Algorithms Part I

# Week 1:

## Lecture 2.1: Dynamic Connectivity

How to make a usable algorithm?

Steps:

* + Model the problem.
  + Find an algorithm to solve it.
  + Fast enough? Fits in memory?
  + If not, figure out why.
  + Find a way to address the problem.
  + Iterate until satisfied.

This is a scientific approach to design an algorithm.

*Dynamic Connectivity Problem:*

Given a set of N objects labeled 0 – N-1.

* **Union Command:** connects 2 objects
* **Find/Connected Query** *(Boolean)***:**  is there a path connecting two objects?
* Must be efficient – Only needs to answer the question - **is there a path** *not* **what is the path**.

This problem has many applications:

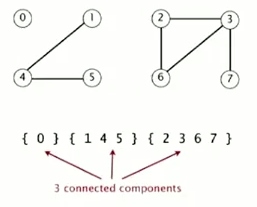
* Pixels in a digital photo
* Computers in a network
* Friends in a social network
* Transistors in a computer chip
* Elements in a mathematical set
* Variable names in a Fortran program
* Metallic sites in a composite system

We assume the “is connected to” is an equivalence relation:

Meaning:

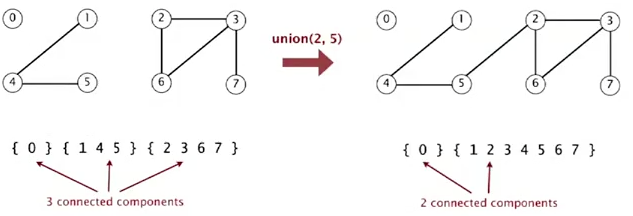
* Reflexive: *p*is connected to *p*
* Symmetric: if *p* is connected to *q*, then *q* is connected to *p*
* Transitive: if *p* is connected to *q* and *q* is connected to *r*, then *p* is connected to *r*.

*Connected Components:* Maximal **set** of objects that are mutually connected.



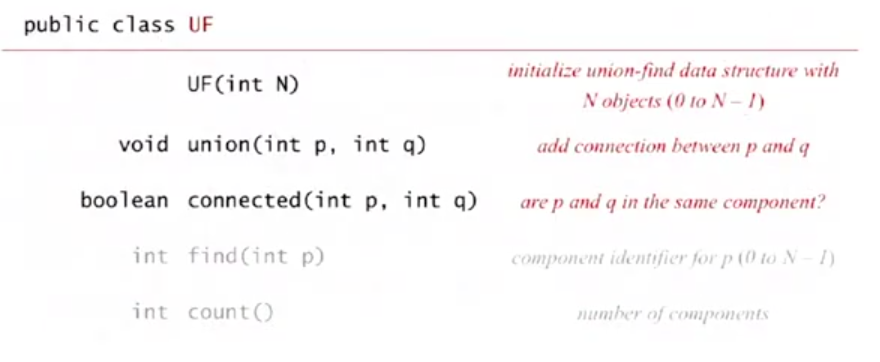
*Find Query:*  Check if two objects are in the same component.

*Union Command:* Replace components containing two objects with their union.



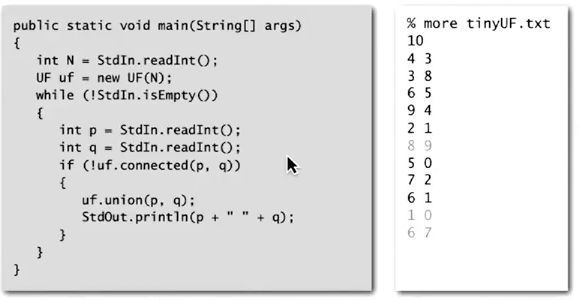
*Goal:*  Design an efficient data structure for union-find.

* Number of objects *N* can be huge.
* Number of operations *M* can be huge.
* Find queries and union commands may be intermixed.



*Dynamic Connectivity Test Client:*

* Read in number of objects *N* from standard input.
* Repeat:
* Read in pairs of integers from standard input
* If they are not yet connected connect them and print out the pair

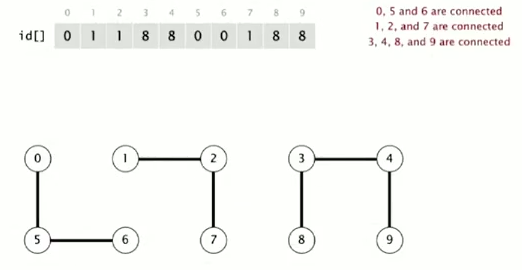


## Lecture 2.2: Quick Find

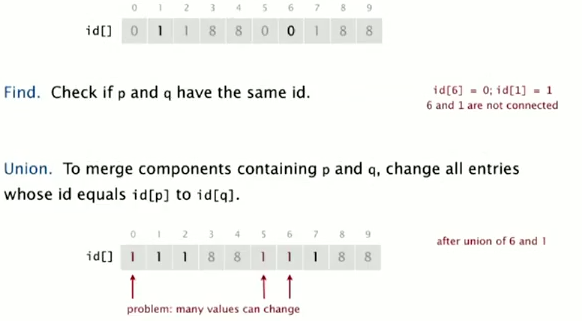
*Quick Find [eager approach/algorithm]:*

Data Structure:

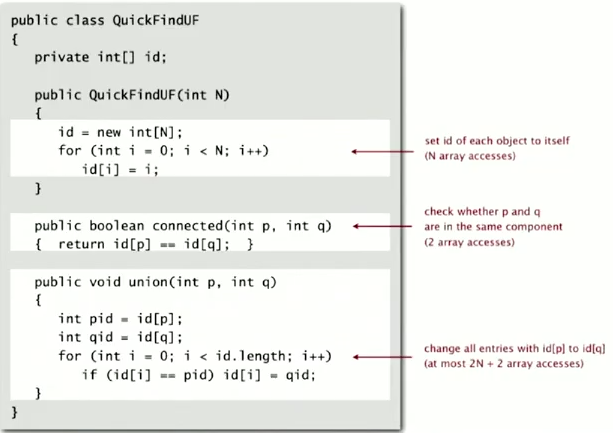
* Integer array id[] of size N
* Interpretation: *p* and *q*  are connected if and only if they have the same id.



How Find and Union Operations will work:

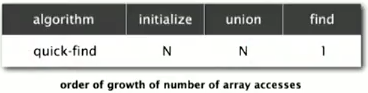


Quick-Find: Java Implementation



Quick-find is too Slow:

*Cost Model:* Number of Array accesses (for read or write).



*Quick-find Defect:* Union is too expensive

*Ex.* Takes *N2* (quadratic) array accesses to process sequence *N* union commands on *N* objects.

*Quadratic Algorithms do not Scale:*

**Rough Standard (for now).**

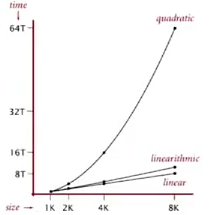
* 109 operations per second.
* 109 words of main memory.
* Touch all words in approximately 1 second.

**Ex. Huge problem for quick-find.**

* 109 union commands on 109 objects.
* Quick-find takes more than 1018 operations.
* 30+ years of computer time.

**Quadratic Algorithms don’t scale with technology.**

* New computers may be 10x as fast
* But, has 10x as much memory – want to solve a problem that is 10x as big.
* With quadratic algorithm, takes 10x as long!

****

## Lecture 2.3 Quick Union

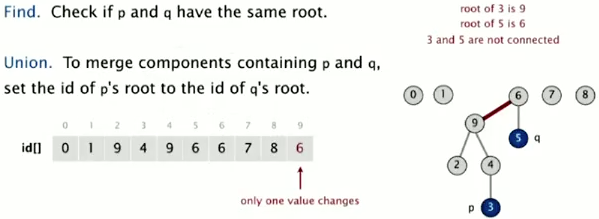
*Quick Union [lazy approach]:*

Data Structure:

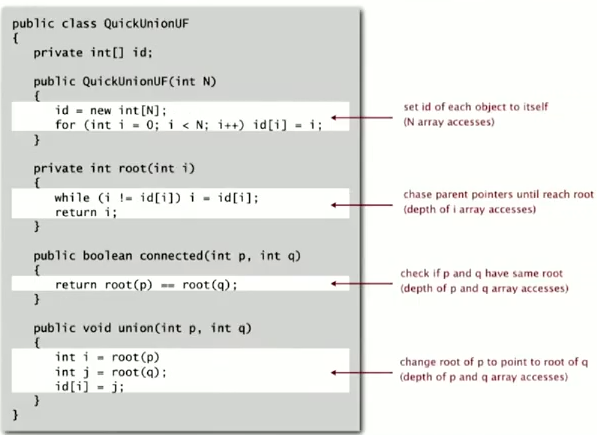
* Integer array id[] of size N.
* Interpretation: id[i] is parent of i.
* **Root** of i is id[id[id[…id[id]…]]].



How Find and Union Operations will work.

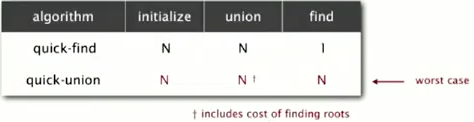


Quick-union Java Implementation:



Quick-union is also too Slow:

*Cost Model:* Number of array accesses (for read or write).



*Quick-union defect:*

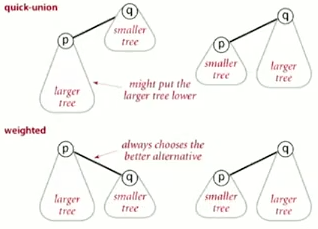
* Trees can get tall.
* Find too expensive (could be *N* array accesses).

## Lecture 2.4 Quick-union Improvements

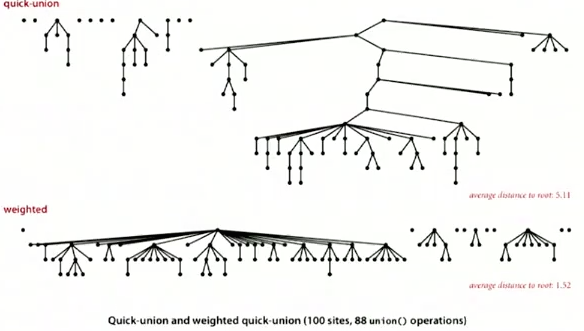
*Improvement 1: Weighting*

Weighted Quick-union.

* Modify quick-union to avoid tall trees
* Keep track of size of each tree (number of objects).
* Balance by linking root of smaller tree to root of larger tree.



Quick-union vs. Weighted quick-union



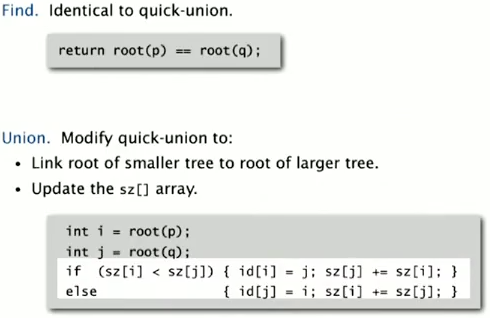
Average distance to root:

* Quick-union: 5.11
* Weighted quick-union: 1.52

Data Structure:

* Same as quick-union but has extra array sz[i] to count the number of objects in the tree rooted at i.

Weighted quick-union Java Implementation:



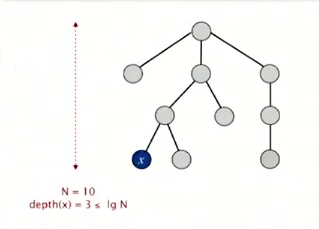
Weighted Quick-union Analysis:

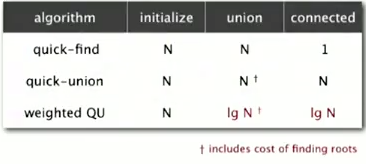
*Running Time:*

* Find: takes time proportional to depth of *p* and *q*.
* Union: takes constant time, given roots

*Proposition:*

* Depth of any node *x* is at most lg *N* (log2 *N*).

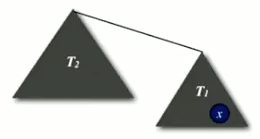




*Proof:* When does the depth of *x* increase?

Increases by 1 when tree *T*1 containing *x* is merged into another tree *T*2.

* The size of the tree containing *x* at least doubles since |*T*2| |*T*1|.
* Size of tree containing *x* can double at most lg *N* times.



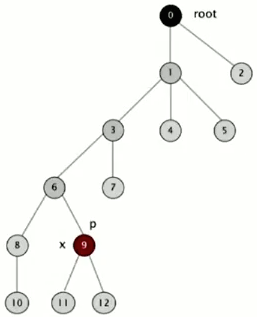
We can stop at this acceptable performance, but it is easy to improve further.

*Improvement 2: Path Compression:*

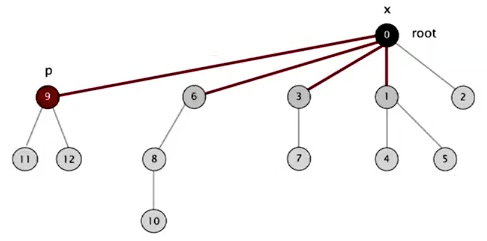
Quick-union with path compression.

* After computing root of *p*, set the id of each examined node to point to that root.

Start:



End:



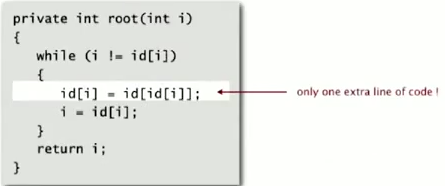
Quick-union with Path Compression Java Implementation:

*Two-pass implementation:*

* Add second loop to root() to set the id[] of each examined node to the root.

*Simpler one-pass:*

* Make every other node in path point to its grandparent (thereby halving path length).



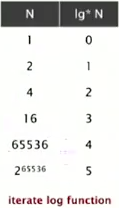
Use this because it keeps the tree almost completely flat.

Weighted Quick-union with Path Compression Analysis (WQUPC):

*Proposition* [Hopcroft-Ulman, Tarjan]

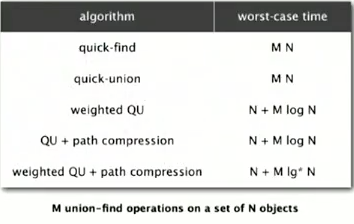
Starting from an empty data structure, any sequence of *M* union-find ops on *N* objects makes array accesses.

* Analysis can be improved to *N* + *M*
* Simple algorithm with fascinating mathematics.



*There is no such algorithm for this that is linear, but this is so close to linear that it is called linear.*

Summary: Dynamic Connectivity Problem



* WQUPC reduces time from 30 years to 6 seconds.
* Supercomputers won’t help; good algorithms do.

## Lecture 2.5 Union-find Applications

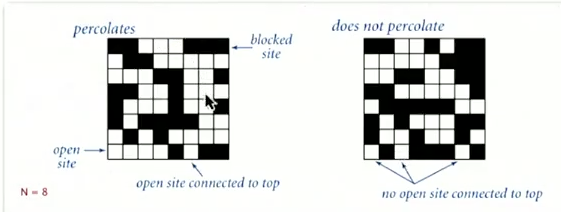
Union-find Applications:

* **Percolation**
* Games (Go, Hex).
* **Dynamic Connectivity**
* Least common ancestor
* Equivalence of finite state automata.
* Hoshen-Kopelman algorithm in physics
* Hinley-Milner polymorphic type inference
* Kruskal’s minimum spanning tree algorithm
* Compiling equivalence statements in Fortran
* Morphological attribute openings and closings
* Matlab’s bwlabel() function in image processing

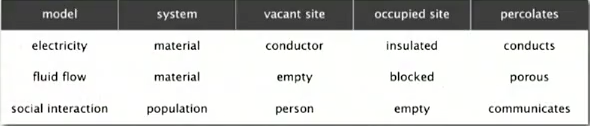
Percolation:

*A model for many physical systems:*

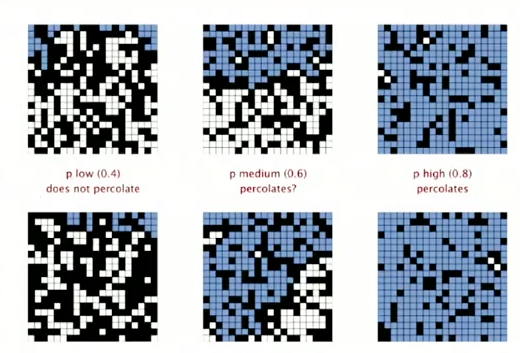
* *N*-by-*N* grid of sites.
* Each site is open with probability *p* (or blocked with probability 1-*p*)
* System **percolates** if and only if the top and bottom are connected by open sites.



Examples of the Percolation Model:



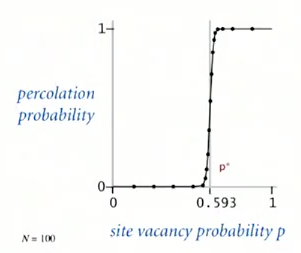
Likelihood of percolation depends on site vacancy probability *p*.



Percolation Phase Transition:

*When* N *is large, theory guarantees a sharp threshold* p\**.*

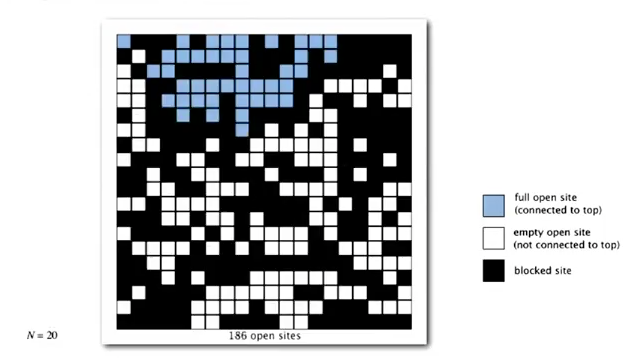
* *p* > *p*\*: almost certainly percolates
* *p* < *p*\*: almost certainly does not percolate



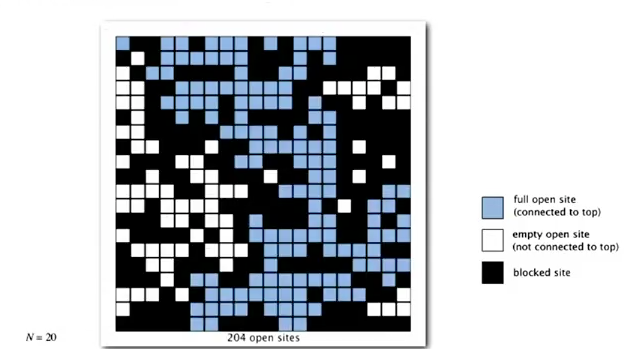
Monte Carlo Simulation

* Initialize *N*-by-*N* whole grid to be blocked
* Declare random sites open until top connected to bottom
* Vacancy percentage estimates *p*\*.

Start:



End:

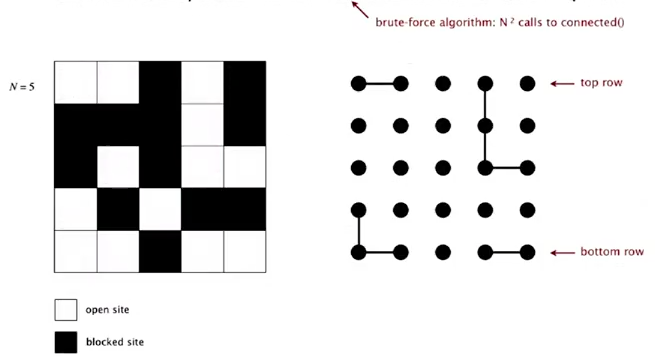


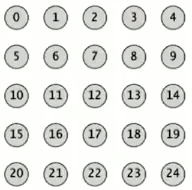
Run this millions of times.

Dynamic Connectivity solution to estimate percolation threshold

*How to check whether an* N*-by-*N *system percolates*

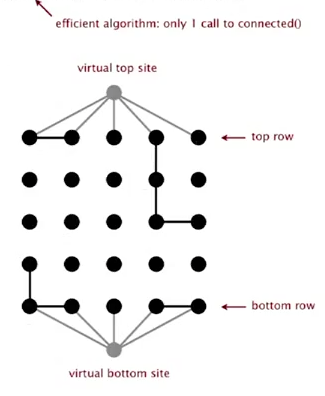
* Create an object for each site and number it 0-*N*2 – 1
* Sites are in same component if connected by open sites
* Percolates if and only if any site on bottom row is connected to site on top row.





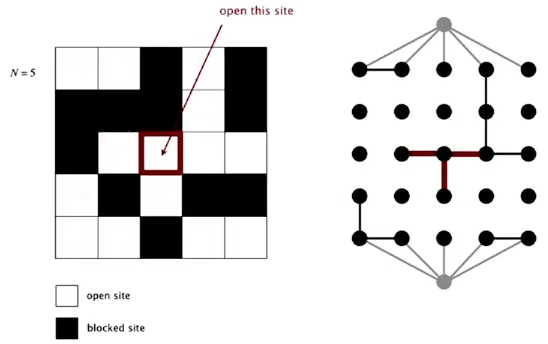
*Clever Trick:* Introduce 2 virtual sites (and connections to top and bottom)

* Percolates if and only if virtual top site is connected to virtual bottom site



Modeling Opening a New Site

* Connect new site to all of its adjacent open sites (up to 4 calls to union())



Using this quick algorithm we can get an accurate percolation threshold *p*\*.

## Lecture 3.1 Analysis of Algorithms

Cast of Characters [different viewpoints]

**Student** might play

any or all of these

roles someday.

**Programmer** needs **Client** wants to **Theoretician** Basic

to develop a working solve problem wants to **blocking and**

solution. efficiently. understand. **tackling** is

sometimes

necessary.

[this lecture]

Running Time:

* Number of billions of operations per second
* In an Analytic Engine – times crank is turned
* In other words: time taken to run a program

Reasons to Analyze Algorithms:

* Predict Performance
* Compare algorithms
* Provide guarantees
* Understand theoretical basis
* **Primary Reason:** avoid performance bugs

*Qs*: Practical challenge: Will my program solve a large input?

*Ans*: Use the Scientific Method to Analyze Algorithms:

*Scientific Method:*

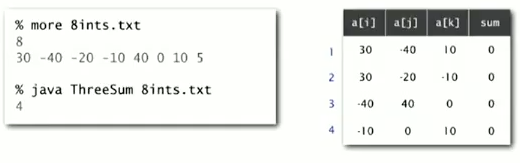
* **Observe** some feature of the natural world – a running time of a program (computer)
* **Hypothesize** a model that is consistent with the observations
* **Predict** events using the hypothesis
* **Verify** the predictions by making further observations
* **Validate** by repeating until the hypothesis and observations agree

*Principles:*

* Experiments must be **reproducible**
* Hypothesis must be **falsifiable**

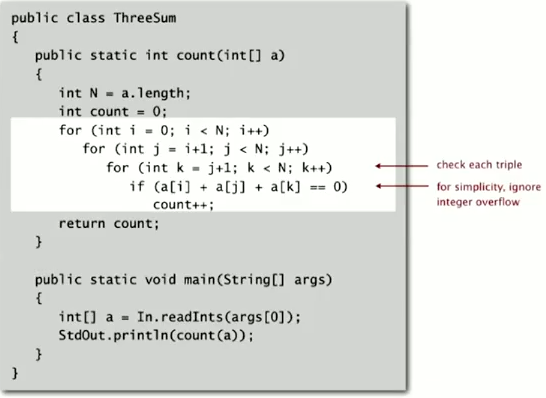
## Lecture 3.2 Observations

*3-Sum Example:* Given *N* distinct integers, how many triples sum to exactly zero?



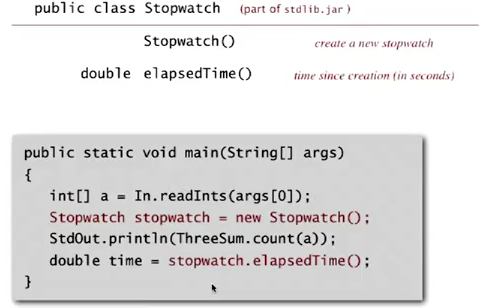
*Context:* Deeply related to problems in computational geometry.

Java Implementation: Brute-force algorithm



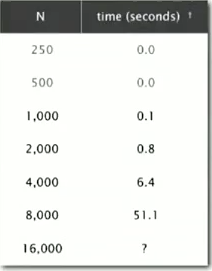
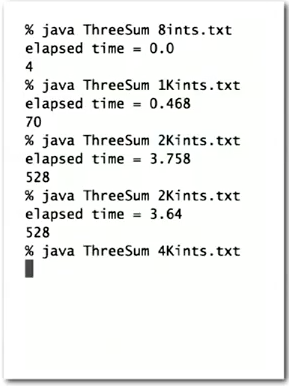
Measuring Running Time

* Manually using a clock
* Automatically using Stopwatch Library



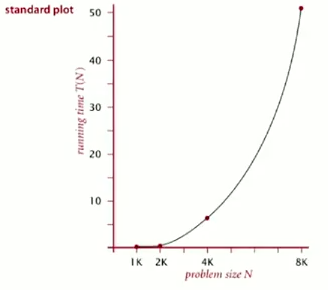
Empirical Analysis

*Run program for various input sizes and measure running time*

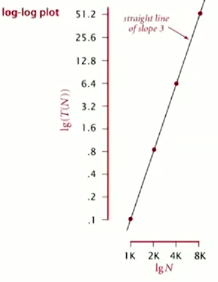
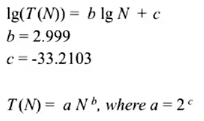


Data Analysis

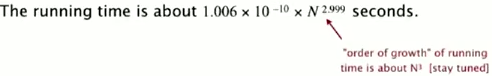
*Standard Plot:* Plot running time *T(N)* vs. input size *N.*



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

*Regression: *

*Hypothesis:* **

*Predictions:*

* 51.0 seconds for *N* = 8,000.
* 408.1 seconds for *N* = 16,000.

*Observations:*

