Algorithms: Theory, Design and Implementation

ANALYSIS OF ALGORITHMS

- *Introduction*
- *Observations via exercises*
- *Mathematical models and theory (optional)*
- *Order-of-growth classifications*

RECAP OF PREVIOUS WEEK

- We already studied various problems (e.g., connectivity, multiplication of integer numbers).
- We elaborated various approaches towards algorithmic solution of the problem.
- We discussed the Karatsumba algorithm in the lecture.

THIS WEEK (LW2)

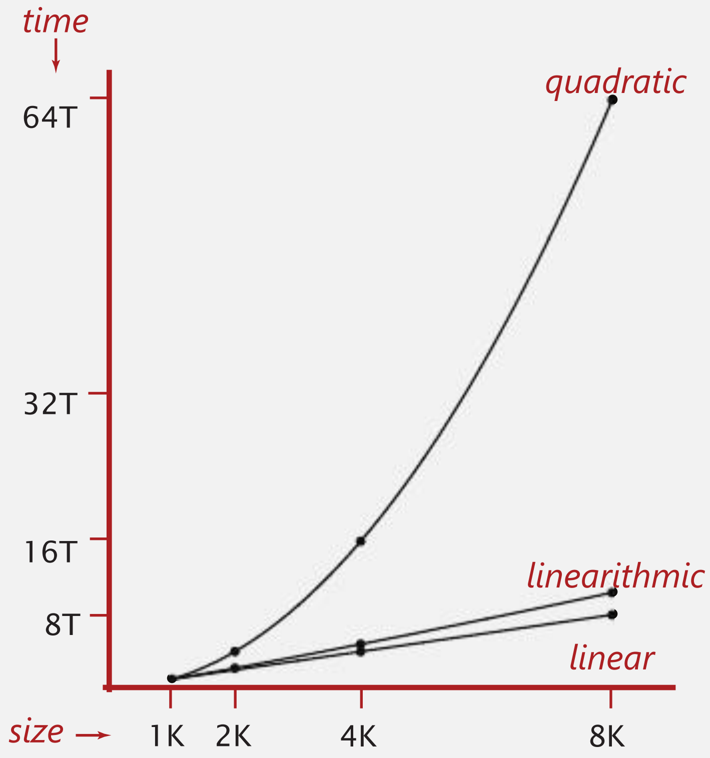
- We will learn what it means to critically compare, evaluate and contrast various algorithmic approaches, designs and implementations, in terms of:
  - Time being spent to computationally resolve the problem
  - Scalability when it comes to operate with different input data sizes
  - Practicality when it comes to keeping things as they are due to their simplicity
- We will start getting an idea of what is considered as **Brute Force strategy** in algorithmic design and implementation

Some more practical algorithmic successes

Discrete Fourier Transformation.

Friedrich Gauss
1805

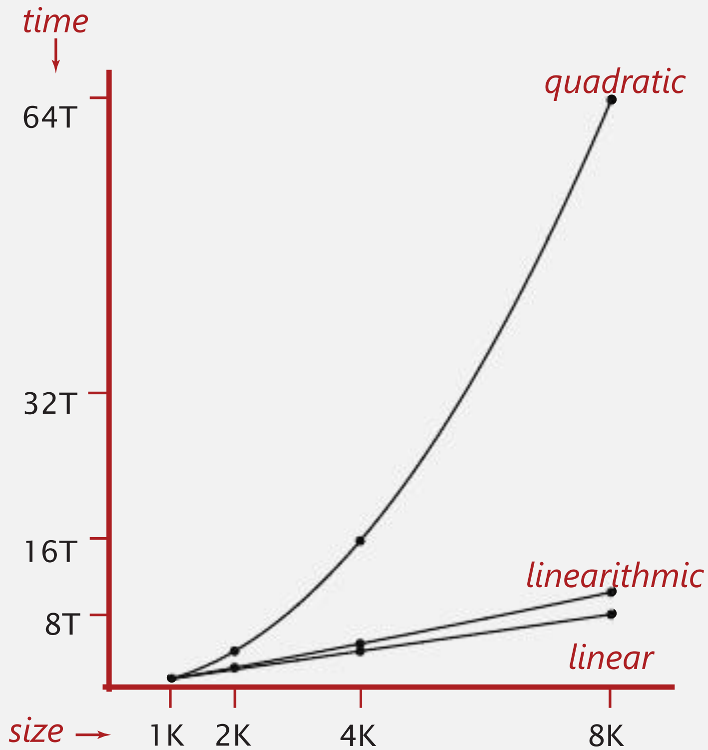
- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ….
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, enables new technology.

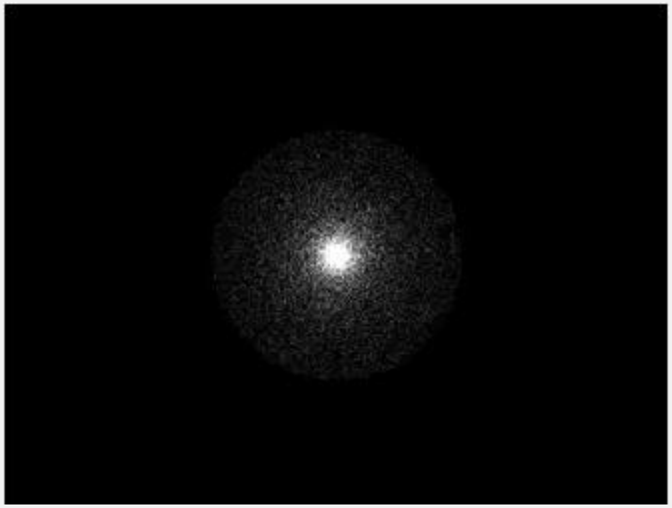


Some algorithmic successes

N-body simulation.

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





Scientific method applied to analysis of algorithms

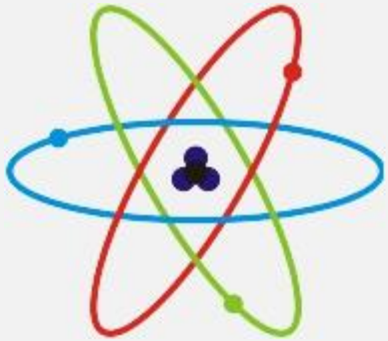
A framework for predicting performance and comparing algorithms.

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.

Analysis of Algorithms

- *Introduction*
- ***Observations via exercises***
- *Mathematical models*
- *Order-of-growth classifications*

Example: 3-SUM

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

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

Context. Deeply related to problems in computational geometry.

3-SUM: brute force algorithm

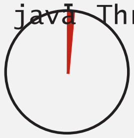
```
public  class  ThreeSum
{
        public  static  int  count(int[]  a)
        {
                int  N  =  a.length;
                int  count  =  0;
                // Check each triple of indices (i, j, k)
                //  We can restrict to i < j < k: re-ordering gives the same sum
                for (int i = 0; i < N; i++)
                        for (int j = i+1; j < N; j++)
                                for (int k = j+1; k < N; k++)
                                        // Ignore possible overflow for simplicity
                                         if (a[i] + a[j] + a[k] == 0)
                                                count++;
        }
        return count;
}
```

Measuring the running time

Q. How to time a program?

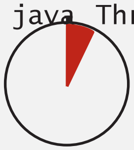
A. Manual.

```
% java ThreeSum 1Kints.txt
```



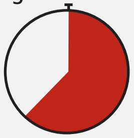
tick tick tick

```
70
% java ThreeSum 2Kints.txt
```



tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick

```
528
% java ThreeSum 4Kints.txt
```



tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick

```
4039
```

Observing the running time of a program

Measuring the running time

Q. How to time a program?

A. Automatic.

`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)
{
   int[] a = In.readInts(args[0]);
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
}
```

client code

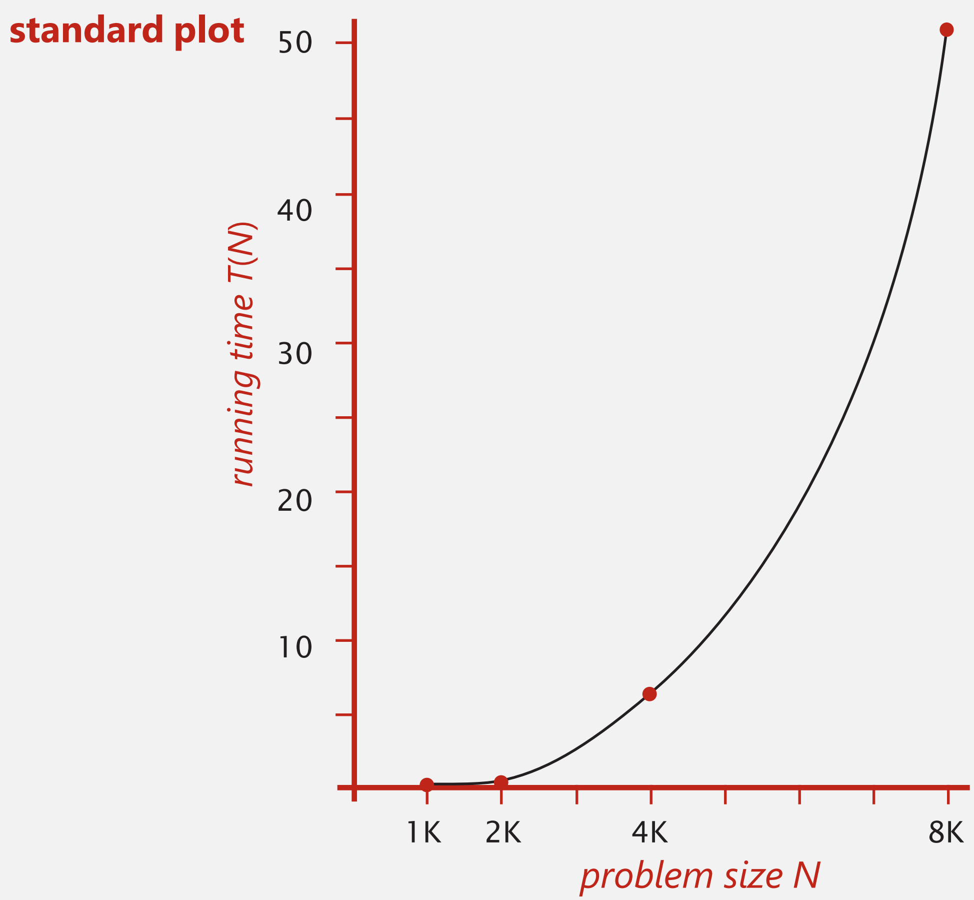
Empirical analysis

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

N	time (seconds) †
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

Data analysis

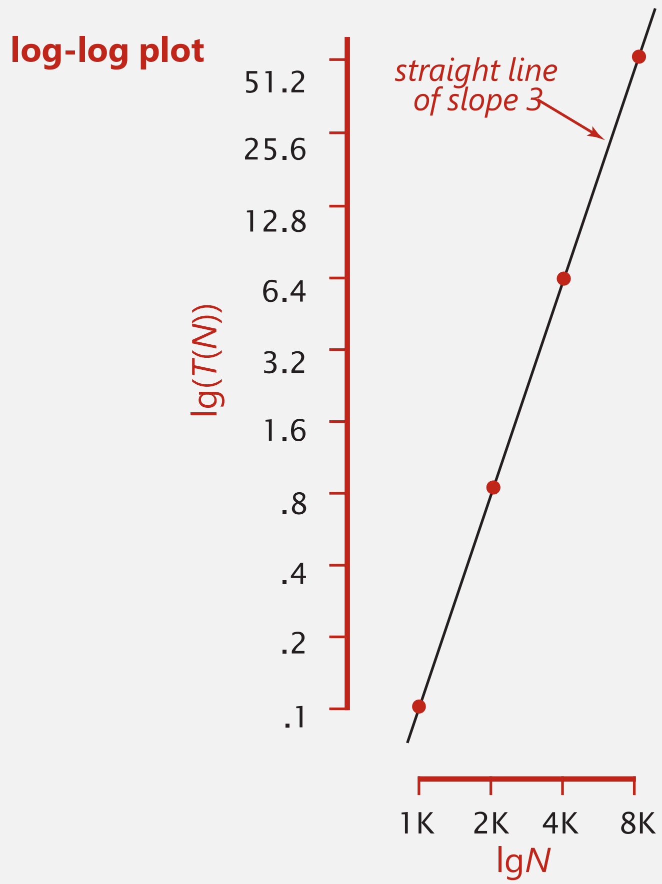
Standard plot. Plot running time $T(N)$ vs. input size N.



Analysis of experimental data (the running time ofThreeSum)

Data analysis

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



$$\lg(T(N)) = b \lg N + c$$
$$b = 2.999$$
$$c = -33.2103$$

$$T(N) = a\, N^{b}, \textit{ where } a = 2^{c}$$

power law

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.

slope

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

"order of growth" of running time is about N power 3.... [stay tuned]

Predictions.

- 51.0 seconds for $N = 8{,}000$.
- 408.1 seconds for $N = 16{,}000$.

Observations.

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

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

Run program, doubling the size of the input.

N	time (seconds) †	ratio	lg ratio
250	0.0		–
500	0.0		
1,000	0.1		
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0
8,000	51.1	8.0	3.0

seems to converge to a constant b ≈ 3

3-SUM: sorting-based algorithm

```
public  class  ThreeSum
{
        public  static  int  count(int[]  a)
        {
                // First, sort the array in ascending order
                sort(a);
                int  N  =  a.length;
                int  count  =  0;
                // For each pair of indices (i, j):
                // Check if the array contains -a[i]-a[j]
                // Using binary search, this only takes logarithmic time
                for (int i = 0; i < N; i++)
                        for (int j = i+1; j < N; j++)
                                if (binarySearch (a, -a[i]-a[j]))
                                        count++;
        }
        return count;
}
```

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth $\rightarrow$ faster in practice.

How did we improve it?

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

ThreeSum.java

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

ThreeSumDeluxe.java

Answer. Sorting + Binary Search [stay tuned….].

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent b in power law

System dependent effects.

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

determines constant a in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.

e.g., can run huge number of experiments

Analysis of Algorithms

- *introduction*
- *observations*
- *mathematical models (optional)*
- ***order-of-growth classifications***

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

TIME AND SPACE COMPLEXITY

For a deeper understanding of asymptotics, Big-O notation and algorithmic performance analysis, please check the uploaded transcript and videos, as well as recommendation from reading list.