr>
<td>8000</td>
<td>137.76</td>
</tr>
</tbody>
</table>

↑ Running Linux on Sun-Fire-X4100
```

3-sum: brute-force algorithm

```java
public class ThreeSum {
    public static int count(int[] a) {
        int N = a.length;
        int cnt = 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)
                        cnt++;
        return cnt;
    }
    public static void main(String[] args) {
        long[] a = StdArrayIO.readInt1D();
        StdOut.println(count(a));
    }
}
```
Measuring the running time

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

Plot running time as a function of input size $N$.

Data analysis

Log-log plot. Plot running time vs. input size $N$ on log-log scale.

Regression. Fit straight line through data points: $a N^b$.

Hypothesis. Running time grows with the cube of the input size: $a N^3$. 

Measuring the running time

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

Stopwatch stopwatch = new Stopwatch();
ThreeSum.count(a);
double time = stopwatch.elapsedTime();
StdOut.println("Running time: " + time + " seconds");

public class Stopwatch
{
    private final long start = System.currentTimeMillis();
    public double elapsedTime()
    {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
}
Doubling hypothesis

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

Run program, doubling the size of the input.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
<th>ratio</th>
<th>$\lg$ ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>0.03</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>0.26</td>
<td>7.88</td>
<td>2.98</td>
</tr>
<tr>
<td>2,000</td>
<td>2.16</td>
<td>8.43</td>
<td>3.08</td>
</tr>
<tr>
<td>4,000</td>
<td>17.18</td>
<td>7.96</td>
<td>2.99</td>
</tr>
<tr>
<td>8,000</td>
<td>137.76</td>
<td>7.96</td>
<td>2.99</td>
</tr>
</tbody>
</table>

seems to converge to a constant $b = 3$

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

Caveat. Can’t identify logarithmic factors with doubling hypothesis.

Prediction and verification

Hypothesis. Running time is about $a N^3$ for input of size $N$.

Q. How to estimate $a$?

A. Run the program!

Refined hypothesis. Running time is about $2.7 \times 10^{-10}N^3$ seconds.

Prediction. 1,100 seconds for $N = 16,000$.

Observation.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>1118.86</td>
</tr>
</tbody>
</table>

validates hypothesis!

Experimental algorithmics

Many obvious factors affect running time:
- Machine.
- Compiler.
- Algorithm.
- Input data.

More factors (not so obvious):
- Caching.
- Garbage collection.
- Just-in-time compilation.
- CPU use by other applications.

Bad news. It is often difficult to get precise measurements.
Good news. Easier than other sciences.

E.g., can run huge number of experiments.

War story (from COS 126)

Q. How long does this program take as a function of $N$?

```java
public class EditDistance
{
    String s = StdIn.readString();
    int N = s.length();
    ...
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            distance[i][j] = ...
    ...
}
```

Jenny. $\sim c_1 N^2$ seconds.

Kenny. $\sim c_2 N$ seconds.
Mathematical models for running time

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

In principle, accurate mathematical models are available.

Cost of basic operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer add</td>
<td>a + b</td>
<td>2.1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>a * b</td>
<td>2.4</td>
</tr>
<tr>
<td>integer divide</td>
<td>a / b</td>
<td>5.4</td>
</tr>
<tr>
<td>floating point add</td>
<td>a + b</td>
<td>4.6</td>
</tr>
<tr>
<td>floating point multiply</td>
<td>a * b</td>
<td>4.2</td>
</tr>
<tr>
<td>floating point divide</td>
<td>a / b</td>
<td>13.5</td>
</tr>
<tr>
<td>sine</td>
<td>Math.sin(theta)</td>
<td>91.3</td>
</tr>
<tr>
<td>arctangent</td>
<td>Math.atan2(y, x)</td>
<td>129.0</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM.

Novice mistake. Abusive string concatenation.
Example: 1-sum

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>2</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>2</td>
</tr>
<tr>
<td>Less than compare</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>Equal to compare</td>
<td>$N$</td>
</tr>
<tr>
<td>Array access</td>
<td>$N$</td>
</tr>
<tr>
<td>Increment</td>
<td>$\leq 2N$</td>
</tr>
</tbody>
</table>

between $N$ (no zeros) and $2N$ (all zeros)

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. $6N^3 + 20N + 16$ ~ $6N^3$
Ex 2. $6N^3 + 100N^{4/3} + 56$ ~ $6N^3$
Ex 3. $6N^3 + 17N^2 \lg N + 7N$ ~ $6N^3$

discard lower-order terms
(e.g., $N = 10^{100}$: 6 billion vs. 169 million)

Technical definition. $f(N) \sim g(N)$ means

$$
\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1
$$

Example: 2-sum

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

```
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++;
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>Less than compare</td>
<td>$1/2(N + 1)(N + 2)$</td>
</tr>
<tr>
<td>Equal to compare</td>
<td>$1/2N(N - 1)^2$</td>
</tr>
<tr>
<td>Array access</td>
<td>$N(N - 1)$</td>
</tr>
<tr>
<td>Increment</td>
<td>$\leq N^2$</td>
</tr>
</tbody>
</table>

tedious to count exactly

0 + 1 + 2 + … + $(N - 1) = \frac{1}{2}N(N - 1)$

```
\sum_{k=1}^{N-1} k = \frac{1}{2}N(N - 1)
```

Example: 2-sum

Q. How long will it take as a function of $N$?

```
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++;
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
<th>Time per op</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration</td>
<td>~ N</td>
<td>$c_1$</td>
<td>~ $c_1N$</td>
</tr>
<tr>
<td>Assignment statement</td>
<td>~ N</td>
<td>$c_2$</td>
<td>~ $c_2N$</td>
</tr>
<tr>
<td>Less than compare</td>
<td>~ $1/2N^3$</td>
<td>$c_3$</td>
<td>~ $c_3N^2$</td>
</tr>
<tr>
<td>Equal to compare</td>
<td>~ $1/2N^2$</td>
<td>$c_4$</td>
<td>~ $c_4N^2$</td>
</tr>
<tr>
<td>Array access</td>
<td>~ $N^2$</td>
<td>$c_5$</td>
<td>~ $c_5N^2$</td>
</tr>
<tr>
<td>Increment</td>
<td>$\leq N^2$</td>
<td>$c_5$</td>
<td>$\leq c_5N^2$</td>
</tr>
</tbody>
</table>

```
total ~ $cN^2$
```

depends on input data

“inner loop”
Example: 3-sum

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

```c
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++;
```

Remark. Focus on instructions in inner loop; ignore everything else!

exponential bound: \( N^3 / 2 \)

may be in inner loop, depends on input data

### 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.

Bottom line. We use approximate models in this course: \( T_N \sim c N^3 \).

Bounding the sum by an integral trick

Q. How to estimate a discrete sum?


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

Ex 1. \( 1 \cdot 2 \cdot \ldots \cdot N \).

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

Ex 2. \( 1 + 1/2 + 1/3 + \ldots + 1/N \).

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

Ex 3. 3-sum triple loop.

\[
\sum_{i=1}^{N} \sum_{j=i+1}^{N} \sum_{k=j+1}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3
\]
### Common order-of-growth hypotheses

To determine order-of-growth:
- Assume a power law $T_N \sim a N^k$.
- Estimate exponent $k$ with doubling hypothesis.
- Validate with mathematical analysis.

#### Ex. ThreeSumDeluxe.java

**Food for precept. How is it implemented?**

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.26</td>
</tr>
<tr>
<td>2,000</td>
<td>2.16</td>
</tr>
<tr>
<td>4,000</td>
<td>17.18</td>
</tr>
<tr>
<td>8,000</td>
<td>137.76</td>
</tr>
</tbody>
</table>

### ThreeSum.java

#### Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</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; statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>log $N$</td>
<td>logarithmic</td>
<td>while ($N &gt; 1$) { $N = N / 2$; ... }</td>
<td>divide in half, binary search</td>
<td>~ 1</td>
</tr>
<tr>
<td>$N$</td>
<td>linear</td>
<td>for (int $i = 0$; $i &lt; N$; $i++$) { ... }</td>
<td>loop, 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>~ 2</td>
</tr>
<tr>
<td>$N^2$</td>
<td>quadratic</td>
<td>for (int $i = 0$; $i &lt; N$; $i++$) for (int $j = 0$; $j &lt; N$; $j++$) { ... }</td>
<td>double loop, check all pairs</td>
<td>4</td>
</tr>
<tr>
<td>$N^3$</td>
<td>cubic</td>
<td>for (int $i = 0$; $i &lt; N$; $i++$) for (int $j = 0$; $j &lt; N$; $j++$) for (int $k = 0$; $k &lt; N$; $k++$) { ... }</td>
<td>triple loop, 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, check all possibilities</td>
<td>$T(N)$</td>
</tr>
</tbody>
</table>

### Good news. the small set of functions

1. $N \log N$, $N$, $N \log N$, $N^3$, and $2^N$ suffices to describe order-of-growth of typical algorithms.
Types of analyses

**Best case.** Lower bound on cost.
- Determined by "easiest" input.
- Provides a goal for all inputs.

**Worst case.** Upper bound on cost.
- Determined by "most difficult" input.
- Provides guarantee for all inputs.

**Average case.** "Expected" cost.
- Need a model for "random" input.
- Provides a way to predict performance.

Commonly-used notations

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tilde</strong></td>
<td>leading term</td>
<td>$\sim 10 N^2$</td>
<td>$10 N^2 + 22 N \log N + 37$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td><strong>Big Theta</strong></td>
<td>asymptotic growth rate</td>
<td>$\Theta(N^2)$</td>
<td>$N^2 9000 N^2$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td><strong>Big Oh</strong></td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$100 N$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td><strong>Big Omega</strong></td>
<td>$\Theta(N^2)$ and larger</td>
<td>$\Omega(N^2)$</td>
<td>$9000 N^2$</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

**Common mistake.** Interpreting big-Oh as an approximate model.

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.
- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).
Typical memory requirements for arrays in Java

**Array overhead.** 16 bytes.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>2N + 16</td>
</tr>
<tr>
<td>int[]</td>
<td>4N + 16</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 16</td>
</tr>
</tbody>
</table>

**Ex.** An N-by-N array of doubles consumes ~ 8N² bytes of memory.

Typical memory requirements for objects in Java

**Object overhead.** 8 bytes.

**Reference.** 4 bytes.

**Ex 1.** A Complex object consumes 24 bytes of memory.

```java
public class Complex {
    private double re;
    private double im;
    ...}
```

<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>

Typical memory requirements for primitive types in Java

**Bit.** 0 or 1.

**Byte.** 8 bits.

**Megabyte (MB).** 1 million bytes.

**Megabyte (MB).** 1 billion bytes.

---

Typical object memory requirements

**Ex 1.**

```java
public class Complex {
    private double re;
    private double im;
    ...}
```
Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 2. A virgin String of length N consumes ~ 2N bytes of memory.

```java
public class String {
    private int offset;
    private int count;
    private int hash;
    private char[] value;
    ...}
```

8 bytes overhead for object

4 bytes

4 bytes

4 bytes

4 bytes for reference (plus 2N + 16 bytes for array)

2N + 40 bytes

Example 1

Q. How much memory does QuickUWPC use as a function of N?

A.

```java
public class QuickUWPC {
    private int[] id;
    private int[] sz;
    ...}
```

```java
public QuickUWPC(int N) {
    id = new int[N];
    sz = new int[N];
    for (int i = 0; i < N; i++) id[i] = i;
    for (int i = 0; i < N; i++) sz[i] = 1;
}
```

```java
public boolean find(int p, int q) {
    ...}
```

```java
public void unite(int p, int q) {
    ...}
```

Example 2

Q. How much memory does this code fragment use as a function of N?

A.

```java
... int N = Integer.parseInt(args[0]);
for (int i = 0; i < N; i++) {
    int[] a = new int[N];
    ... }
```

Remark. Java automatically reclaims memory when it is no longer in use.

Turning the crank: summary

In principle, accurate mathematical models are available.

In practice, approximate mathematical models are easily achieved.

Timing may be flawed?
- Limits on experiments insignificant compared to other sciences.
- Mathematics might be difficult?
- Only a few functions seem to turn up.
- Doubling hypothesis cancels complicated constants.

Actual data might not match input model?
- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.