CS 112 Introduction to Programming
(Spring 2012)

Lecture #24: Performance

Zhong Shao

Department of Computer Science
Yale University
Office: 314 Watson

http://flint.cs.yale.edu/cs112

Acknowledgements: some slides used in this class are taken directly or adapted from those accompanying the textbook: Introduction to Programming in Java: An Interdisciplinary Approach by Robert Sedgewick and Kevin Wayne (Copyright 2002-2010)
“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?

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**.
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

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

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.

Freidrich Gauss
1805
Algorithmic Successes

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.
Three-Sum Problem

Three-sum problem. Given \( N \) integers, how many triples sum to 0?

Context. Deeply related to problems in computational geometry.

```plaintext
% 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 {

    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));
    }
}

all possible triples $i < j < k$ such that $a[i] + a[j] + a[k] = 0$
Empirical Analysis
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>1,024</td>
<td>0.26</td>
</tr>
<tr>
<td>2,048</td>
<td>2.16</td>
</tr>
<tr>
<td>4,096</td>
<td>17.18</td>
</tr>
<tr>
<td>8,192</td>
<td>136.76</td>
</tr>
</tbody>
</table>

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

**Caveat.** If $N$ is too small, you will measure mainly noise.
Q. How to time a program?
A. A stopwatch.
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 class Stopwatch {
    Stopwatch() {
        // create a new stopwatch and start it running
    }
    double elapsedTime() {
        // return the elapsed time since creation, in seconds
    }
}

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 approximately obeys a power law $T(N) = a N^b$.

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

**Consequence.** Power law yields straight line.

**Refined hypothesis.** Running time grows as cube of input size: $a N^3$. 

**Empirical Analysis**

- **slope** = 3
- **slope** = $b$
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 $^\dagger$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.033</td>
<td>-</td>
</tr>
<tr>
<td>1,024</td>
<td>0.26</td>
<td>7.88</td>
</tr>
<tr>
<td>2,048</td>
<td>2.16</td>
<td>8.43</td>
</tr>
<tr>
<td>4,096</td>
<td>17.18</td>
<td>7.96</td>
</tr>
<tr>
<td>8,192</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 $b = \log_c c$. 
Performance Challenge 1

Let $T(N)$ be running time of main() as a function of input size $N$.

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

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

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

1. $N$
2. $N^2$
3. $N^3$
4. $N^4$
5. $2^N$
Performance Challenge 2

Let $T(N)$ be running time of `main()` as a function of input size $N$.

Scenario 2. $T(2N) / T(N)$ converges to about 2.

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

1. $N$   
2. $N^2$   
3. $N^3$   
4. $N^4$   
5. $2^N$
Prediction and Validation

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

**Q.** How to estimate $a$?  
**A.** Run the program!

| $N$  | $time$  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4,096</td>
<td>17.18</td>
</tr>
<tr>
<td>4,096</td>
<td>17.15</td>
</tr>
<tr>
<td>4,096</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.**

| $N$  | $time$  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16,384</td>
<td>1118.86</td>
</tr>
</tbody>
</table>

validates hypothesis
Mathematical Analysis

Donald Knuth
Turing award '74
**Mathematical Analysis**

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

```java
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>variable assignment</td>
<td>2</td>
</tr>
<tr>
<td>less than comparison</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to comparison</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)
Mathematical Analysis

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

<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>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to comparison</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>$\leq N^2$</td>
</tr>
</tbody>
</table>

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

becoming very tedious to count
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.** $6N^3 + 100N^{4/3} + 56 \sim 6N^3$

**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.

```java
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;
}
```

**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.

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

- \( \lg N \) (logarithmic)
- \( \lg N = \log_2 N \)

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

- \( N \) (linear)

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

- \( N^2 \) (quadratic)

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

- \( N\lg N \) (linearithmic)

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

- \( 2^N \) (exponential)
Order of Growth Classifications

Orders of growth (log-log plot)

<table>
<thead>
<tr>
<th>order of growth</th>
<th>function</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 N</td>
<td>1</td>
</tr>
<tr>
<td>linear</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>linearithmic</td>
<td>N log N</td>
<td>2</td>
</tr>
<tr>
<td>quadratic</td>
<td>N^2</td>
<td>4</td>
</tr>
<tr>
<td>cubic</td>
<td>N^3</td>
<td>8</td>
</tr>
<tr>
<td>exponential</td>
<td>2^N</td>
<td>2^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>
<th>Order of Growth</th>
<th>Predicted Factor of Problem Size Increase if Computer Speed is Increased by a Factor of 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>A few minutes</td>
<td>Linear</td>
<td>10</td>
</tr>
<tr>
<td>Linearithmic</td>
<td>A few minutes</td>
<td>Linearithmic</td>
<td>10</td>
</tr>
<tr>
<td>Quadratic</td>
<td>Several hours</td>
<td>Quadratic</td>
<td>3-4</td>
</tr>
<tr>
<td>Cubic</td>
<td>A few weeks</td>
<td>Cubic</td>
<td>2-3</td>
</tr>
<tr>
<td>Exponential</td>
<td>Forever</td>
<td>Exponential</td>
<td>1</td>
</tr>
</tbody>
</table>

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

*Effect of increasing computer speed on problem size that can be solved in a fixed amount of time*
Memory
Typical Memory Requirements for Primitive Types

**Bit.** 0 or 1.

**Byte.** 8 bits.

**Megabyte (MB).** 1 million bytes ~ $2^{10}$ bytes.

**Gigabyte (GB).** 1 billion bytes ~ $2^{20}$ bytes.

<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>

**Q.** How much memory (in bytes) does your computer have?
Typical Memory Requirements for Reference Types

**Memory of an object.**
- Memory for each instance variable, plus
- Object overhead = 8 bytes on a 32-bit machine.

16 bytes on a 64-bit machine

```java
public class Charge {
    private double rx;
    private double ry;
    private double q;
    ...
}
```

**Memory of a reference.** 4 byte pointer on a 32-bit machine.

8 bytes on a 64-bit machine
Typical Memory Requirements for Array Types

Memory of an array.

- Memory for each array entry.
- Array overhead = 16 bytes on a 32-bit machine.

24 bytes on a 64-bit machine

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[]</td>
<td>4N + 16</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 16</td>
</tr>
<tr>
<td>Charge[]</td>
<td>36N + 16</td>
</tr>
<tr>
<td>int[][]</td>
<td>4N^2 + 20N + 16</td>
</tr>
<tr>
<td>double[][]</td>
<td>8N^2 + 20N + 16</td>
</tr>
<tr>
<td>String</td>
<td>2N + 40</td>
</tr>
</tbody>
</table>

Q. What's the biggest double[][] array you can store on your computer?
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 or more memory.
- Learn a better algorithm. see CS 223
- Discover a new algorithm. see CS 365

<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>
Dynamic Programming
Binomial Coefficients

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

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

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

\[
\frac{\begin{pmatrix} 13 \\ 1 \end{pmatrix} \begin{pmatrix} 4 \\ 4 \end{pmatrix} \times \begin{pmatrix} 48 \\ 3 \end{pmatrix}}{\begin{pmatrix} 52 \\ 7 \end{pmatrix}} = \frac{224,848}{133,784,560} \approx \frac{1}{594} \]
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.
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));
    }

    }

Pascal's identity
Performance Challenge 3

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

<table>
<thead>
<tr>
<th>(2N, N)</th>
<th>time (t)</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>

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

increase \(N\) by 1, running time increases by about 4x
Let $F(N)$ be running time to compute $\text{binomial}(2N, N)$.

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

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

A. Exponential: $a 4^N$. will not finish unless $N$ is small
Dynamic Programming

**Key idea.** Save solutions to subproblems to avoid recomputation.

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}
\]

\[
20 = 10 + 10
\]

**Tradeoff.** Trade (a little) memory for (a huge amount of) time.
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 = 1; 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: \( a N^2 \).  

**Remark.** There is a profound difference between \( 4^N \) and \( N^2 \).