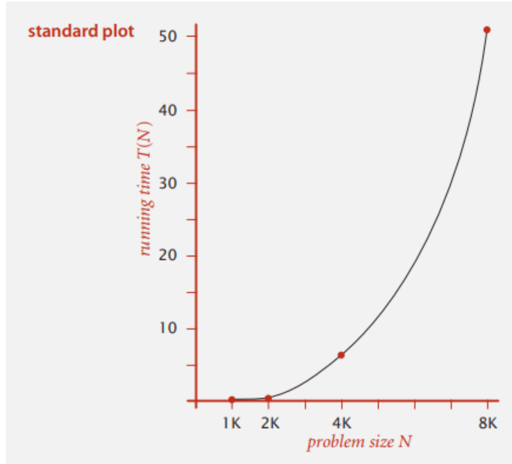
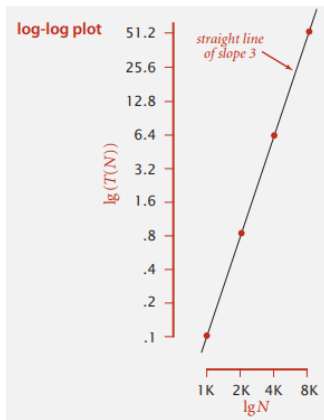
<u>Analysis of Algorithms (Week 1):</u>

- Introduction
    - Reasons to analyze algorithms:
        - Predict performance
        - Compare algorithms
        - Provide guarantees
        - Understand theoretical basis
        - PRIMARY PRACTICAL REASON
            - To avoid performance bugs
    - Will your program be able to solve a large practical input?
        - Is it too slow?
        - Does it run out of memory?
        - Insight
            - 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.
                - In our case, the run time of our program.
            - Hypothesize a model that is consistent with the observations.
            - Predict events using the hypothesis.
                - Maybe the run time of our program on a larger problem size.
            - 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.
- Observations
    - The first step of the scientific method is to make some observations about the running time of the programs.
        - For analysis of algorithms, thats a lot easier than in a lot of scientific disciplines.
        - Example 1: 3-Sum
            - Given N distinct integers, how many triples sum to exactly zero? (Groups of 3 numbers)
            - 3-Sum is actually deeply related to problems in computational geometry.
            - Brute-force method is to just use a triple-for-loop and sum all iterations (i + j + k).
    - Empirical analysis
        - Run the program for various input sizes and measure the running time of each go.
        - Data analysis
            - Standard plot
                - Plots the running time T(N) vs. input size N.

            - 

        - Log-Log plot (Usually results in a straight line)
            - Plots the running time T(N) vs. input size N using log-log scale.
            - 

            - The slope of the straight line is the key to what's going on. In this case, it is 3 (2.999).
        - Regression
            - Fit straight line through data points: aN^b.
            - Slope = b.
            - $$\lg(T(N)) = b \lg N + c$$
              $$b = 2.999$$
              $$c = -33.2103$$
              $$T(N) = a N^b, \text{ where } a = 2^c$$
            - Can disregard image, it's the math explaining aN^b.
            - We got b and c from empirical analysis.
    - Hypothesis
        - The second step in the scientific method,
        - The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.
            - We got 2.999 from our slope in the log-log plot.
            - This means that the “order of growth” of the running time is about N^3.

- Predicting and Verifying
    - The third and fourth steps in the scientific method, we predict future values using our hypothesis and plugging in for N. Then we test them by actually running the program.
    - 51.0 seconds for N = 8000.
        - $1.006 \times 10^{-10} \times 8000^{2.999}$ seconds.
        - Checking with empirical gives us 51.1 as the actual runtime which validates our hypothesis.
    - 408.1 seconds for N = 16000.
        - $1.006 \times 10^{-10} \times 16000^{2.999}$ seconds.
        - Checking with empirical gives us 410.8 as the actual runtime which validates our hypothesis.
- Sidenote
    - Doubling hypothesis
        - A quick way to estimate b in a power-law relationship.
        - Run the program, doubling the size of the input and collecting the run times.
            - Then take the ratio between each runtime.
            - After getting a few, take the lg of all the ratios and it will converge to some constant.
            -

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

seems to converge to a constant $b \approx 3$

        - Hypothesis - Running time is about aN^b with b = lg ratio.
        - Caveat - Cannot identify logarithmic factors with double hypothesis.
        - 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

$51.1 = a \times 8000^3$

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

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 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 - Since so many factors, hard to get precise measurements.
  - Good news - Much easier and cheaper than other sciences.
    - Since it's just a program, we can run it many many times with no cost.

- Mathematical Models
  - Total running time = sum of cost x frequency for all operations.
    - Need to analyze program to determine set of operations.
    - Cost depends on machine, compiler.
    - Frequency depends on algorithm, input data.
  - Simplification 1 - Cost model
    - Cost model
      - Use some basic operation as a proxy for running
    - Simplification
      - Use the most expensive operation (highest frequency) as your cost model and disregard the others.

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	½ N (N − 1) to N (N − 1)

$= \binom{N}{2}$

cost model = array accesses

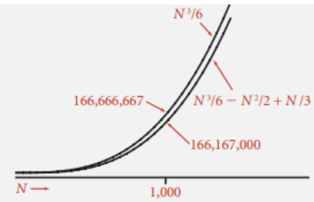
(we assume compiler/JVM do not optimize array accesses away!)

- Simplification 2 - Tilde Notation
  - Estimate running time (or memory) as a function of input size N.
  - Ignore lower order terms.
    - When N is large, the terms are negligible.
    - When N is small, we don't care.
    - Ex 1. $\frac{1}{6}N^3 + 20N + 16 \quad \sim \frac{1}{6}N^3$

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

      Ex 3. $\frac{1}{6}N^3 \underbrace{- \frac{1}{2}N^2 + \frac{1}{3}N}_{\text{discard lower-order terms}} \quad \sim \frac{1}{6}N^3$

      (e.g., N = 1000: 500 thousand vs. 166 million)

      Leading-term approximation
    - By ignoring lower terms, as we get to higher and higher values of N, the terms will be so negligible that it literally wont matter.
      - Technical definition. $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$
  -

operation	frequency	tilde notation
variable declaration	N + 2	~ N
assignment statement	N + 2	~ N
less than compare	½ (N + 1) (N + 2)	~ ½ N²
equal to compare	½ N (N − 1)	~ ½ N²
array access	N (N − 1)	~ N²
increment	½ N (N − 1) to N (N − 1)	~ ½ N² to ~ N²

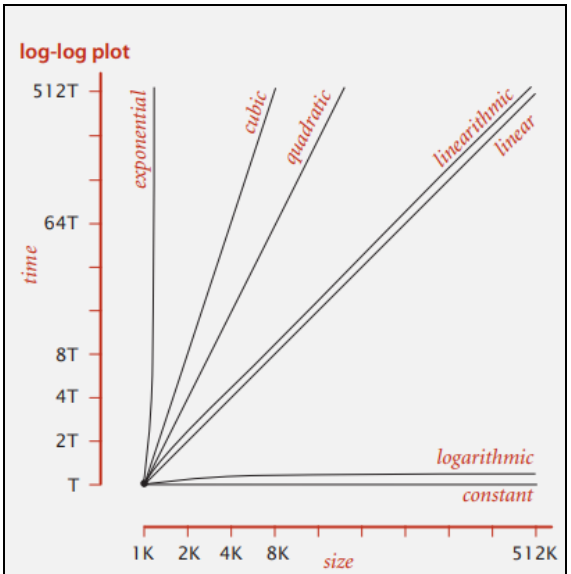
- Estimating a discrete sum
  - How to estimate a discrete sum?
    - Take discrete mathematics course.
    - Replace the sum with an integral, and use calculus.
    - Ex 1. $1 + 2 + \ldots + N$. $\quad \sum_{i=1}^{N} i \sim \int_{x=1}^{N} x\,dx \sim \frac{1}{2}N^2$

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

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

      Ex 4. 3-sum triple loop. $\quad \sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \sim \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dz\,dy\,dx \sim \frac{1}{6}N^3$
- In this course we'll use approximate models.

- Order-of-Growth Classifications
  - Common order-of-growth classifications

        - There is a small set of functions that suffice to describe order-of-growth of typical algorithms.
            - 1
            - logN
            - N
            - NlogN
            - N^2
            - N^3
            - 2^N
        - 

Typical orders of growth

        -

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)

    - Guiding principle
        - Typically, better order of growth means faster in practice.
- Theory of Algorithms

- 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.
- Goals
    - Establish “difficulty” of a problem.
    - Develop “optimal” algorithms.
- Approach
    - Suppress details in analysis
        - Analyze “to within a constant factor”.
        - Eliminate variability in the input model by focusing on the worst case.
- Optimal algorithm
    - Performance guarantee (to within a constant factor) for any input.
    - Algorithm must be good enough that no other algorithm can provide a better performance guarantee.
- Commonly-used notations in the theory of algorithms
notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds
- Example 1
    - Goals
        - Establish “difficulty” of a problem.
        - Develop “optimal” algorithms.

- E.g., 1-sum: "Is there a 0 in the array?"
- Upper bound
  - A specific algorithm
    - E.g., Brute-force algorithm for 1-sum: Look at every array entry.
    - Running time of the optimal algorithm for 1-sum is O(N).
- Lower bound
  - Proof that no algorithm can do better
    - Eg., Have to examine all N entries (any unexamined might be 0).
    - Running time of the optimal algorithm for 1-sum is Ω(N).
- Optimal algorithm
  - Lower bound equals upper bound (to within a constant factor).
  - E.g., Brute-force algorithm for 1-sum is optimal.
    - Its running time is **Θ**(N).

- Example 2
  - Goals
    - Establish "difficulty" of a problem.
    - Develop "optimal" algorithms.
    - E.g., 3-sum: "Is there a combination of 3 numbers that add up to 0 in the array?"
  - Upper bound
    - A specific algorithm
      - E.g., Improved algorithm for 3-sum.
      - Running time of the optimal algorithm for 1-sum is O(N^2logN).
  - Lower bound
    - Proof that no algorithm can do better
      - Eg., Have to examine all N entries to solve 3-sum.
      - Running time of the optimal algorithm for 1-sum is Ω(N).
  - Open problems
    - What is the optimal algorithm for 3-sum? Is there even an optimal algorithm for 3-sum? Is it even subquadratic or quadratic lower bound for 3-sum?
- Algorithm design approach
  - Start
    - Develop an algorithm.
    - Prove a lower bound
  - Gap?
    - Lower the upper bound (discover a new algorithm).
    - Raise the lower bound (more difficult).

- Memory
  - Basics
    - How many bits/bytes does our program use?
      - Bit - 0 or 1
      - Byte - 8 bits
      - Megabyte (MB) - 1 million or 2^{20} bytes.

- Gigabyte (GB) - 1 billion or 2^{30} bytes.

- Typical memory usage for primitive types and arrays in Java

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

for primitive types

type	bytes
char[]	2N + 24
int[]	4N + 24
double[]	8N + 24

for one-dimensional arrays

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

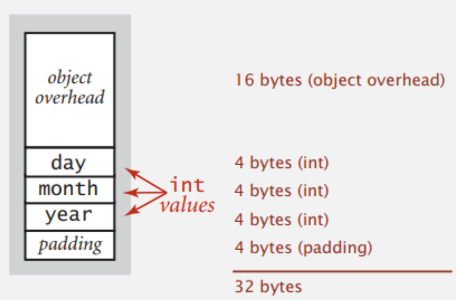
for two-dimensional arrays

- Typical memory usage for objects in Java

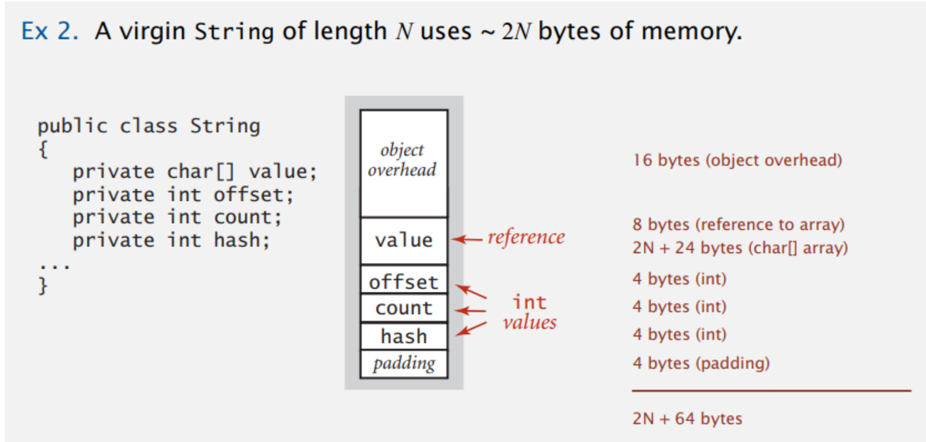
Object overhead. 16 bytes.
Reference. 8 bytes.
Padding. Each object uses a multiple of 8 bytes.

Ex 1. A Date object uses 32 bytes of memory.

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```



- Padding adds enough so that the overall memory adds up to a multiple of 8.

        - 

    - Shallow vs Deep memory usage
        - Shallow
            - Don’t count referenced objects.
        - Deep
            - If array entry or instance variable is a reference, add memory (recursively) for referenced object.
- 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 behaviour.
    - Scientific method
        - Mathematical model is independent of a particular computer system; applies to machines not yet built.
        - Empirical analysis is necessary to validate mathematical models and to make predictions.