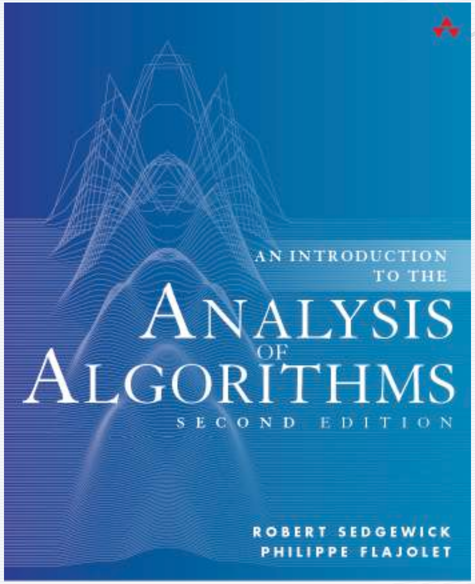
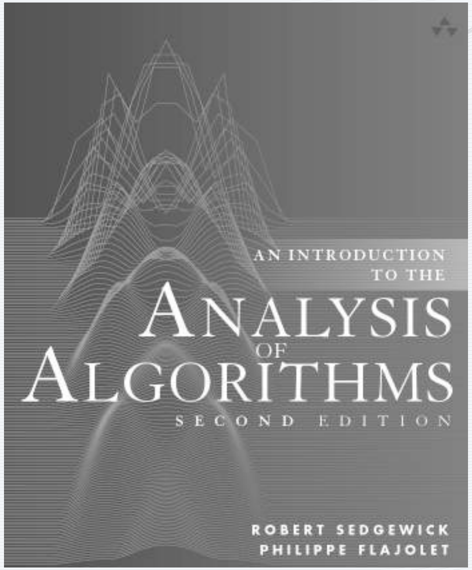


http://aofa.cs.princeton.edu

1. Analysis of Algorithms



http://aofa.cs.princeton.edu

1. Analysis of Algorithms

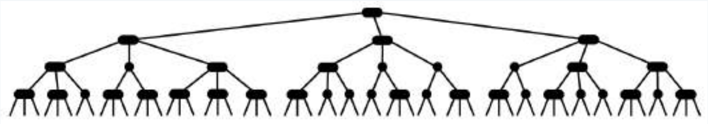
- **History and motivation**
- A scientific approach
- Example: Quicksort
- Resources

Why Analyze an Algorithm?

1. Classify problems and algorithms by difficulty.

2. Predict performance, compare algorithms, tune parameters.

3. Better understand and improve implementations and algorithms.



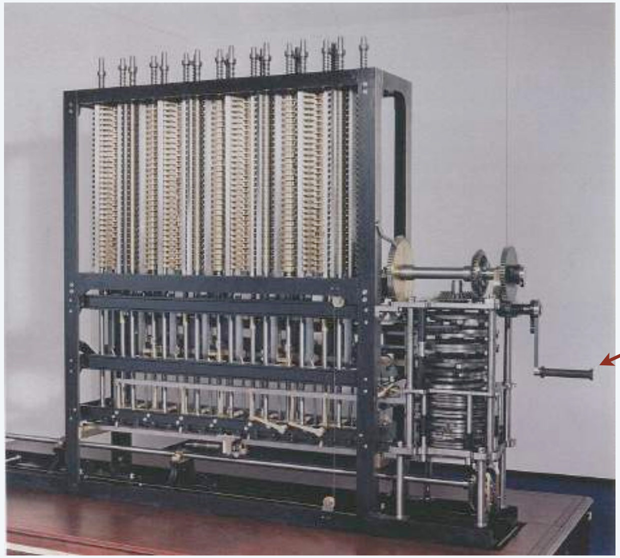
Intellectual challenge: AofA is even more interesting than programming!

Analysis of Algorithms (Babbage, 1860s)

"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)

Analytic Engine



how many times do you have to turn the crank?

Analysis of Algorithms (Turing (!), 1940s)

"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process . . ."

— Alan Turing (1947)

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Analysis of Algorithms (Knuth, 1960s)

To analyze an algorithm:

- Develop a good implementation.
- Identify unknown quantities representing the basic operations.
- Determine the cost of each basic operation.
- Develop a realistic model for the input.
- Analyze the frequency of execution of the unknown quantities.
- Calculate the total running time: $\sum_{q} \text{frequency}(q) \times \text{cost}(q)$

BENEFITS:

Scientific foundation for AofA.

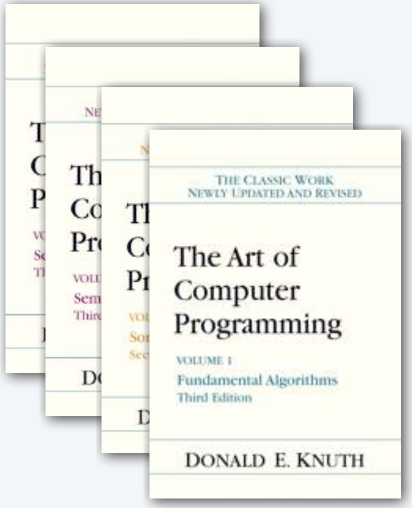
Can predict performance and compare algorithms.

DRAWBACKS:

Model may be unrealistic.

Too much detail in analysis.

D. E. Knuth



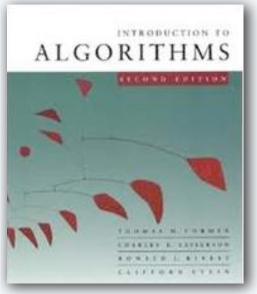
Theory of Algorithms (AHU, 1970s; CLR, present day)

To address Knuth drawbacks:

- Analyze worst-case cost
  [takes model out of the picture].
- Use O-notation for upper bound
  [takes detail out of analysis].
- Classify algorithms by these costs.

Aho, Hopcroft and Ullman

Cormen, Leiserson, Rivest, and Stein



BENEFIT: Enabled a new Age of Algorithm Design.

DRAWBACK: Cannot use to predict performance or compare algorithms.
(An elementary fact that is often overlooked!)

Example: Two sorting algorithms

Quicksort
- Worst-case number of compares: $O(N^2)$
- Classification $O(N^2)$

Mergesort
- Worst-case number of compares: $N \log N$
- Classification $O(N \log N)$

BUT

Quicksort is twice as fast as Mergesort in practice and uses half the space

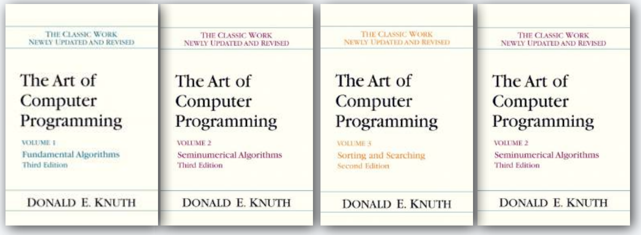
How do we know?
By analyzing both algorithms! (stay tuned)

Cannot use O- upper bounds to predict performance or compare algorithms.

Analytic combinatorics context

Drawbacks of Knuth approach:

- Model may be unrealistic.
- Too much detail in analysis.

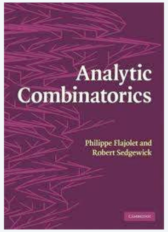


Drawbacks of AHU/CLRS approach:

- Worst-case performance may not be relevant.
- Cannot use O- upper bounds to predict or compare.

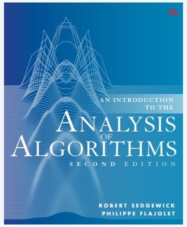
Analytic combinatorics can provide:

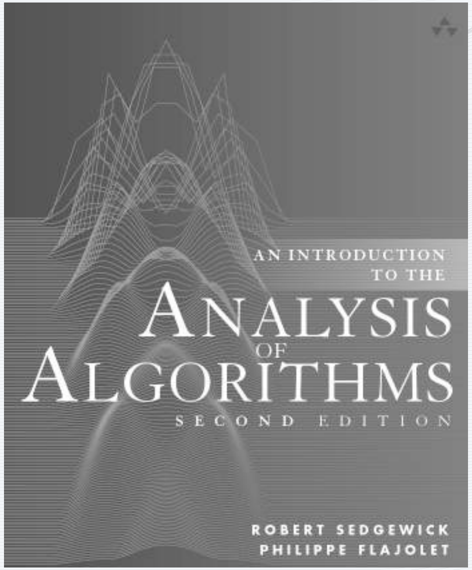
- A calculus for developing models.
- General theorems that avoid detail in analysis.



AC Part I (this course):

- Underlying mathematics.
- Introduction to analytic combinatorics.
- Classical applications in AofA and combinatorics.

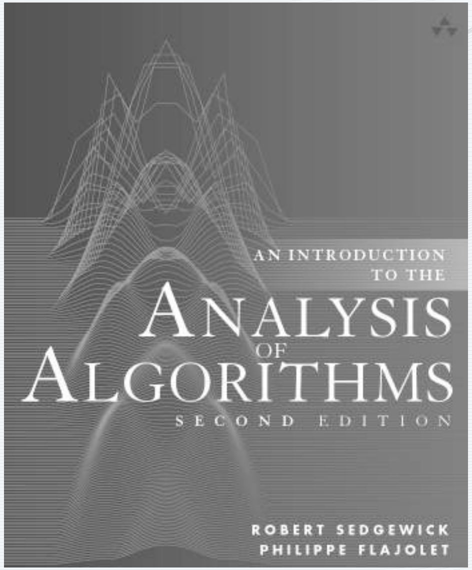




http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- Resources



http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- Resources

Notation for theory of algorithms

"Big-Oh" notation for upper bounds

$$g(N) = O(f(N)) \text{ iff } |g(N)/f(N)| \text{ is bounded from above as } N \to \infty$$

"Omega" notation for lower bounds

$$g(N) = \boldsymbol{\Omega}(f(N)) \text{ iff } |g(N)/f(N)| \text{ is bounded from below as } N \to \infty$$

"Theta" notation for order of growth ("within a constant factor")

$$g(N) = \Theta(f(N)) \text{ iff } g(N) = O(f(N)) \text{ and } g(N) = \boldsymbol{\Omega}(f(N))$$

O-notation considered dangerous

How to predict performance (and to compare algorithms)?

Not the scientific method: O-notation

Theorem: Running time is $O(N^c)$ ✘

not at all useful for predicting performance

Scientific method calls for tilde-notation.

Hypothesis: Running time is $\sim aN^c$ ✓

an effective path to predicting performance

O-notation is useful for many reasons, BUT
Common error: Thinking that O-notation is useful for predicting performance

Surely, we can do better

A typical exchange in Q&A

RS (in a talk):	O-notation considered dangerous. Cannot use it to predict performance.
Q:	?? O(N log N) surely beats O(N^2)
RS:	Not by the definition. O expresses upper bound.
Q:	So, use Theta.
RS:	Still (typically) bounding the worst case. Is the input a worst case?
Q:	(whispers to colleague) I'd use the Θ(N log N) algorithm, wouldn't you?

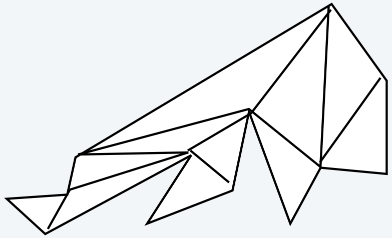
Galactic algorithms

R.J. Lipton: A galactic algorithm is one that will never be used.

Why? Any effect would never be noticed in this galaxy.

Ex. Chazelle's linear-time triangulation algorithm

- theoretical tour-de-force
- too complicated to implement
- cost of implementing would exceed savings in this galaxy, anyway



One blogger's conservative estimate:

75% SODA, 95% STOC/FOCS are galactic

OK for basic research to drive agenda, BUT

Common error: Thinking that a galactic algorithm is useful in practice.

Surely, we can do better

An actual exchange with a theoretical computer scientist:

TCS (in a talk): Algorithm A is bad.
Google should be interested in my new Algorithm B.

RS: What's the matter with Algorithm A?

TCS: It is not optimal. It has an extra O(log log N) factor.

RS: But Algorithm B is very complicated, lg lg N is less than 6 in this universe, and that is just an upper bound. Algorithm A is certainly going to run 10 to 100 times faster in any conceivable real-world situation. Why should Google care about Algorithm B?

TCS: Well, I like Algorithm B. I don't care about Google.

Analysis of Algorithms (scientific approach)

Start with complete implementation suitable for application testing.

Analyze the algorithm by

- Identifying an abstract operation in the inner loop.
- Develop a realistic model for the input to the program.
- Analyze the frequency of execution C_N of the op for input size N.

Hypothesize that the cost is $\sim aC_N$ where a is a constant.

Validate the hypothesis by

- Developing generator for input according to model.
- Calculate a by running the program for large input.
- Run the program for larger inputs to check the analysis.

Validate the model by testing in application contexts.

Refine and repeat as necessary

Sedgewick and Wayne
Algorithms, 4th edition
Section 1.4

Notation (revisited)

"Big-Oh" notation for upper bounds

$$g(N) = O(f(N)) \text{ iff } |g(N)/f(N)| \text{ is bounded from above as } N \to \infty$$

"Omega" notation for lower bounds

$$g(N) = \Omega(f(N)) \text{ iff } |g(N)/f(N)| \text{ is bounded from below as } N \to \infty$$

"Theta" notation for order of growth ("within a constant factor")

$$g(N) = \Theta(f(N)) \text{ iff } g(N) = O(f(N)) \text{ and } g(N) = \Omega(f(N))$$

for theory of algorithms

for analysis to predict performance and to compare algorithms

"Tilde" notation for asymptotic equivalence

$$g(N) \sim f(N) \text{ iff } |g(N)/f(N)| \to 1 \text{ as } N \to \infty$$

Components of algorithm analysis

Empirical

- Run algorithm to solve real problem.
- Measure running time and/or count operations.

Challenge: need good implementation

```
% java SortTest 1000000
        10        44.44
       100       847.85
      1000     12985.91
     10000    175771.70
    100000   2218053.41
```

Mathematical

- Develop mathematical model.
- Analyze algorithm within model.

Challenge: need good model, need to do the math

$$C_N = N + 1 + \sum_{1 \le k \le N} \frac{1}{N}(C_k + C_{N-k-1})$$

Scientific

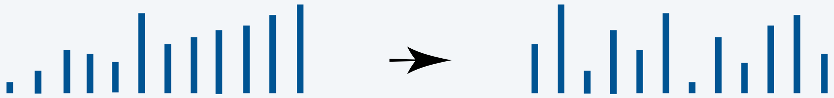
- Run algorithm to solve real problem.
- Check for agreement with model.

Challenge: need all of the above

```
% java QuickCheck 1000000
        10        44.44        26.05
       100       847.85       721.03
      1000     12985.91     11815.51
     10000    175771.70    164206.81
    100000   2218053.41   2102585.09
```

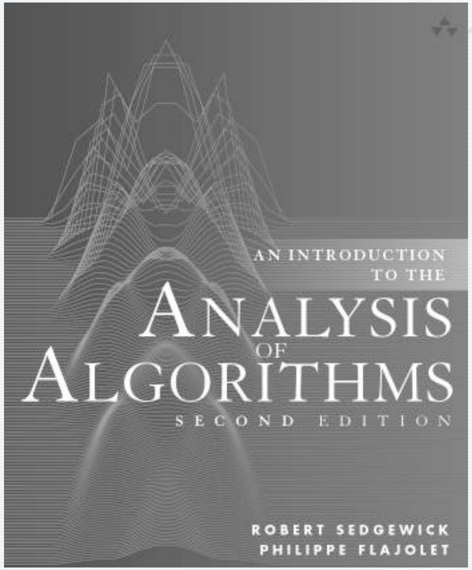
Potential drawbacks to the scientific approach

1. Model may not be realistic.
   - A challenge in any scientific discipline.
   - Advantage in CS: we can *randomize* to make the model apply.



2. Math may be too difficult.
   - A challenge in any scientific discipline (cf. statistical physics).
   - A “calculus” for AofA is the motivation for this course!

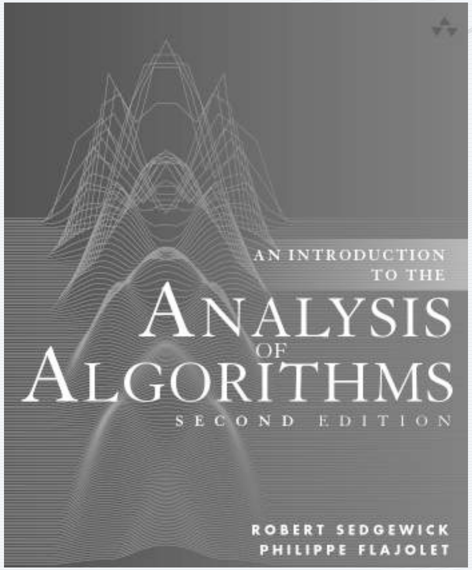
3. Experiments may be too difficult.
   - Not compared to other scientific disciplines.
   - Can’t implement? Why analyze?



http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- **A scientific approach**
- Example: Quicksort
- Resources



http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- Resources

Example: Quicksort

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {
      int i = lo, j = hi+1;
      while (true)
      {
         while (less(a[++i], a[lo])) if (i == hi) break;
         while (less(a[lo], a[--j])) if (j == lo) break;
         if (i >= j) break;
         exch(a, i, j);
      }
      exch(a, lo, j);
      return j;
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
   }
}
```

Section 2.3

http://algs4.cs.princeton.edu/23quicksort/Quick.java

Start: Preliminary decisions

Cost model

- running time?
- better approach: separate algorithm from implementation
- for sorting, associate *compares* with inner loop.
- Hypothesis: if number of compares is C, running time is $\sim aC$

timing

Input model

- assume input randomly ordered (easy to arrange)
- assume keys all different (not always easy to arrange)

counting

Key question: Are models/assumptions realistic?

Stay tuned.

Setup: Relevant questions about quicksort

Assume array of size *N* with entries distinct and randomly ordered.

Q. How many compares to partition?

A. ***N*+1**

Q. What is the probability that the partitioning item is the kth smallest?

A. **1/*N***

Q. What is the size of the subarrays in that case?

A. ***k* −1** and ***N* − *k***

Q. Are the subarrays randomly ordered after partitioning?

A. **YES.**

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {
      int i = lo, j = hi+1;
      while (true)
      {
         while (less(a[++i], a[lo])) if (i == hi) break;
         while (less(a[lo], a[--j])) if (j == lo) break;
         if (i >= j) break;
         exch(a, i, j);
      }
      exch(a, lo, j);
      return j;
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
   }
}
```

Main step: Formulate a mathematical problem

Recursive program and input model lead to a *recurrence relation*.

Assume array of size N with entries distinct and randomly ordered.

Let C_N be the expected number of compares used by quicksort.

$$\underbrace{C_N = N + 1}_{\text{for partitioning}} + \sum_{1 \le k \le N} \underbrace{\frac{1}{N}}_{\text{probability k is the partitioning element}} \underbrace{(C_{k-1} + C_{N-k})}_{\text{compares for subarrays when k is the partitioning element}}$$

Simplifying the recurrence

$$C_N = N + 1 + \sum_{1 \le k \le N} \frac{1}{N}(C_{k-1} + C_{N-k}) \qquad C_0 = 0$$

both sums are
$C_0 + C_1 + \ldots + C_{N-1}$

Apply symmetry.

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \le k \le N} C_{k-1}$$

Multiply both sides by N.

$$NC_N = N(N+1) + 2 \sum_{1 \le k \le N} C_{k-1}$$

Subtract same formula for *N*–1.

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

Collect terms.

$$NC_N = (N+1)C_{N-1} + 2N$$

Aside

Simplified recurrence gives efficient algorithm for computing result

$$C_N = N + 1 + \sum_{0 \le k \le N-1} \frac{1}{N}(C_k + C_{N-k-1})$$

QUADRATIC time

```
c[0] = 0;
for (int N = 1; N <= maxN; N++)
{
   c[N] = N+1;
   for (int k = 0; k < N;  k++)
      c[N] += (c[k] + c[N-1-k])/N;
}
```

$$NC_N = (N+1)C_{N-1} + 2N$$

LINEAR time

```
c[0] = 0;
for (int N = 1; N <= maxN; N++)
   c[N] = (N+1)*c[N-1]/N + 2;
```

AofA: Finding a fast way to compute the running time of a program

Solving the recurrence

$$NC_N = (N+1)C_{N-1} + 2N$$

Tricky (but key) step:
divide by $N(N+1)$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Telescope.

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$= \frac{C_1}{2} + \frac{2}{3} + \ldots + \frac{2}{N} + \frac{2}{N+1}$$

Simplify (ignore small terms).

$$C_N \sim 2N \sum_{1 \geq k \geq N} \frac{1}{k} - 2N$$

Approximate with an
integral (stay tuned)

N

$$C_N \sim 2N\left(\int_1^{\infty} \frac{1}{x} dx + \gamma\right) - 2N$$

$$= 2N \ln N - 2(1-\gamma)N$$

Euler's constant $\doteq .57721$

Finish: Validation (mathematical)

It is always worthwhile to check your math with your computer.

```
public class QuickCheck
{
   public static void main(String[] args)
   {
      int maxN = Integer.parseInt(args[0]);
      double[] c = new double[maxN+1];
      c[0] = 0;
      for (int N = 1; N <= maxN; N++)
         c[N] = (N+1)*c[N-1]/N + 2;

      for (int N = 10; N <= maxN; N *= 10)
      {
         double approx = 2*N*Math.log(N) - 2*(1-.577215665)*N;
         StdOut.printf("%10d %15.2f %15.2f\n", N, c[N], approx);
      }
   }
}
```

$$NC_N = (N+1)C_{N-1} + 2N$$

$$2N \ln N - 2(1-\gamma)N$$

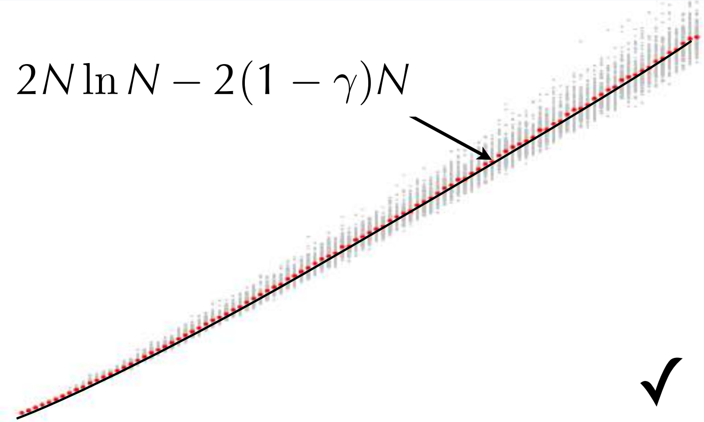
```
% java QuickCheck 1000000
        10           44.44           37.60
       100          847.85          836.48
      1000        12985.91        12969.94
     10000       175771.70       175751.12
    100000      2218053.41      2218028.23
   1000000     26785482.23     26785452.45
```

Finish: Validation (checking the model)

It is always worthwhile to use your computer to check your model.

Example: Mean number of compares used by Quicksort for randomly ordered distinct keys is $2N \ln N - 2(1-\gamma)N$

Experiment: Run code for randomly ordered distinct keys, count compares

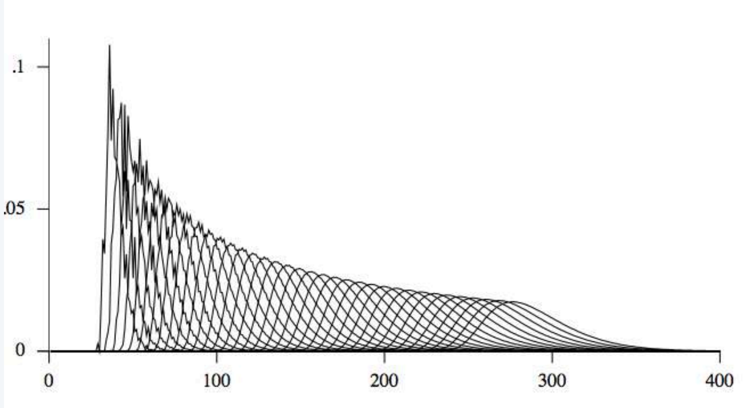


1000 trials for each *N*
one grey dot for each trial
red dot: average for each *N*

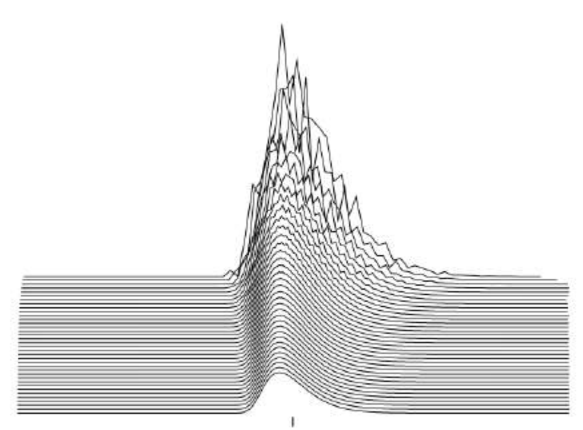
Observation: May be interested in distribution of costs

Quicksort compares: limiting distribution is not "normal"

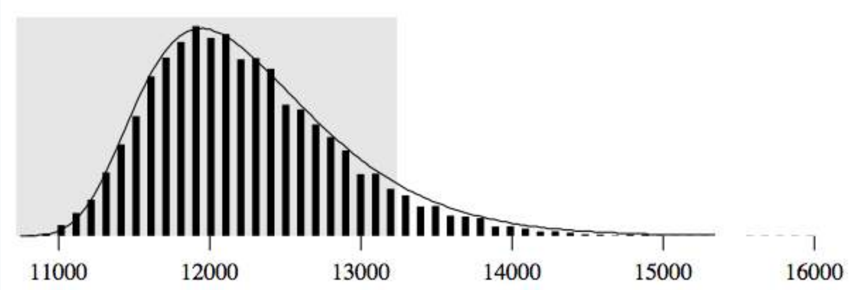
see "Approximating the Limiting Quicksort Distribution." by Fill and Janson (RSA 2001).



exact distribution
(from recurrence)
for small N



centered
on mean



empirical
validation
N = 1000

Bottom line:

- A great deal is known about the performance of Quicksort.
- AofA leads to intriguing new research problems.

Easy method to predict (approximate) performance

Hypothesis: Running time of Quicksort is ~$aN \ln N$.

Experiment.

- Run for input size N. Observe running time.
- [Could solve for a.]
- Predict time for $10N$ to increase by a factor of

$$\frac{a(10N)\ln(10N)}{aN\ln N} = 10 + \frac{\overset{10}{\ln 10}}{\ln N} = 10 + \frac{\overset{10}{1}}{\log_{10} N}$$

Example:

- Run quicksort 100 times for N = 100,000: Elapsed time: 4 seconds.
- Predict running time of 4 x 10.2 = 40.8 seconds for N = 1M.
- Observe running time of 41 seconds for N = 1M
- Confidently predict running time of 41 x 1000.5 = 11.4 hours for N = 1B.

12

48

48

48 x (70/6) x (80/7) x (90/8)
= 20 hours

Note: Best to also have accurate mathematical model. Why?

Validate-refine-analyze cycle

It is always worthwhile to validate your model in applications.

Quicksort: Validation ongoing for 50 years!

Example 1 (late 1970s): Sorting on the CRAY-1.

- Application: cryptography.
- Need to "sort the memory" 1M pseudo-random 64-bit words.
- Bottom line: analysis could predict running time to within 10^{-6} seconds.

as many times as possible!

Example 2 (1990s): UNIX system sort.

- Application: general-purpose.
- User app involving files with only a few distinct values performed poorly.
- Refinements: 3-way partitioning, 3-way string quicksort (see Algs4).
- Refined models (not simple): research ongoing.

see "The number of symbol comparisons in QuickSort and QuickSelect." by Vallee, Clement, Fill, and Flajolet (ICALP 2009).

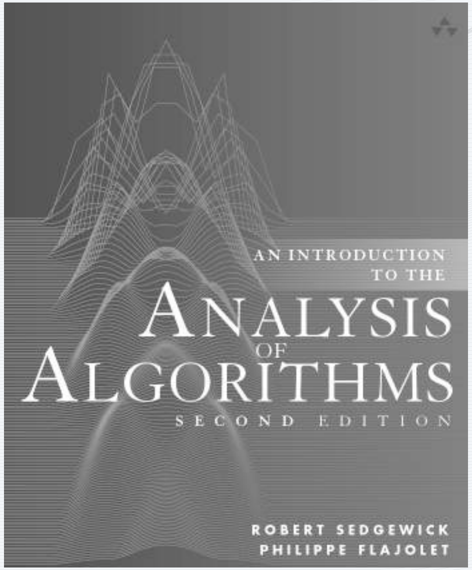
Example 3 (2010s): Sorting for networking.

- Application: sort ~1B records ~1K characters each.
- Need to beat the competition or go out of business.
- Refinement: adapt to long stretches of equal chars (avoid excessive caching)

Double happiness

"People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically."

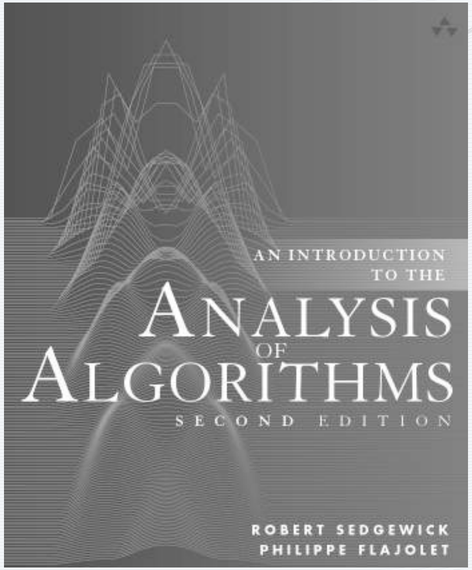
— D. E. Knuth (1995)



http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- Resources



http://aofa.cs.princeton.edu

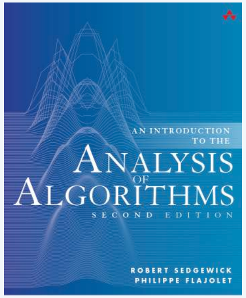
1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- **Resources**

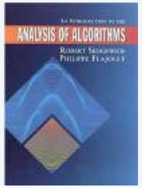
Books

are the prime resources associated with this course.

Main text (2013)

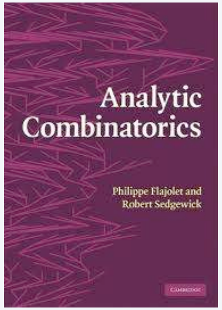


First edition (1995)



Reference for Algorithms

Text for Part II



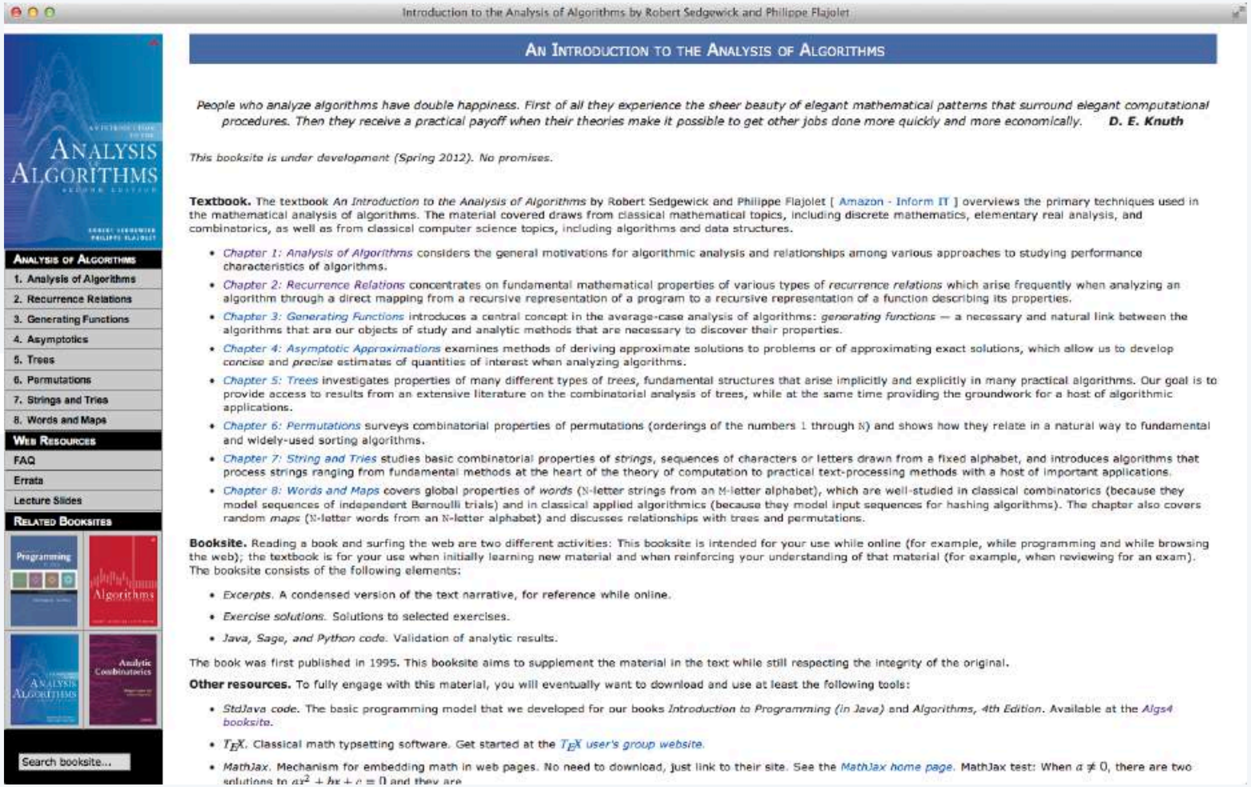
Reference for Java

Reading the books is the best way to develop understanding.

Booksites

are web resources associated with the books.

http://aofa.cs.princeton.edu


Introduction to the Analysis of Algorithms by Robert Sedgewick and Philippe Flajolet

AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically. **D. E. Knuth**

This booksite is under development (Spring 2012). No promises.

Textbook. The textbook *An Introduction to the Analysis of Algorithms* by Robert Sedgewick and Philippe Flajolet [Amazon · Inform IT] overviews the primary techniques used in the mathematical analysis of algorithms. The material covered draws from classical mathematical topics, including discrete mathematics, elementary real analysis, and combinatorics, as well as from classical computer science topics, including algorithms and data structures.

- *Chapter 1: Analysis of Algorithms* considers the general motivations for algorithmic analysis and relationships among various approaches to studying performance characteristics of algorithms.
- *Chapter 2: Recurrence Relations* concentrates on fundamental mathematical properties of various types of *recurrence relations* which arise frequently when analyzing an algorithm through a direct mapping from a recursive representation of a program to a recursive representation of a function describing its properties.
- *Chapter 3: Generating Functions* introduces a central concept in the average-case analysis of algorithms: *generating functions* — a necessary and natural link between the algorithms that are our objects of study and analytic methods that are necessary to discover their properties.
- *Chapter 4: Asymptotic Approximations* examines methods of deriving approximate solutions to problems or of approximating exact solutions, which allow us to develop *concise* and *precise* estimates of quantities of interest when analyzing algorithms.
- *Chapter 5: Trees* investigates properties of many different types of *trees*, fundamental structures that arise implicitly and explicitly in many practical algorithms. Our goal is to provide access to results from an extensive literature on the combinatorial analysis of trees, while at the same time providing the groundwork for a host of algorithmic applications.
- *Chapter 6: Permutations* surveys combinatorial properties of permutations (orderings of the numbers 1 through N) and shows how they relate in a natural way to fundamental and widely-used sorting algorithms.
- *Chapter 7: String and Tries* studies basic combinatorial properties of *strings*, sequences of characters or letters drawn from a fixed alphabet, and introduces algorithms that process strings ranging from fundamental methods at the heart of the theory of computation to practical text-processing methods with a host of important applications.
- *Chapter 8: Words and Maps* covers global properties of *words* (N-letter strings from an M-letter alphabet), which are well-studied in classical combinatorics (because they model sequences of independent Bernoulli trials) and in classical applied algorithmics (because they model input sequences for hashing algorithms). The chapter also covers random *maps* (N-letter words from an N-letter alphabet) and discusses relationships with trees and permutations.

Booksite. Reading a book and surfing the web are two different activities: This booksite is intended for your use while online (for example, while programming and while browsing the web); the textbook is for your use when initially learning new material and when reinforcing your understanding of that material (for example, when reviewing for an exam). The booksite consists of the following elements:

- *Excerpts.* A condensed version of the text narrative, for reference while online.
- *Exercise solutions.* Solutions to selected exercises.
- *Java, Sage, and Python code.* Validation of analytic results.

The book was first published in 1995. This booksite aims to supplement the material in the text while still respecting the integrity of the original.

Other resources. To fully engage with this material, you will eventually want to download and use at least the following tools:

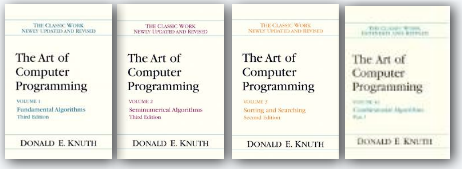
- *StdJava code.* The basic programming model that we developed for our books *Introduction to Programming (in Java)* and *Algorithms, 4th Edition*. Available at the Algs4 booksite.
- *TeX.* Classical math typesetting software. Get started at the TeX user's group website.
- *MathJax.* Mechanism for embedding math in web pages. No need to download, just link to their site. See the MathJax home page. MathJax test: When $a \neq 0$, there are two solutions to $ax^2 + bx + c = 0$ and they are

Surf the booksite to search for information, code, and data.

Extensive original research

is the basis for the material in this course.

Knuth's collected works



Flajolet's collected works

collected works
to appear 2014
Cambridge U. Press

research papers and books by hundreds of others

A prime goal of this course: *make this work accessible to you.*

20,000+ pages of material (!)

More resources

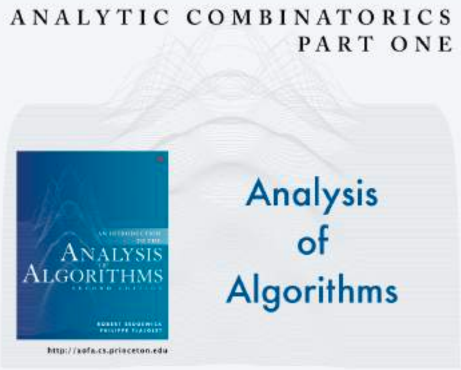
Math typesetting

Symbolic math

Web references

Introduce, read, discuss

1. We introduce topics in lecture.



2. You read the book and do assignments before the next lecture.

Exercise 1.14 Follow through the steps above to solve the recurrence

$$A_N = 1 + \frac{2}{N} \sum_{1 \le j \le N} A_{j-1} \qquad \text{for } N > 0.$$

3. We discuss reading and exercises online. [No assessments.]

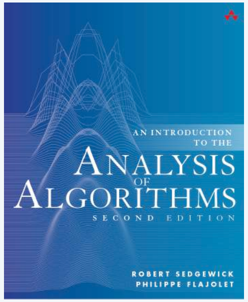
The main resource in this class is YOU!

Goal: For you to *learn* quite a few things that you do not now know.

Exercises 1.14 and 1.15

How many recursive calls in Quicksort?

How many exchanges?



Exercise 1.14 Follow through the steps above to solve the recurrence

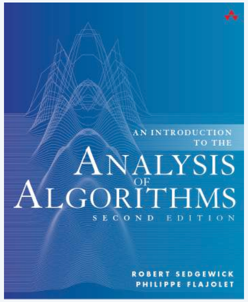
$$A_N = 1 + \frac{2}{N} \sum_{1 \le j \le N} A_{j-1} \qquad \text{for } N > 0.$$

Exercise 1.15 Show that the average number of exchanges used during the first partitioning stage (before the pointers cross) is $(N-2)/6$. (Thus, by linearity of the recurrences, $B_N = \frac{1}{6}C_N - \frac{1}{2}A_N$.)

Exercises 1.17 and 1.18

Switch to insertion sort for small subarrays.

What choice of the threshold minimizes the number of compares?



Exercise 1.17 If we change the first line in the quicksort implementation above to

```
if r-l<=M then insertionsort(l,r) else
```

(see §7.6) then the total number of compares to sort N elements is described by the recurrence

$$C_N = \begin{cases} N+1+\dfrac{1}{N}\displaystyle\sum_{1\le j\le N}(C_{j-1}+C_{N-j}) & \text{for } N > M; \\ \frac{1}{4}N(N-1) & \text{for } N \le M \end{cases}$$

Solve this exactly as in the proof of Theorem 1.3.

Exercise 1.18 Ignoring small terms (those significantly less than N) in the answer to the previous exercise, find a function $f(M)$ so that the number of compares is approximately

$$2N\ln N + f(M)N.$$

Plot the function $f(M)$, and find the value of M that minimizes the function.

Assignments for next lecture

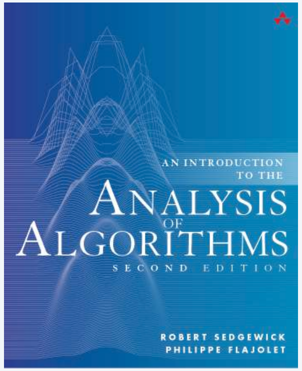
1. Surf booksites
   - http://aofa.cs.princeton.edu
   - http://algs4.cs.princeton.edu

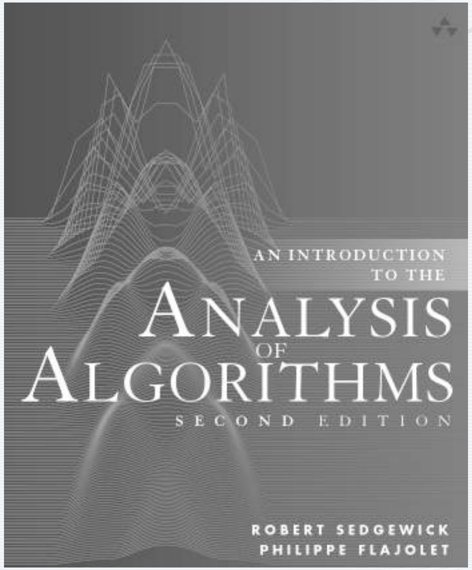
2. Start learning to use software.
   - StdJava (from Algs4 booksite)
   - TeX (optional: .html/MathJax)

3. Download Quicksort and predict performance on your computer.

4. Read pages 1-39 in text.

5. Write up solutions to Exercises 1.14, 1.15, 1.17, and 1.18.



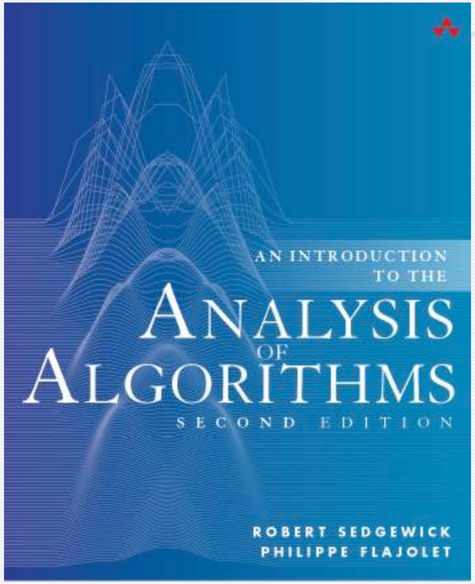


http://aofa.cs.princeton.edu

1. Analysis of Algorithms

- History and motivation
- A scientific approach
- Example: Quicksort
- **Resources**

ANALYTIC COMBINATORICS
PART ONE



http://aofa.cs.princeton.edu

1. Analysis of Algorithms