CSU22011: ALGORITHMS AND DATA STRUCTURES I

Lecture 2: Analysis of Algorithms

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

Programs = Algorithms + Data structures.

- → Must be correct
- → Must be efficient

LAST LECTURE

Programs = Algorithms + Data structures.

- → Must be correct
- → Must be efficient
  - → What is efficiency?
  - → Analysis of Algorithms: How to evaluate efficiency?

LAST LECTURE

Programs = Algorithms + Data structures.

- → Must be correct
- → Must be efficient
  - → What is efficiency?
  - → Analysis of Algorithms: How to evaluate efficiency?
  - → Established measure of efficiency (from now on **performance**): how the program **scales** to larger inputs. Measure the effect of doubling the input size to
    - → the **running time** of the algorithm
    - → the **memory footprint** of the algorithm

LAST LECTURE

Programs = Algorithms + Data structures.

- → Must be correct
- → Must be efficient
  - → What is efficiency?
  - → Analysis of Algorithms: How to evaluate efficiency?
  - → Established measure of efficiency (from now on **performance**): how the program **scales** to larger inputs. Measure the effect of doubling the input size to
    - → the **running time** of the algorithm
    - → the **memory footprint** of the algorithm
  - → We want to be able to analyse algorithms w.r.t. their performance.

WHY ANALYSE ALGORITHMS?

- **Good programmer:** to predict the performance of our programs.
- **Good client:** to choose between alternative algorithms/implementations.
- **Good manager:** to provide guarantees to clients / avoid client complaints.
- **Good scientist:** to understand the nature of computing.

Linear Search: simple search in an array:

```
boolean linearSearch1(int[] ar, int s) {
  boolean found = false;
  for (int i = 0; i < ar.length; i++) {
    if (ar[i] == s) found = true;
  }
  return found;
}
```

How can we evaluate how well the running time of this algorithm scales to larger inputs?

[1] We will see a number of approaches how to evaluate running times

EXAMPLE: LINEAR SEARCH

Linear Search: simple search in an array:

```
boolean linearSearch1(int[] ar, int s) {
  boolean found = false;
  for (int i = 0; i < ar.length; i++) {
    if (ar[i] == s) found = true;
  }
  return found;
}
```

How can we evaluate how well the running time of this algorithm scales to larger inputs?

→ Evaluate[1] its running time for various sizes of array `ar`. Then calculate the rate of growth of the running time with respect to the input size (stay tuned).

[1]We will see a number of approaches how to evaluate running times

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {
  for (int i = 0; i < ar.length; i++) {
    if (ar[i] == s) return true;
  }
  return false;
}
```

Consider inputs arrays of size 100. Is the running time the same?

EXAMPLE: LINEAR SEARCH 2

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {
  for (int i = 0; i < ar.length; i++) {
    if (ar[i] == s) return true;
  }
  return false;
}
```

Consider inputs arrays of size 100. Is the running time the same?

NO! Even for the same input size, some inputs will require more time than others.

- best case inputs; e.g: `ar=[1,2,...,100], s = 1`
- worst case inputs; e.g: `ar=[1,2,...,100], s = 101`
- everything in between (sometimes we talk about average case)

EXAMPLE: LINEAR SEARCH 2

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {
  for (int i = 0; i < ar.length; i++) {
    if (ar[i] == s) return true;
  }
  return false;
}
```

Consider inputs arrays of size 100. Is the running time the same?

NO! Even for the same input size, some inputs will require more time than others.

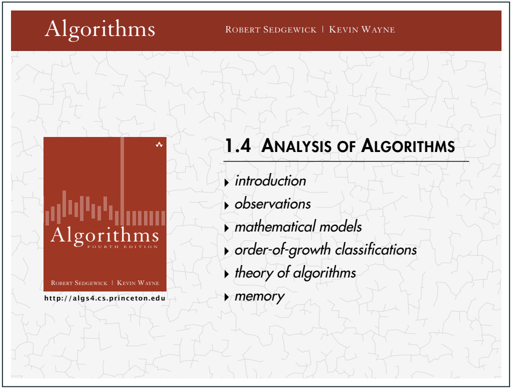
- best case inputs; e.g: `ar=[1,2,...,100], s = 1`
- worst case inputs; e.g: `ar=[1,2,...,100], s = 101`
- everything in between (sometimes we talk about average case)

In this module (and usually in programming practice) we focus on the **worst-case inputs**.

In the course of the next lectures we will see the following methods for evaluating how an algorithm's running time scales:

1. **Scientific method**: measure running times through experiments
2. **Mathematical methods**: consider a convenient model of computation and:
   - → sum up the number of program steps for **worst-case** inputs of size N (N is a variable).
   - → calculate the number of program steps for **worst-case** inputs of size N, when N nears infinity.
     This is the asymptotic (worst-case) running time — notation big-O, Ω, Θ.

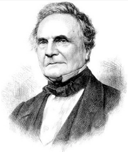
TODAY

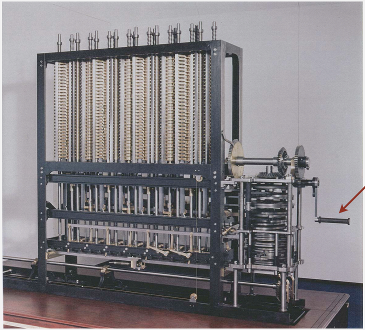


- → Parts from S&W 1.4
- → Estimate the performance of algorithms by
  - → Experiments & Observations
  - → Precise Mathematical Calculations

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 (1864)*





how many times do you have to turn the crank?

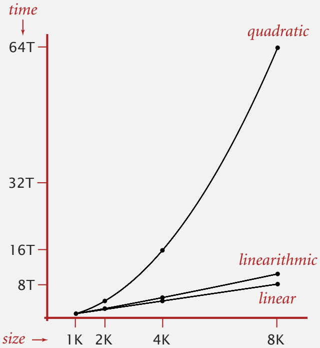
Analytic Engine

Some 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.

Friedrich Gauss
1805

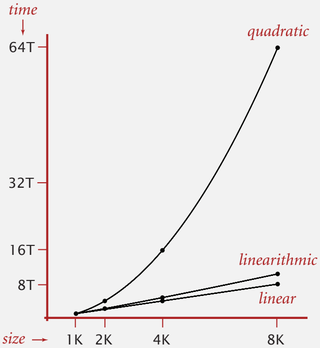


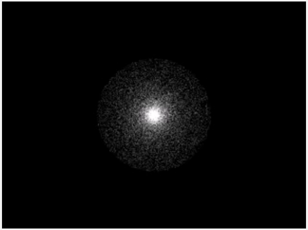
Some algorithmic successes

N-body simulation.

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

Andrew Appel
PU '81





The challenge

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

Insight. [Knuth 1970s] Use scientific method to understand performance.

Scientific method applied to analysis of algorithms

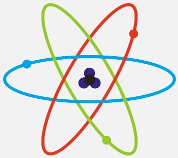
A framework for predicting performance and comparing algorithms.

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.

Scientific Approach:

Evaluating Performance by **Experiments**

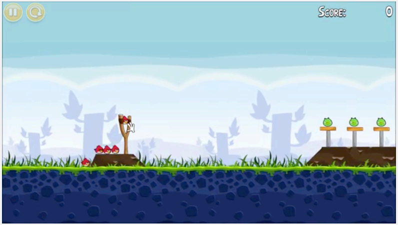
Example: 3-SUM

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;
      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)
                  count++;
      return count;
   }

   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int[] a = in.readAllInts();
      StdOut.println(count(a));
   }
}
```

check each triple

for simplicity, ignore integer overflow

QUESTION

Suppose we care only about 100-element arrays.

Q. What is a worst-case input for `ThreeSum`?

```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int count = 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)
                  count++;
      return count;
   }
```

QUESTION

Suppose we care only about 100-element arrays.

Q. What is a worst-case input for `ThreeSum`?

A. They are all worst case inputs. For-loops always run to the end.

```
public class ThreeSum
{
   public static int count(int[] a)
   {
      int N = a.length;
      int count = 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)
                  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
```

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

Empirical analysis

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

```
%
```

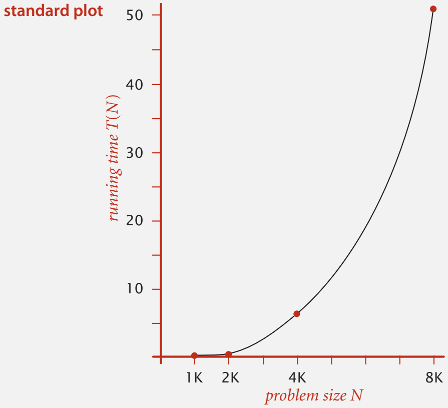
Empirical analysis

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	?

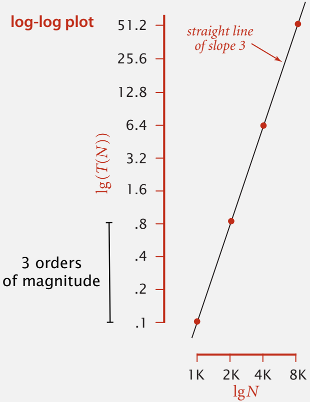
Data analysis

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



Data analysis

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



$$\lg(T(N)) = b \lg N + c$$

$$b = 2.999$$

$$c = -33.2103$$

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

power law

Regression. Fit straight line through data points: $a N^{b}$.

slope

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

Try out the scientific analysis through experiments:

`https://docs.google.com/spreadsheets/d/1WnihyK6g1pYdcT2ndZOqNNRkTitXkWKnOrTgCnM-bw8/edit?usp=sharing`

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^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	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0
8,000	51.1	8.0	3.0

$$\frac{T(2N)}{T(N)} = \frac{a(2N)^b}{aN^b} = 2^b$$

lg (6.4 / 0.8) = 3.0

seems to converge to a constant b ≈ 3

Hypothesis. Running time is about $a N^b$ with b = lg ratio.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

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

Q. How to estimate a (assuming we know b) ?

A. Run the program (for a sufficient large value of N) and solve for a.

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

$$51.1 = a \times 8000^3$$

$$\Rightarrow \quad a = 0.998 \times 10^{-10}$$

Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.

almost identical hypothesis
to one obtained via linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
- Input data.

determines exponent in power law

System dependent effects.

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

determines constant 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

This was the scientific approach to algorithm analysis.

In the mathematical approach we do **calculations** instead of experiments.