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.

**Client** wants to solve problem efficiently.

**Theoretician** wants to understand.

**Student** might play any or all of these roles someday.

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)
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
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.
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.
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

Context. Deeply related to problems in computational geometry.
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));
    }
}
Q. How to time a program?
A. Manual.

Measuring the running time

% java ThreeSum < 1Kints.txt
% java ThreeSum < 2Kints.txt
% java ThreeSum < 4Kints.txt
**Measuring the running time**

**Q.** How to time a program?

**A.** Automatic.

```java
public class Stopwatch {
    public Stopwatch() {
        // create a new stopwatch
    }

    public double elapsedTime() {
        // time since creation (in seconds)
    }
}

class ThreeSum {
    public static int count(int[] a) {
        // Some implementation
    }
}

class ClientGcode {
    public static void main(String[] args) {
        int[] a = StdArrayIO.readInt1D();
        Stopwatch stopwatch = new Stopwatch();
        StdOut.println(ThreeSum.count(a));
        double time = stopwatch.elapsedTime();
    }
}
```
Q. How to time a program?
A. Automatic.

```java
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.

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds) †</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>?</td>
</tr>
</tbody>
</table>
Data analysis

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

![Standard plot diagram](image.png)
Data analysis

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

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.

**Predictions.**
- 51.0 seconds for \(N = 8,000\).
- 408.1 seconds for \(N = 16,000\).

**Observations.**

<table>
<thead>
<tr>
<th>N</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>8,000</td>
<td>51.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
<tr>
<td>16,000</td>
<td>410.8</td>
</tr>
</tbody>
</table>

validates hypothesis!
Experimental algorithmics

System independent effects.
• Algorithm.
• Input data.

\[ \text{determines exponent } b \text{ in power law} \]

System dependent effects.
• Hardware: CPU, memory, cache, ...
• Software: compiler, interpreter, garbage collector, ...
• System: operating system, network, other applications, ...

\[ \text{helps determines constant } a \text{ in power law} \]

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

\[ \text{e.g., can run huge number of experiments} \]
observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory
Mathematical models for running time

Total running time: \textit{sum of cost} \times \textit{frequency for all operations}.

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

\textbf{In principle}, accurate mathematical models are available.
## Cost of basic operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer add</td>
<td>a + b</td>
<td>2.1</td>
</tr>
<tr>
<td>integer multiply</td>
<td>a * b</td>
<td>2.4</td>
</tr>
<tr>
<td>integer divide</td>
<td>a / b</td>
<td>5.4</td>
</tr>
<tr>
<td>floating-point add</td>
<td>a + b</td>
<td>4.6</td>
</tr>
<tr>
<td>floating-point multiply</td>
<td>a * b</td>
<td>4.2</td>
</tr>
<tr>
<td>floating-point divide</td>
<td>a / b</td>
<td>13.5</td>
</tr>
<tr>
<td>sine</td>
<td>Math.sin(theta)</td>
<td>91.3</td>
</tr>
<tr>
<td>arctangent</td>
<td>Math.atan2(y, x)</td>
<td>129.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM
### Cost of basic operations

<table>
<thead>
<tr>
<th>operation</th>
<th>example</th>
<th>nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>int a</td>
<td>$c_1$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>a = b</td>
<td>$c_2$</td>
</tr>
<tr>
<td>integer compare</td>
<td>a &lt; b</td>
<td>$c_3$</td>
</tr>
<tr>
<td>array element access</td>
<td>a[i]</td>
<td>$c_4$</td>
</tr>
<tr>
<td>array length</td>
<td>a.length</td>
<td>$c_5$</td>
</tr>
<tr>
<td>1D array allocation</td>
<td>new int[N]</td>
<td>$c_6 \ N$</td>
</tr>
<tr>
<td>2D array allocation</td>
<td>new int[N][N]</td>
<td>$c_7 \ N^2$</td>
</tr>
<tr>
<td>string length</td>
<td>s.length()</td>
<td>$c_8$</td>
</tr>
<tr>
<td>substring extraction</td>
<td>s.substring(N/2, N)</td>
<td>$c_9$</td>
</tr>
<tr>
<td>string concatenation</td>
<td>s + t</td>
<td>$c_{10} \ N$</td>
</tr>
</tbody>
</table>
Example: 1-sum

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

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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>2</td>
</tr>
<tr>
<td>less than compare</td>
<td>$N + 1$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$N$</td>
</tr>
<tr>
<td>array access</td>
<td>$N$</td>
</tr>
<tr>
<td>increment</td>
<td>$N$ to $2N$</td>
</tr>
</tbody>
</table>
Example: 2-sum

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

```c
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++;
```

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
</tr>
<tr>
<td>increment</td>
<td>$N$ to $2N$</td>
</tr>
</tbody>
</table>

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

tedious to count exactly
Simplification 1: cost model

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

```c
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++;
```

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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>N + 2</td>
</tr>
<tr>
<td>assignment statement</td>
<td>N + 2</td>
</tr>
<tr>
<td>less than compare</td>
<td>( \frac{1}{2} (N + 1) (N + 2) )</td>
</tr>
<tr>
<td>equal to compare</td>
<td>( \frac{1}{2} N (N - 1) )</td>
</tr>
<tr>
<td><strong>array access</strong></td>
<td>N (N - 1)</td>
</tr>
<tr>
<td>increment</td>
<td>N to 2 N</td>
</tr>
</tbody>
</table>

The cost model equals the number of 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. $\frac{1}{6} N^3 - \frac{1}{2} N^2 + \frac{1}{3} N \sim \frac{1}{6} N^3$

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

**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

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>tilde notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable declaration</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>assignment statement</td>
<td>$N + 2$</td>
<td>$\sim N$</td>
</tr>
<tr>
<td>less than compare</td>
<td>$\frac{1}{2} (N + 1) (N + 2)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>equal to compare</td>
<td>$\frac{1}{2} N (N - 1)$</td>
<td>$\sim \frac{1}{2} N^2$</td>
</tr>
<tr>
<td>array access</td>
<td>$N (N - 1)$</td>
<td>$\sim N^2$</td>
</tr>
<tr>
<td>increment</td>
<td>$N$ to $2N$</td>
<td>$\sim N$ to $\sim 2N$</td>
</tr>
</tbody>
</table>
Example: 2-sum

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

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

Bottom line. Use cost model and tilde notation to simplify frequency counts.
Q. **Approximately how many array accesses** as a function of input size $N$?

A. $\sim \frac{1}{2} N^3$ 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?
A2. Replace the sum with an integral, and use calculus!

Ex 1. 1 + 2 + … + 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 + … + 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.
\[ \sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz \, dy \, dx \sim \frac{1}{6} N^3 \]
In principle, accurate mathematical models are available.

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

Mathematical models for running time

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

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

$A = \text{array access}$
$B = \text{integer add}$
$C = \text{integer compare}$
$D = \text{increment}$
$E = \text{variable assignment}$

Costs (depend on machine, compiler)
Frequencies (depend on algorithm, input)
A reasonable model

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

- **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 $1, \log N, N, N \log N, N^2, \text{ and } N^3$ 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: \( T(N) \sim a N^c \)

- cycle time
- instruction set
- cache structure
- code

- GFs
- model
- asymptotics
- analysis

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

1. Implement the program

2. Compute $T(N_0)$ and $T(2N_0)$ by running it

3. Calculate $b$ as follows:

\[
\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^b}{aN_0^b} = 2^b
\]

\[
\text{lg}(T(2N_0)/T(N_0)) \rightarrow b \quad \text{as } N_0 \text{ grows}
\]

4. Calculate $a$ as follows:

\[
\frac{T(N_0)}{N_0^b} \rightarrow a \quad \text{as } N_0 \text{ grows}
\]

Note: log factor is more difficult

<table>
<thead>
<tr>
<th>N</th>
<th>time</th>
<th>ratio</th>
<th>lg ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
<td>4.8</td>
<td>2.3</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>6.9</td>
<td>2.8</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>7.7</td>
<td>2.9</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8.0</td>
<td>3.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

$b \approx 3$

$a \approx 51.1 / 8000^3 \approx 9.98 \times 10^{-11}$
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 $N_0$, $2N_0$, $4N_0$, $8N_0$, …

3. Ratio of running times approaches $2^b$

$$\frac{T(2N_0)}{T(N_0)} \sim \frac{a(2N_0)^b}{aN_0^b} = 2^b$$

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>0.0</td>
<td>4.8</td>
</tr>
<tr>
<td>1,000</td>
<td>0.1</td>
<td>6.9</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
<td>7.7</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
<td>8.0</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
<td>8.0</td>
</tr>
<tr>
<td>16,000</td>
<td>409.3</td>
<td></td>
</tr>
</tbody>
</table>

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 $N$?

```java
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] = ...
...
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.11</td>
</tr>
<tr>
<td>2,000</td>
<td>0.35</td>
</tr>
<tr>
<td>4,000</td>
<td>1.6</td>
</tr>
<tr>
<td>8,000</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Jenny ~ $c_1 N^2$ seconds

<table>
<thead>
<tr>
<th>$N$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.5</td>
</tr>
<tr>
<td>500</td>
<td>1.1</td>
</tr>
<tr>
<td>1,000</td>
<td>1.9</td>
</tr>
<tr>
<td>2,000</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Kenny ~ $c_2 N$ seconds
observations
mathematical models
**order-of-growth classifications**
dependencies on inputs
memory
Good news. the small set of functions

1, log \(N\), \(N\), \(N \log N\), \(N^2\), \(N^3\), and \(2^N\)
suffices to describe order of growth of the running time of typical algorithms.
## Common order-of-growth classifications

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>typical code framework</th>
<th>description</th>
<th>example</th>
<th>$T(2N) / T(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>$a = b + c;$</td>
<td>statement</td>
<td>add two numbers</td>
<td>1</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td><code>while (N &gt; 1) { N = N / 2; ... }</code></td>
<td>divide in half</td>
<td>binary search</td>
<td>~ 1</td>
</tr>
<tr>
<td>$N$</td>
<td>linear</td>
<td><code>for (int i = 0; i &lt; N; i++) { ... }</code></td>
<td>loop</td>
<td>find the maximum</td>
<td>2</td>
</tr>
<tr>
<td>$N \log N$</td>
<td>linearithmic</td>
<td>[see mergesort lecture]</td>
<td>divide and conquer</td>
<td>mergesort</td>
<td>~ 2</td>
</tr>
</tbody>
</table>
| $N^2$       | 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**

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N²</td>
<td>hundreds</td>
</tr>
<tr>
<td>N³</td>
<td>hundred</td>
</tr>
<tr>
<td>2^N</td>
<td>20</td>
</tr>
</tbody>
</table>

**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.
Binary search

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

**Successful search.** Binary search for 33.

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>13</th>
<th>14</th>
<th>25</th>
<th>33</th>
<th>43</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

↑ 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.
**Binary search**

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

**Successful search.** Binary search for 33.
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;
}
```

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.** $T(N) \equiv \# \text{ compares to binary search in a sorted subarray of size } N$.

**Binary search recurrence.** $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.

**Pf sketch.**

\[
T(N) \leq T(N/2) + 1 \\
\leq T(N/4) + 1 + 1 \\
\leq T(N/8) + 1 + 1 + 1 \\
\ldots
\leq T(N/N) + 1 + 1 + \ldots + 1 \\
= 1 + \lg N
\]

---

given

apply recurrence to first term

apply recurrence to first term

stop applying, $T(1) = 1$
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.
Comparing programs

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

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.1</td>
</tr>
<tr>
<td>2,000</td>
<td>0.8</td>
</tr>
<tr>
<td>4,000</td>
<td>6.4</td>
</tr>
<tr>
<td>8,000</td>
<td>51.1</td>
</tr>
</tbody>
</table>

ThreeSum.java

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.14</td>
</tr>
<tr>
<td>2,000</td>
<td>0.18</td>
</tr>
<tr>
<td>4,000</td>
<td>0.34</td>
</tr>
<tr>
<td>8,000</td>
<td>0.96</td>
</tr>
<tr>
<td>16,000</td>
<td>3.67</td>
</tr>
<tr>
<td>32,000</td>
<td>14.88</td>
</tr>
<tr>
<td>64,000</td>
<td>59.16</td>
</tr>
</tbody>
</table>

ThreeSumDeluxe.java

Bottom line. Typically, better order of growth $\Rightarrow$ faster in practice.
- observations
- mathematical models
- order-of-growth classifications
- dependencies on inputs
- memory
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: $\sim 1$
- Average: $\sim \log N$
- Worst: $\sim \log N$
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

<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>Tilde</td>
<td>leading term</td>
<td>$\sim 10 N^2$</td>
<td>$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic growth rate</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$10 N^2$ $100 N$ $22 N \log N + 3 N$</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>$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3 N$</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
</tbody>
</table>

for primitive types
Typical memory requirements for arrays in Java

**Array overhead.** 16 bytes.

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>2N + 16</td>
</tr>
<tr>
<td>int[]</td>
<td>4N + 16</td>
</tr>
<tr>
<td>double[]</td>
<td>8N + 16</td>
</tr>
</tbody>
</table>

for one-dimensional arrays

<table>
<thead>
<tr>
<th>type</th>
<th>bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[][]</td>
<td>~2 MN</td>
</tr>
<tr>
<td>int[][]</td>
<td>~4 MN</td>
</tr>
<tr>
<td>double[][]</td>
<td>~8 MN</td>
</tr>
</tbody>
</table>

for two-dimensional arrays

**Ex.** An $N$-by-$N$ array of doubles consumes $\sim 8N^2$ 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.

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

24 bytes

- 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 $\sim 2N$ bytes of memory.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>4</td>
</tr>
<tr>
<td>count</td>
<td>4</td>
</tr>
<tr>
<td>hash</td>
<td>4</td>
</tr>
<tr>
<td>value</td>
<td>$2N + 16$</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>$2N + 40$</td>
</tr>
</tbody>
</table>

**Object overhead:** 8 bytes

**Value:**

<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>C</td>
</tr>
<tr>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>G</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>T</td>
</tr>
<tr>
<td>C</td>
<td>T</td>
</tr>
<tr>
<td>G</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

**String genome:** `CGCCTGGCGTCTGTAC`

**Substring:** `String codon = genome.substring(6, 3);`
Example

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

```java
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.