4.1 Performance

"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

The Challenge

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

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 we design must be reproducible.
- Hypothesis must be falsifiable.

Key insight. [Knuth 1970s]
Use the scientific method to understand performance.
Reasons to Analyze Algorithms

**Predict performance.**
- Will my program finish?
- When will my program finish?

**Compare algorithms.**
- Will this change make my program faster?
- How can I make my program faster?

**Basis for inventing new ways to solve problems.**
- Enables new technology.
- Enables new research.

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.

**N-body Simulation.**
- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- Barnes-Hut: N log N steps, enables new research.

Three-Sum Problem

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

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

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

Q. How would you write a program to solve the problem?
public class ThreeSum {
    // return number of distinct triples (i, j, k)
    // such that (a[i] + a[j] + a[k] == 0)
    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) {
        int[] a = StdArrayIO.readInt1D();
        StdOut.println(count(a));
    }
}

Empirical Analysis

Empirical analysis. Run the program for various input sizes.

<table>
<thead>
<tr>
<th>N</th>
<th>time  †</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.03</td>
</tr>
<tr>
<td>1024</td>
<td>0.26</td>
</tr>
<tr>
<td>2048</td>
<td>2.16</td>
</tr>
<tr>
<td>4096</td>
<td>17.18</td>
</tr>
<tr>
<td>8192</td>
<td>136.76</td>
</tr>
</tbody>
</table>

† Running Linux on Sun-Fire-X4100 with 16GB RAM

Stopwatch

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

% java ThreeSum < 1Kints.txt

% java ThreeSum < 2Kints.txt
Stopwatch

Q. How to time a program?
A. A Stopwatch object.

```java
public class Stopwatch {
    private final long start;

    public Stopwatch() {
        start = System.currentTimeMillis();
    }

    public double elapsedTime() {
        return (System.currentTimeMillis() - start) / 1000.0;
    }
}
```

Q. How to time a program?
A. A Stopwatch object.

```java
public static void main(String[] args) {
    int[] a = StdArrayIO.readInt1D();
    Stopwatch timer = new Stopwatch();
    StdOut.println(count(a));
    StdOut.println(timer.elapsedTime());
}
```

Empirical Analysis

Data analysis. Plot running time vs. input size $N$.

Q. How fast does running time grow as a function of input size $N$?

Initial hypothesis. Running time obeys power law $f(N) = a N^b$.

Data analysis. Plot running time vs. input size $N$ on a log-log scale.

Consequence. Power law yields straight line (slope = b).

Refined hypothesis. Running time grows as cube of input size: $a N^3$. 
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 †</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.033</td>
<td>-</td>
</tr>
<tr>
<td>1024</td>
<td>0.26</td>
<td>7.88</td>
</tr>
<tr>
<td>2048</td>
<td>2.16</td>
<td>8.43</td>
</tr>
<tr>
<td>4096</td>
<td>17.18</td>
<td>7.96</td>
</tr>
<tr>
<td>8192</td>
<td>136.76</td>
<td>7.96</td>
</tr>
</tbody>
</table>

seems to converge to a constant $c = 8$

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

Performance Challenge 1

Let $F(N)$ be running time of $\text{main()}$ as a function of input $N$.

```java
public static void main(String[] args) {
    ...
    int N = Integer.parseInt(args[0]);
    ...
}
```

**Scenario 1.** $F(2N) / F(N)$ converges to about 4.

Q. What is order of growth of the running time?

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

Q. How to estimate $a$?

A. Run the program!

<table>
<thead>
<tr>
<th>$N$</th>
<th>time †</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>17.18</td>
</tr>
<tr>
<td>4096</td>
<td>17.15</td>
</tr>
<tr>
<td>4096</td>
<td>17.17</td>
</tr>
</tbody>
</table>

$17.17 = a 4096^3$  
$\Rightarrow a = 2.5 \times 10^{-10}$

**Refined hypothesis.** Running time is about $2.5 \times 10^{-10} \times N^3$ seconds.

**Prediction.** 1,100 seconds for $N = 16,384$.

**Observation.**

<table>
<thead>
<tr>
<th>$N$</th>
<th>time †</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>1118.86</td>
</tr>
</tbody>
</table>
Mathematical Analysis

Running time. Count up frequency of execution of each instruction and weight by its execution time.

```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++;
```

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>N + 2</td>
</tr>
<tr>
<td>variable assignment</td>
<td>N + 2</td>
</tr>
<tr>
<td>less than comparison</td>
<td>1/2 (N + 1) (N + 2)</td>
</tr>
<tr>
<td>equal to comparison</td>
<td>1/2 N (N - 1)</td>
</tr>
<tr>
<td>array access</td>
<td>N (N-1)</td>
</tr>
<tr>
<td>increment</td>
<td>≤ 2N</td>
</tr>
</tbody>
</table>

Tilde Notation

Tilde notation.

- Estimate running time 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. \(6N^3 + 17N^2 + 56\) \(\sim 6N^3\)
Ex 2. \(6N^3 + 100N^{43} + 56\) \(\sim 6N^3\)
Ex 3. \(6N^3 + 17N^2\log N\) \(\sim 6N^3\)

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

Running time. Count up frequency of execution of each instruction and weight by its execution time.

Inner loop. Focus on instructions in "inner loop."

Constants in Power Law

Power law. Running time of a typical program is $\sim a N^b$.

Exponent $b$ depends on: algorithm.

Leading constant $a$ depends on:
- Algorithm.
- Input data.
- Caching.
- Machine.
- Compiler.
- Garbage collection.
- Just-in-time compilation.
- CPU use by other applications.

Our approach. Use doubling hypothesis (or mathematical analysis) to estimate exponent $b$, run experiments to estimate $a$.

Analysis: Empirical vs. Mathematical

Empirical analysis.
- Measure running times, plot, and fit curve.
- Easy to perform experiments.
- Model useful for predicting, but not for explaining.

Mathematical analysis.
- Analyze algorithm to estimate # ops as a function of input size.
- May require advanced mathematics.
- Model useful for predicting and explaining.

Critical difference. Mathematical analysis is independent of a particular machine or compiler; applies to machines not yet built.

Order of Growth Classifications

Observation. A small subset of mathematical functions suffice to describe running time of many fundamental algorithms.
Order of Growth Classifications

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Description</th>
<th>Factor for Doubling Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>\log \text{N}</td>
<td>1</td>
</tr>
<tr>
<td>Linear</td>
<td>\text{N}</td>
<td>2</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>\text{N} \log \text{N}</td>
<td>2</td>
</tr>
<tr>
<td>Quadratic</td>
<td>\text{N}^2</td>
<td>4</td>
</tr>
<tr>
<td>Cubic</td>
<td>\text{N}^3</td>
<td>8</td>
</tr>
<tr>
<td>Exponential</td>
<td>\text{2}^\text{N}</td>
<td>\text{2}^\text{N}</td>
</tr>
</tbody>
</table>

Commonly encountered growth functions

Order of Growth: Consequences

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Predicted Running Time if Problem Size is Increased by a Factor of 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>a few seconds</td>
</tr>
<tr>
<td>Linear</td>
<td>a few minutes</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>a few minutes</td>
</tr>
<tr>
<td>Quadratic</td>
<td>several hours</td>
</tr>
<tr>
<td>Cubic</td>
<td>a few weeks</td>
</tr>
<tr>
<td>Exponential</td>
<td>forever</td>
</tr>
</tbody>
</table>

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

Dynamic Programming

Binomial Coefficients

Binomial coefficient: \binom{n}{k} = \text{number of ways to choose } k \text{ of } n \text{ elements.}

Ex. Number of possible 7-card poker hands = \binom{52}{7} = 2,598,960.

Ex. Probability of "quads" in Texas hold 'em:

\binom{13}{4} \times \binom{4}{3} \times \binom{52}{4} \times 133,784,360 = 224,848 (about 594:1)
Binomial Coefficients

Binomial coefficient. \( \binom{n}{k} \) = number of ways to choose \( k \) of \( n \) elements.

Pascal’s identity. \( \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \)

contains first element
excludes first element

Pascal’s triangle.

Binomial Coefficients: Sierpinski Triangle

Binomial coefficient. \( \binom{n}{k} \) = number of ways to choose \( k \) of \( n \) elements.

Sierpinski triangle. Color black the odd integers in Pascal’s triangle.

Performance Challenge 3

Q. Is this an efficient way to compute binomial coefficients?
A. No, no, no! [same essential recomputation flaw as naïve Fibonacci]

public class SlowBinomial {
   // natural recursive implementation
   public static long binomial(long n, long k) {
      if (k == 0) return 1;
      if (n == 0) return 0;
      return binomial(n-1, k-1) + binomial(n-1, k);
   }
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int K = Integer.parseInt(args[1]);
      StdOut.println(binomial(N, K));
   }
}
Timing Experiments

Timing experiments: direct recursive solution.

Q. Is running time linear, quadratic, cubic, exponential in N?

<table>
<thead>
<tr>
<th>(2N, N)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(26, 13)</td>
<td>0.46</td>
</tr>
<tr>
<td>(28, 14)</td>
<td>1.27</td>
</tr>
<tr>
<td>(30, 15)</td>
<td>4.30</td>
</tr>
<tr>
<td>(32, 16)</td>
<td>15.69</td>
</tr>
<tr>
<td>(34, 17)</td>
<td>57.40</td>
</tr>
<tr>
<td>(36, 18)</td>
<td>230.42</td>
</tr>
</tbody>
</table>

Increase N by 1, running time increases by about 4x

Performance Challenge 4

Let F(N) be running time to compute binomial(2N, N).

Observation. F(N+1) / F(N) converges to about 4.

Q. What is order of growth of the running time?

A. Exponential: a 4^N.

```
public static long binomial(long n, long k) {
    if (k == 0) return 1;
    if (n == 0) return 0;
    return binomial(n-1, k-1) + binomial(n-1, k);
}
```

Will not finish unless N is small

Binomial Coefficients: Dynamic Programming

Key idea. Save solutions to subproblems to avoid recomputation.

```
public class Binomial {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int K = Integer.parseInt(args[1]);
        long[][] bin = new long[N+1][K+1];

        // base cases
        for (int k = 0; k <= K; k++) bin[0][K] = 0;
        for (int n = 0; n <= N; n++) bin[N][0] = 1;

        // bottom-up dynamic programming
        for (int n = 1; n <= N; n++)
            for (int k = 1; k <= K; k++)
                bin[n][k] = bin[n-1][k-1] + bin[n-1][k];

        // print results
        StdOut.println(bin[N][K]);
    }
}
```

Tradeoff. Trade (a little) memory for (a huge amount of) time.

Binomial Coefficients: Dynamic Programming

```
public class Binomial {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int K = Integer.parseInt(args[1]);
        long[][] bin = new long[N+1][K+1];

        // base cases
        for (int k = 0; k <= K; k++) bin[0][K] = 0;
        for (int n = 0; n <= N; n++) bin[N][0] = 1;

        // bottom-up dynamic programming
        for (int n = 1; n <= N; n++)
            for (int k = 1; k <= K; k++)
                bin[n][k] = bin[n-1][k-1] + bin[n-1][k];

        // print results
        StdOut.println(bin[N][K]);
    }
}
```
Timing Experiments

Timing experiments for binomial coefficients via dynamic programming.

<table>
<thead>
<tr>
<th>(2N, N)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(26, 13)</td>
<td>instant</td>
</tr>
<tr>
<td>(28, 14)</td>
<td>instant</td>
</tr>
<tr>
<td>(30, 15)</td>
<td>instant</td>
</tr>
<tr>
<td>(32, 16)</td>
<td>instant</td>
</tr>
<tr>
<td>(34, 17)</td>
<td>instant</td>
</tr>
<tr>
<td>(36, 18)</td>
<td>instant</td>
</tr>
</tbody>
</table>

Q. Is running time linear, quadratic, cubic, exponential in N?

Performance Challenge 5

Let $F(N)$ be running time to compute $\text{binomial}(2N, N)$ using DP.

```java
for (int n = 1; n <= N; n++)
    for (int k = 1; k <= K; k++)
        bin[n][k] = bin[n-1][k-1] + bin[n-1][k];
```

Q. What is order of growth of the running time?

A. Quadratic: $N^2$.

Remark. There is a profound difference between $4^N$ and $N^2$.

Digression: Stirling’s Approximation

Alternative: $\binom{n}{k} = \frac{n!}{k! (n-k)!}$

Caveat. $52!$ overflows a long, even though final result doesn’t.

Instead of computing exact values, use Stirling’s approximation:

$$\ln n! = n \ln n - n + \ln(2\pi n) \frac{1}{2} + \frac{1}{12n} - \frac{1}{360n^2} + \frac{1}{1260n^3}$$

Application. Probability of exact $k$ heads in $n$ flips with a biased coin.

$$\binom{n}{k} p^k (1-p)^{n-k}$$ (easy to compute approximate value with Stirling’s formula)

Memory
**Typical Memory Requirements for Java Data Types**

- **Bit.** 0 or 1.
- **Byte.** 8 bits.
- **Megabyte (MB).** 1 million bytes ~ 2^{10} bytes.
- **Gigabyte (GB).** 1 billion bytes ~ 2^{20} bytes.

---

**Performance Challenge 6**

Q. How much memory does this program require as a function of \( N \)?

```java
public class RandomWalk {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int[][] count = new int[N][N];
        int x = N/2;
        int y = N/2;
        for (int i = 0; i < N; i++) {
            // no new variable declared in loop
            ...
            count[x][y]++;
        }
    }
}
```

---

**Summary**

Q. How can I evaluate the performance of my program?

A. Computational experiments, mathematical analysis, scientific method.

Q. What if it’s not fast enough? Not enough memory?

- Understand why.
- Buy a faster computer.
- Learn a better algorithm (COS 226, COS 423).
- Discover a new algorithm.

<table>
<thead>
<tr>
<th>attribute</th>
<th>better machine</th>
<th>better algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>$$$ or more</td>
<td>$ or less</td>
</tr>
<tr>
<td>applicability</td>
<td>makes &quot;everything&quot; run faster</td>
<td>does not apply to some problems</td>
</tr>
<tr>
<td>improvement</td>
<td>quantitative improvements</td>
<td>dramatic qualitative improvements possible</td>
</tr>
</tbody>
</table>