CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 2: Analysis of Algorithms

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin
→ Parts from S&W 1.4
→ Estimate the performance of algorithms by
  → Experiments & Observations
  → Precise Mathematical Calculations
“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 (1864)
→ **Good programmer:** to predict the performance of our programs.

→ **Good client:** to choose between alternative algorithms/implementations.

→ **Good manager:** to provide guarantees to clients / avoid client complaints.

→ **Good theoritician:** to understand the nature of computing.
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.

<table>
<thead>
<tr>
<th>time</th>
<th>64T</th>
<th>32T</th>
<th>16T</th>
<th>8T</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>1K</td>
<td>2K</td>
<td>4K</td>
<td>8K</td>
</tr>
</tbody>
</table>

Legend:
- quadratic
- linearithmic
- linear

Friedrich Gauss
1805
Some algorithmic successes

N-body simulation.

- Simulate gravitational interactions among $N$ bodies.
- Brute force: $N^2$ steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.

Andrew Appel
PU '81
The challenge

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

Insight. [Knuth 1970s] Use scientific method to understand performance.
Scientific method applied to 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.
Experimental Approach:

Measuring Precise Running Time
Example: 3-SUM

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

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

<table>
<thead>
<tr>
<th>a[i]</th>
<th>a[j]</th>
<th>a[k]</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>-40</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>-20</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>-40</td>
<td>40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-10</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Context. Deeply related to problems in computational geometry.
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.readAllInts();
        StdOut.println(count(a));
    }
}
The input of ThreeSum is an array of size $N$. Suppose we care only about 100-element arrays. There are many different 100-element arrays.

```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;
    }
}
```
The input of ThreeSum is an array of size N.
Suppose we care only about 100-element arrays.
There are many different 100-element arrays.

Q. Is the running time of ThreeSum dependent on which 100-element array we provide as input?

```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;
    }
}
```
Measuring the running time

Q. How to time a program?
A. Manual.
Measuring the running time

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

```java
public class Stopwatch (part of stdlib.jar)

Stopwatch()
create a new stopwatch

double elapsedTime()
time since creation (in seconds)

public static void main(String[] args)
{
    In in = new In(args[0]);
    int[] a = in.readAllInts();
    Stopwatch stopwatch = new Stopwatch();
    StdOut.println(ThreeSum.count(a));
    double time = stopwatch.elapsedTime();
    StdOut.println("elapsed time " + time);
}
```
Empirical analysis

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

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

<table>
<thead>
<tr>
<th>( N )</th>
<th>time (seconds) ( t )</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>
Data analysis

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

![Diagram showing a standard plot with problem size on the x-axis and running time on the y-axis.]
Data analysis

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

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

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.
Try out the experimental analysis:

https://docs.google.com/spreadsheets/d/
1WnhyK6g1pYdcT2ndZOqNRRkTtTitXkWKn0rTgCnM-bw8/edit?usp=sharing
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>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>8,000</td>
<td>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>410.8</td>
</tr>
</tbody>
</table>

"order of growth" of running time is about $N^3$ [stay tuned]

validates hypothesis!
**Doubling hypothesis**

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

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>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>3.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

\[
\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b
\]

$\lg (6.4 / 0.8) = 3.0$

seems to converge to a constant $b \approx 3$

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

**Caveat.** Cannot identify logarithmic factors with doubling hypothesis.
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 $a$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds) $\uparrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>8,000</td>
<td>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

\[ 51.1 = a \times 8000^3 \]

\[ \Rightarrow a = 0.998 \times 10^{-10} \]

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

almost identical hypothesis to one obtained via linear regression
Experimental algorithmics

**System independent effects.**

- Algorithm.
- Input data.

**System dependent effects.**

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

**Bad news.** Difficult to get precise measurements.

**Good news.** Much easier and cheaper than other sciences.

e.g., can run huge number of experiments
This was the experimental approach to algorithm analysis.

Is there a mathematical approach where we can do calculations instead of experiments?
Mathematical Approach 1:

Calculating Precise Running Time
Mathematical models for running time

**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.

---

In principle, accurate mathematical models are available.
Cost of basic operations

**Challenge.** How to estimate constants.

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM
Cost of basic operations

**Observation.** Most primitive operations take constant time.

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds $\uparrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>int a</td>
<td>$c_1$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>$c_2$</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>$c_3$</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>$c_4$</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>$c_6 N$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>$c_7 N^2$</td>
</tr>
</tbody>
</table>

**Caveat.** Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation
Example: 1-SUM

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

```c
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>$N$ to $2N$</td>
</tr>
</tbody>
</table>
**Example: 2-SUM**

**Q.** How many instructions 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++;
```

**Pf.** [ $n$ even ]

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

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

half of square

half of diagonal
String theory infinite sum

$$1 + 2 + 3 + 4 + \ldots = -\frac{1}{12}$$

http://www.nytimes.com/2014/02/04/science/in-the-end-it-all-adds-up-to.html
Example: 2-SUM

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

```c
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)
```

<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>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$\frac{1}{2} N (N - 1)$ to $N (N - 1)$</td>
</tr>
</tbody>
</table>

tedious to count exactly
Calculating precise running time

\[ T_N = c_1A + c_2B + c_3C + c_4D + c_5E \]

Where

- \( c_1 \): cost of array access
- \( A \): number of array accesses
- \( c_2 \): cost of integer addition
- \( B \): number of integer additions
- \( c_3 \): cost of integer comparison
- \( C \): number of integer comparisons
- \( c_4 \): cost of increment
- \( D \): number of increments
- \( c_5 \): cost of assignment
- \( E \): number of assignments
Calculating Precise Running Time

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

Where

- \( c_1 \): cost of array access
- \( A \): number of array accesses
- \( c_2 \): cost of integer addition
- \( B \): number of integer additions
- \( c_3 \): cost of integer comparison
- \( C \): number of integer comparisons
- \( c_4 \): cost of increment
- \( D \): number of increments
- \( c_5 \): cost of assignment
- \( E \): number of assignments

Q. Advantages / Disadvantages?