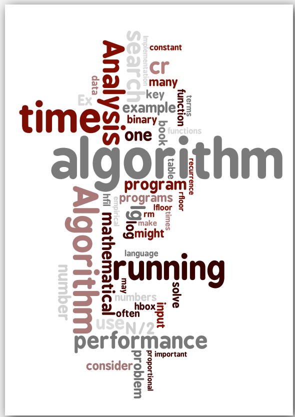
1.4 Analysis of Algorithms



- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory

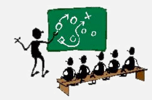
Cast of characters

Programmer needs to develop a working solution.

Student might play any or all of these roles someday.

Client wants to solve problem efficiently.

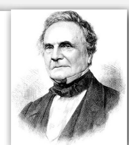
Theoretician wants to understand.

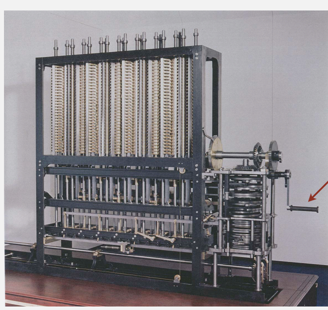


Basic blocking and tackling is sometimes necessary. [this lecture]

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

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

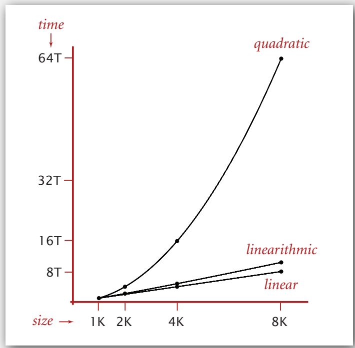
client gets poor performance because programmer did not understand performance characteristics

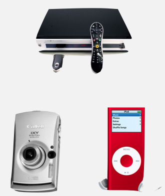
Some algorithmic successes

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



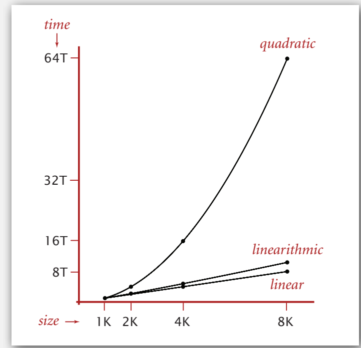


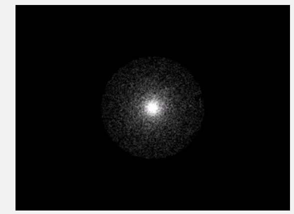
Some algorithmic successes

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?

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

Scientific method applied to analysis of algorithms

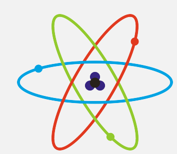
A framework for predicting performance and comparing algorithms.

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.

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory

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)
   {
      int[] a = StdArrayIO.readInt1D();
      StdOut.println(count(a));
   }
}
```

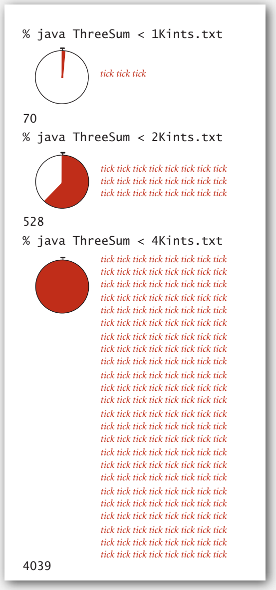
check each triple

we ignore any integer overflow

Measuring the running time

Q. How to time a program?

A. Manual.



Measuring the running time

Q. How to time a program?

A. Automatic.

```
public class Stopwatch

             Stopwatch()          create a new stopwatch

      double elapsedTime()        time since creation (in seconds)
```

```
public static void main(String[] args)
{
   int[] a = StdArrayIO.readInt1D();
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
}
```


Measuring the running time

Q. How to time a program?

A. Automatic.

```
public class Stopwatch

             Stopwatch()          create a new stopwatch

      double elapsedTime()        time since creation (in seconds)
```

```
public class Stopwatch
{
   private final long start = System.currentTimeMillis();

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

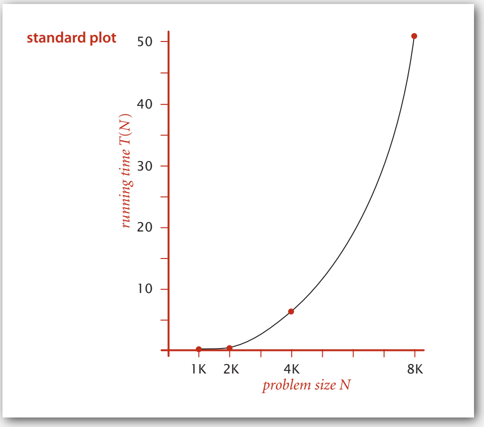

Empirical analysis

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	?

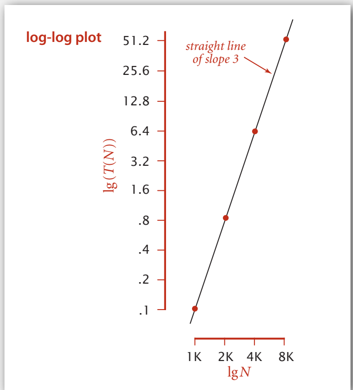
Data analysis

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



Data analysis

Log-log plot. Plot running time vs. input size N using log-log scale.



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

$T(N) = a N^b$, where $a = 2^c$

power law

slope

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.

Prediction and validation (experimental)

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

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!

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 applications, …

helps determines constant a in power law

Bad news. Difficult to get precise measurements.
Good news. Much easier and cheaper than other sciences.

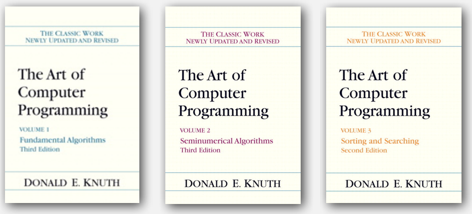
e.g., can run huge number of experiments

- observations
- **mathematical models**
- order-of-growth classifications
- dependencies on inputs
- memory

Mathematical models for running time

Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6 N$
2D array allocation	`new int[N][N]`	$c_7 N^2$
string length	`s.length()`	c_8
substring extraction	`s.substring(N/2, N)`	c_9
string concatenation	`s + t`	$c_{10} N$

Novice mistake. Abusive string concatenation.

Example: 1-sum

Q. How many instructions as a function of input size N ?

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	N + 1
equal to compare	N
array access	N
increment	N to 2 N

Example: 2-sum

Q. How many instructions as a function of input size N?

```
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++;
```

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

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N − 1)
array access	N (N − 1)
increment	N to 2 N

tedious to count exactly

Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```
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++;
```

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

operation	frequency
variable declaration	N + 2
assignment statement	N + 2
less than compare	½ (N + 1) (N + 2)
equal to compare	½ N (N − 1)
array access	N (N − 1)
increment	N to 2 N

cost model = array accesses

Simplification 2: tilde notation

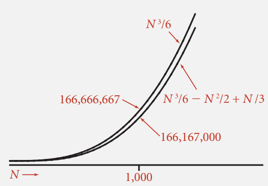
- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

Ex 1. $\frac{1}{6} N^3 + 20 N + 16 \sim \frac{1}{6} N^3$

Ex 2. $\frac{1}{6} N^3 + 100 N^{4/3} + 56 \sim \frac{1}{6} N^3$

Ex 3. $\underbrace{\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N}_{}$ $\sim \frac{1}{6} N^3$

discard lower-order terms
(e.g., N = 1000: 500 thousand vs. 166 million)



Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

Simplification 2: tilde notation

- Estimate running time (or memory) 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

operation	frequency	tilde notation
variable declaration	N + 2	~ N
assignment statement	N + 2	~ N
less than compare	½ (N + 1) (N + 2)	~ ½ N^2
equal to compare	½ N (N − 1)	~ ½ N^2
array access	N (N − 1)	~ N^2
increment	N to 2 N	~ N to ~ 2 N

Example: 2-sum

Q. Approximately how many array accesses as a function of input size N ?

```
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++;
```

"inner loop"

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

A. $\sim N^2$ array accesses.

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Example: 3-sum

Q. Approximately how many array accesses as a function of input size N ?

```
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++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6} N^3$$

A. $\sim ½\, N^3$ array accesses.

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take COS 340.

A2. Replace the sum with an integral, and use calculus!

Ex 1. $1 + 2 + \ldots + N$.
$$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x \, dx \sim \frac{1}{2} N^2$$

Ex 2. $1 + 1/2 + 1/3 + \ldots + 1/N$.
$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} dx = \ln N$$

Ex 3. 3-sum triple loop.
$$\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3$$

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,
- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

A = array access
B = integer add
C = integer compare
D = increment
E = variable assignment

frequencies (depend on algorithm, input)

Bottom line. We use approximate models in this course: $T(N) \sim c\, N^3$.

A reasonable model

The running time of your program is $\sim a\,N^b\,(\lg N)^c$

- Specific models of this form are known for many algorithms (stay tuned).
- General laws of this form are known in many circumstances.
  (Interested? Take courses in combinatorics and complex analysis)

Notes

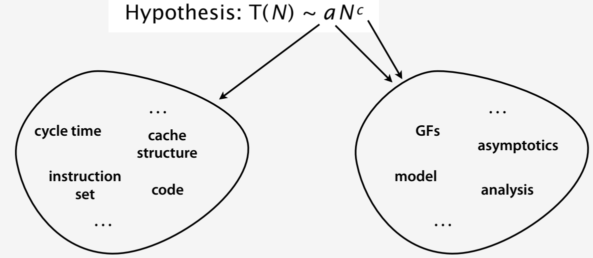
- The existence of the constant a is more significant than its value.
- We often drop the constant and refer to the order of growth.
- The small set of functions
  $1,\ \log N,\ N,\ N \log N,\ N^2$, and N^3
  suffices to describe order of growth of running time of typical algorithms.
- Some algorithms take exponential ($\sim d^N$) time (we consider such algorithms in the last few lectures)

Computing the constants (the hard way)

Knuth showed that it is possible in principle to precisely predict running time

- develop a mathematical model for the frequency of execution of each instruction in the program
- determine the time required to execute each instruction
- multiply and sum

Hypothesis: $T(N) \sim a\,N^c$



machine-dependent part of the constants (harder to determine now than in the 1970s)

algorithm and model-dependent part of the constants (easier to determine now than in the 1970s)

Computing the constants (easy way)

Run the program!

Hypothesis: $T(N) \sim a\,N^b$

Note: log factor is more difficult

1. Implement the program
2. Compute $T(N_0)$ and $T(2N_0)$ by running it
3. Calculate b as follows:

$$\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^b}{a N_0^b} = 2^b$$

$\lg(T(2N_0)/T(N_0)) \rightarrow b$ as N_0 grows

4. Calculate a as follows:

$T(N_0)/N_0^b \rightarrow a$ as N_0 grows

N	time	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

$b \approx 3$

$a \approx 51.1/8000^3 \approx 9.98 \times 10^{-11}$

Predicting performance (the easy way)

Don't bother computing the constants!

Hypothesis: $T(N) \sim a\,N^b$

1. Implement the program
2. Run it for $N_0,\ 2N_0,\ 4N_0,\ 8N_0, \ldots$
3. Ratio of running times approaches 2^b

$$\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^b}{a N_0^b} = 2^b$$

4. Multiply by ratio 2^b to predict next value

N	time	ratio
250	0.0	
500	0.0	4.8
1,000	0.1	6.9
2,000	0.8	7.7
4,000	6.4	8.0
8,000	51.1	8.0
16,000	409.3	

predicted value → 408.8 = 51.1 * 8.0

predicted order of growth N^3 since lg 8 = 3

Plenty of caveats, but provides a basis for predicting program performance

War story (from COS 126)

Q. How long does this program take as a function of N ?

```
String s = StdIn.readString();
int N = s.length();
...
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      distance[i][j] = ...
...
```

N	time
1,000	0.11
2,000	0.35
4,000	1.6
8,000	6.5

Jenny ~ $c_1 N^2$ seconds

N	time
250	0.5
500	1.1
1,000	1.9
2,000	3.9

Kenny ~ $c_2 N$ seconds

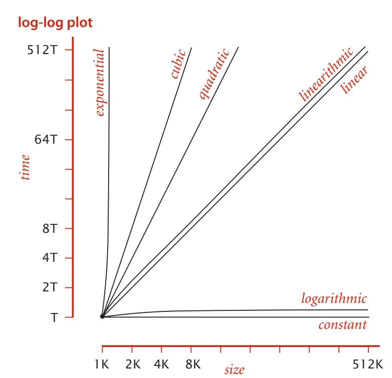
- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory

Common order-of-growth classifications

Good news. the small set of functions

$$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3, \text{ and } 2^N$$

suffices to describe order of growth of the running time of typical algorithms.



Typical orders of growth

Common order-of-growth classifications

growth rate	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.

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo			mid			hi								

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo	mid	hi								

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.

lo = hi

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

mid
return 4

Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946; first bug-free one published in 1962.
- Java bug in `Arrays.binarySearch()` not fixed until 2006.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
  return -1;
}
```

one 3-way compare

Invariant. If `key` appears in the array `a[]`, then `a[lo]` ≤ `key` ≤ `a[hi]`.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ compares to search in a sorted array of size N.

Def. $T(N)$ ≡ # compares to binary search in a sorted subarray of size N.

Binary search recurrence. $T(N) \le \underbrace{T(N/2)}_{\text{left or right half}} + 1$ for $N > 1$, with $T(1) = 1$.

Pf sketch.

$$
\begin{aligned}
T(N) &\le T(N/2) + 1 && \text{given} \\
&\le T(N/4) + 1 + 1 && \text{apply recurrence to first term} \\
&\le T(N/8) + 1 + 1 + 1 && \text{apply recurrence to first term} \\
&\;\;\cdots \\
&\le T(N/N) + 1 + 1 + \ldots + 1 && \text{stop applying, } T(1) = 1 \\
&= 1 + \lg N
\end{aligned}
$$

An $N^2 \log N$ algorithm for 3-sum

Step 1. Sort the N numbers.

Step 2. For each pair of numbers `a[i]` and `a[j]`, binary search for `-(a[i] + a[j])`.

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

input
```
  30 -40 -20 -10 40  0 10  5
```

sort
```
 -40 -20 -10   0  5 10 30 40
```

binary search
```
(-40, -20)    60
(-40, -10)    30
(-40,   0)    40
(-40,   5)    35
(-40,  10)    30
…
(-40,  40)     0
…
(-10,   0)    10
…
(-20,  10)    10
…
( 10,  30)   -40
( 10,  40)   -50
( 30,  40)   -70
```

only count if a[i] < a[j] < a[k] to avoid double counting

Comparing programs

Hypothesis. The $N^2 \log N$ three-sum algorithm is significantly faster in practice than the brute-force N^3 one.

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

Bottom line. Typically, better order of growth ⇒ faster in practice.

Types of analyses

Best case. Lower bound on cost.
- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.
- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.
- Need a model for "random" input.
- Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3 sum.
Best: $\sim \frac{1}{2} N^3$
Average: $\sim \frac{1}{2} N^3$
Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.
Best: ~ 1
Average: $\sim \lg N$
Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.
Worst case. Upper bound on cost.
Average case. "Expected" cost.

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

Commonly-used notations

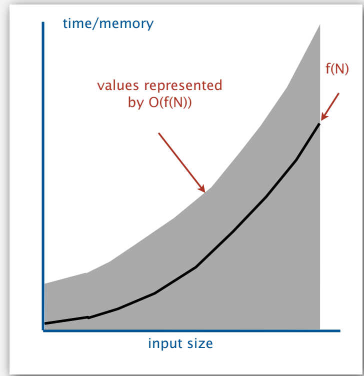
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	$\frac{1}{2}\ N^2$ $10\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\ N^2$ $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

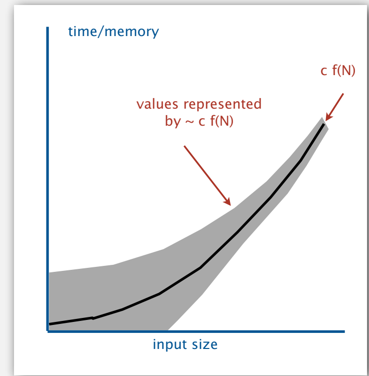
Common mistake. Interpreting big-Oh as an approximate model.

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.

- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).





O-notation considered harmful

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 (stay tuned)

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

- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- **memory**

Typical memory requirements for primitive types in Java

Bit. 0 or 1.
Byte. 8 bits.
Megabyte (MB). 1 million bytes.
Gigabyte (GB). 1 billion bytes.

type	bytes
`boolean`	1
`byte`	1
`char`	2
`int`	4
`float`	4
`long`	8
`double`	8

for primitive types

Typical memory requirements for arrays in Java

Array overhead. 16 bytes.

type	bytes
`char[]`	2N + 16
`int[]`	4N + 16
`double[]`	8N + 16

for one-dimensional arrays

type	bytes
`char[][]`	~ 2 M N
`int[][]`	~ 4 M N
`double[][]`	~ 8 M N

for two-dimensional arrays

Ex. An N-by-N array of doubles consumes ~ $8N^2$ bytes of memory.

Typical memory requirements for objects in Java

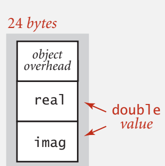
Object overhead. 8 bytes.
Reference. 4 bytes.

Ex 1. A `Complex` object consumes 24 bytes of memory.

```
public class Complex
{
   private double re;
   private double im;
   ...
}
```

8 bytes (object overhead)
8 bytes (double)
8 bytes (double)

24 bytes



Typical memory requirements for objects in Java

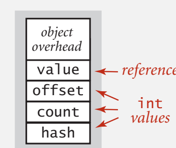
Object overhead. 8 bytes.
Reference. 4 bytes.

Ex 2. A virgin `String` of length N consumes ~ $2N$ bytes of memory.

```
public class String
{
   private int offset;
   private int count;
   private int hash;
   private char[] value;
   ...
}
```

8 bytes (object overhead)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (reference to array)
2N + 16 bytes (char[] array)

2N + 40 bytes



Example

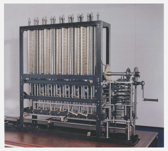
Q. How much memory does `WeightedQuickUnionUF` use as a function of N?

```
public class WeightedQuickUnionUF
{
   private int[] id;
   private int[] sz;

   public WeightedQuickUnionUF(int N)
   {
      id = new int[N];
      sz = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      for (int i = 0; i < N; i++) sz[i] = 1;
   }

   public boolean find(int p, int q)
   { ... }

   public void union(int p, int q)
   { ... }
}
```


Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to make predictions.



Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to explain behavior.

Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.