Analysis of Algorithms

overview
experiments
models
case study
hypotheses

Updated from:
- Algorithms in Java, Chapter 2
  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

Running time
Charles Babbage (1864)
Analytic Engine

1

Reasons to analyze algorithms

- Predict performance
- Compare algorithms
- Provide guarantees
- Understand theoretical basis

Primary practical reason: avoid performance bugs

Client gets poor performance because programmer did not understand performance characteristics

Overview

Scientific analysis of algorithms:
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:
   Decide on model for experiments on large inputs.

Third step:
   Run the program for problems of increasing size.

Experimental evidence: measuring time
• Manual:
• Automatic:  Stopwatch.java

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

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

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

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: we can run a huge number of experiments [stay tuned]

Approach 1: Settle for affordable approximate results
Approach 2: Count abstract operations (machine independent)
Models for the analysis of algorithms

**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.

### Developing models for algorithm performance

- **In principle**, accurate mathematical models are available [Knuth]
- **In practice**, formulas can be complicated
- Advanced mathematics might be required

**Ex.**

\[ T_N = 24 A_N + 11 B_N + 4 C_N + 3 D_N + 7 N + 9 S_N \]

where
- \[ A_N = \frac{2(N+1)}{3} \]
- \[ B_N = (N + 1)(2H_{N+1} - 2H_3 - 1)/6 + 1/2 \]
- \[ C_N = (N + 1)(2H_{N+1} - 2H_3 + 1) \]
- \[ D_N = (N + 1)(1 - 2H_3/3) \]
- \[ S_N = (N + 1)/5 - 1 \]

**Exact models best left for experts**

**Bottom line**: We use approximate models in this course: \( T_N \sim c N \log N \)

### Commonly used notations to model running time

<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>Big Theta</td>
<td>growth rate</td>
<td>( \Theta(N^2) )</td>
<td>( 9000 N^2 )</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>( \Theta(N^2) ) and smaller</td>
<td>( O(N^2) )</td>
<td>( 5 \cdot 10^5 N^2 + 22 N \log N N + 3N )</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td>Big Omega</td>
<td>( \Theta(N^2) ) and larger</td>
<td>( \Omega(N^2) )</td>
<td>( 9000 N^2 )</td>
<td>develop lower bounds</td>
</tr>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>( \sim 10 N^2 )</td>
<td>( 10 N^2 )</td>
<td>provide approximate model</td>
</tr>
</tbody>
</table>

All constants rolled into one.
Predictions and guarantees

Theory of algorithms: The running time of an algorithm is $O(f(N))$

- **advantages**
  - describes guaranteed performance
  - $O$-notation absorbs input model

- **challenges**
  - cannot use to predict performance
  - cannot use to compare algorithms

This course: The running time of an algorithm is $\sim c f(N)$

- **advantages**
  - can use to predict performance
  - can use to compare algorithms

- **challenges**
  - need to develop accurate input model
  - may not provide guarantees

---

Case study [stay tuned for numerous algorithms and applications]

Sorting problem: rearrange $N$ given items into ascending order

```java
public static void less(double x, double y) {
    return x < y;
}
```

```java
public static void exch(double[] a, int i, int j) {
    double t = a[i];
    a[i] = a[j];
    a[j] = t;
}
```

Basic operations: compares and exchanges
Selection sort: an elementary sorting algorithm

Algorithm invariants
- ↑ scans from left to right.
- Elements to the left of ↑ are fixed and in ascending order.
- No element to left of ↑ is larger than any element to its right.

in final order

Selection sort inner loop
- move the pointer to the right
- identify index of minimum item on right.
- Exchange into position.

```
for (int i = 0; i < a.length; i++)
    { int min = i;
      for (int j = i+1; j < a.length; j++)
        if (less(a[j], a[min]))
          min = j;
      exch(a, i, min);
    }
```

Selection sort: Java implementation

```
public static void sort(double[] a)
{
   for (int i = 0; i < a.length; i++)
     { int min = i;
       for (int j = i+1; j < a.length; j++)
         if (less(a[j], a[min]))
           min = j;
       exch(a, i, min);
     }
}
```

Selection sort: initial observations

Observe, tabulate and plot operation counts for various values of N.
- study most frequently performed operation (compares)
- input model: N random numbers between 0 and 1

```
<table>
<thead>
<tr>
<th>N</th>
<th>compares</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000</td>
<td>2.1 million</td>
</tr>
<tr>
<td>4,000</td>
<td>7.9 million</td>
</tr>
<tr>
<td>8,000</td>
<td>32.1 million</td>
</tr>
<tr>
<td>16,000</td>
<td>125.9 million</td>
</tr>
<tr>
<td>32,000</td>
<td>514.7 million</td>
</tr>
</tbody>
</table>
```

Observe, tabulate and plot operation counts for various values of N.
- study most frequently performed operation (compares)
- input model: N random numbers between 0 and 1

```java
public static void sort(double[] a)
{
   for (int i = 0; i < a.length; i++)
     { int min = i;
       for (int j = i+1; j < a.length; j++)
         if (less(a[j], a[min]))
           min = j;
       exch(a, i, min);
     }
}
```
Selection sort: experimental hypothesis

Data analysis. Plot # compares vs. input size on log-log scale.

Regression. Fit straight line through data points.

Hypothesis. # compares is \( \sim N^{2/2} \)

Selection sort: theoretical model

Hypothesis: number of compares is \( N + (N-1) + \ldots + 2 + 1 \sim N^2/2 \)

Selection sort: validation

Implicit assumptions

• constant cost per compare
• cost of compares dominates running time

Hypothesis: Running time is \( \sim c N^2 \)

Regression fit validates hypothesis.

A scientific connection between program and natural world.
Insertion sort: another elementary sorting algorithm

Algorithm invariants
• \( \uparrow \) scans from left to right.
• Elements to the left of \( \uparrow \) are in ascending order.

Insertion sort inner loop
• move the pointer to the right
• moving from right to left, exchange \( a[i] \) with each larger element to its left

for (int \( j = i; j > 0; j-- \))
    if (less(\( a[j] \), \( a[j-1] \)))
        exch(\( a, j, j-1 \));
    else break;

Maintains algorithm invariants

Insertion sort: theoretical model

Insertions are halfway back, on the average, for randomly ordered input

Hypothesis: number of compares is \((1 + 2 + \ldots + (N-1) + N)/2 \sim N^2/4\) on the average, for randomly ordered input
Experimental comparison of insertion sort and selection sort

Plot both running times on log-log scale
- slopes are the same (both 2)
- both are quadratic

Compute ratio of running times
% java SortCompare Insertion Selection 4000
For 4000 random double values
Insertion is 1.7 times faster than selection

Need detailed analysis to prefer one over the other

Neither is useful for huge randomly-ordered files

Would Be Nice (if analysis of algorithms were always this easy), But

Mathematics might be difficult
- Ex. It is known that properties of singularities of functions in the complex plane play a role in analysis of many algorithms

Leading term might not be good enough
- Ex. Selection sort could be linear-time if cost of exchanges is huge
  assumption that compares dominate may be invalid

Actual data might not match model
- Ex. Insertion sort could be linear-time if keys are roughly in order
  assumption that input is randomly ordered may be invalid

Timing may be flawed
- different results on different computers
- different results on same computer at different times

Practical approach to developing hypotheses

First step: determine asymptotic growth rate for chosen model
- approach 1: run experiments, regression
- approach 2: do the math
- best: do both

Good news: the relatively small set of functions $1, \log N, N, N \log N, N^2, N^3,$ and $2^N$ suffices to describe asymptotic growth rate of typical algorithms

After determining growth rate
- use doubling hypothesis (to predict performance)
- use ratio hypothesis (to compare algorithms)
### Common asymptotic-growth hypotheses (summary)

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>a = b + c;</td>
<td>statement</td>
<td>add two numbers</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>while (N &gt; 1)</td>
<td>divide in half</td>
<td>binary search</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>for (int i = 0; i &lt; N; i++)</td>
<td>loop</td>
<td>find the maximum</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>[see next lecture]</td>
<td>divide and conquer</td>
<td>sort an array</td>
</tr>
<tr>
<td>N^2</td>
<td>quadratic</td>
<td>for (int i = 0; i &lt; N; i++)</td>
<td>double loop</td>
<td>check all pairs</td>
</tr>
<tr>
<td>N^3</td>
<td>cubic</td>
<td>for (int i = 0; i &lt; N; i++)</td>
<td>triple loop</td>
<td>check all triples</td>
</tr>
<tr>
<td>2^N</td>
<td>exponential</td>
<td>[see lecture 24]</td>
<td>exhaustive search</td>
<td>check all possibilities</td>
</tr>
</tbody>
</table>

### Aside: practical implications of asymptotic growth

#### For back-of-envelope calculations, assume

<table>
<thead>
<tr>
<th>decade</th>
<th>processor speed</th>
<th>instructions per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970s</td>
<td>1M Hz</td>
<td>10^4</td>
</tr>
<tr>
<td>1980s</td>
<td>10M Hz</td>
<td>10^7</td>
</tr>
<tr>
<td>1990s</td>
<td>100M Hz</td>
<td>10^8</td>
</tr>
<tr>
<td>2000s</td>
<td>1G Hz</td>
<td>10^9</td>
</tr>
</tbody>
</table>

#### How long to process millions of inputs?

Ex. Population of NYC was "millions" in 1970s; still is

#### How many inputs can be processed in minutes?

Ex. Customers lost patience waiting "minutes" in 1970s; still do

### Practical implications of asymptotic-growth: another view

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>description</th>
<th>effect on a program that runs for a few seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>a few seconds</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>a few seconds</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N^2</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
</tr>
<tr>
<td>N^3</td>
<td>cubic</td>
<td>not practical for large problems</td>
<td>several weeks</td>
</tr>
<tr>
<td>2^N</td>
<td>exponential</td>
<td>useful only for tiny problems</td>
<td>forever</td>
</tr>
</tbody>
</table>
Developing asymptotic order of growth hypotheses with doubling

To formulate hypothesis for asymptotic growth rate:
- compute $T(2N)/T(N)$ as accurately (and for $N$ as large) as is affordable
- use this table

<table>
<thead>
<tr>
<th>ratio</th>
<th>hypothesis</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant or logarithmic</td>
<td>$c / c = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c \log 2N / c \log N \approx 1$</td>
</tr>
<tr>
<td>2</td>
<td>linear or linearithmic</td>
<td>$c 2N / c N = 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c 2N\log (2N) / c N \log N \approx 2$</td>
</tr>
<tr>
<td>4</td>
<td>quadratic</td>
<td>$c (2N)^2 / c N^2 = 4$</td>
</tr>
<tr>
<td>9</td>
<td>cubic</td>
<td>$c (2N)^3 / c N^3 = 9$</td>
</tr>
</tbody>
</table>

Example revisited: methods for timing sort algorithms

```java
public static double time(String alg, Double[] a) {
    Stopwatch sw = new Stopwatch();
    if (alg.equals("Insertion")) Insertion.sort(a);
    if (alg.equals("Selection")) Selection.sort(a);
    if (alg.equals("Shell")) Shell.sort(a);
    if (alg.equals("Merge")) Merge.sort(a);
    if (alg.equals("Quick")) Quick.sort(a);
    return sw.elapsedTime();
}
```

```java
public static double timetrials(String alg, int N, int trials) {
    double total = 0.0;
    Double[] a = new Double[N];
    for (int t = 0; t < trials; t++) {
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
        total += time(alg, a);
    }
    return total;
}
```

A practical approach to predict running time:
- analyze algorithm and run experiments to develop hypothesis that asymptotic growth rate of running time is $\sim c T(N)$
- run algorithm for some value of $N$, measure running time
- prediction: increasing input size by a factor of 2 increases running time by a factor of $T(2N)/T(N)$

Use algorithm itself to implicitly compute leading-term constant

Example: selection sort
growth rate | name
--- | ---
$c$ | constant
$\log N$ | logarithmic
$N$ | linear
$N \log N$ | linearithmic
$N^2$ | quadratic
$N^3$ | cubic

Predicting performance with doubling hypotheses

```java
public class SortGrowth {
    public static void main(String[] args) {
        String alg = args[0];
        int N = 1000;
        if (args.length > 1)
            N = Integer.parseInt(args[1]);
        int trials = 100;
        if (args.length > 2)
            trials = Integer.parseInt(args[2]);
        double ratio = timetrials(alg, 2*N, trials) /
                       timetrials(alg, N, trials);
        StdOut.printf("Ratio is %f
", ratio);
        if (ratio > 1.8 && ratio < 2.2)
            StdOut.printf(" %s is linear or linearithmic
", alg);
        if (ratio > 3.8 && ratio < 4.2)
            StdOut.printf(" %s is quadratic
", alg);
    }
}
```

```
% java SortGrowth Selection
Ratio is 4.1
Selection is quadratic
```

```
% java SortGrowth Insertion
Ratio is 3.645756
Insertion is quadratic
```

Example: selection sort

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>$T(2N)/T(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>constant</td>
<td>$1$</td>
</tr>
<tr>
<td>$\log N$</td>
<td>logarithmic</td>
<td>~1</td>
</tr>
<tr>
<td>$N$</td>
<td>linear</td>
<td>$2$</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>linearithmic</td>
<td>~2</td>
</tr>
<tr>
<td>$N^2$</td>
<td>quadratic</td>
<td>$4$</td>
</tr>
<tr>
<td>$N^3$</td>
<td>cubic</td>
<td>$9$</td>
</tr>
</tbody>
</table>
Predicting performance with doubling hypotheses

```java
public class SortPredict {
    public static void main(String[] args) {
        String alg = args[0];
        int trials = 100;
        if (args.length > 1) trials = Integer.parseInt(args[1]);
        StdOut.printf("Seconds for %d trials\n", trials);
        StdOut.printf("predicted actual\n%7.1f
%5d %7.1f", total, 2*N, guess);
        old = total;
    }
}
```

Note: SortGrowth is not needed!

[This code works for any power law.]

Comparing algorithms with ratio hypothesis

```java
public class SortCompare {
    public static void main(String[] args) {
        String alg1 = args[0];
        String alg2 = args[1];
        int N = Integer.parseInt(args[2]);
        int trials = 100;
        if (args.length > 3) trials = Integer.parseInt(args[3]);
        double time1 = 0.0;
        double time2 = 0.0;
        Double[] a1 = new Double[N];
        Double[] a2 = new Double[N];
        for (int t = 0; t < trials; t++) {
            for (int i = 0; i < N; i++) {
                a1[i] = Math.random();
                a2[i] = a1[i];
            }
            time1 += time(alg1, a1);
            time2 += time(alg2, a2);
        }
        StdOut.printf("For %d random Double values\n    %s is %s times faster than %s\n", N, alg1);
        StdOut.println(time2/time1, alg2);
    }
}
```

Summary: turning the crank

Yes, analysis of algorithms might be challenging, BUT

Mathematics might be difficult?
- only a few functions seem to turn up
- doubling, ratio tests cancel complicated constants

Leading term might not be good enough?
- debugging tools are available to identify bottlenecks
- typical programs have short inner loops

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

Timing may be flawed?
- limits on experiments insignificant compared to other sciences
- different computers are different!