Analysis of Algorithms

Frank Walsh/Eamonn Deleaster

Agenda

- Introduction
  - Why analyse algorithms
- Observations
- Mathematical Models
- Growth Classification for algorithms
- Theory of Algorithms

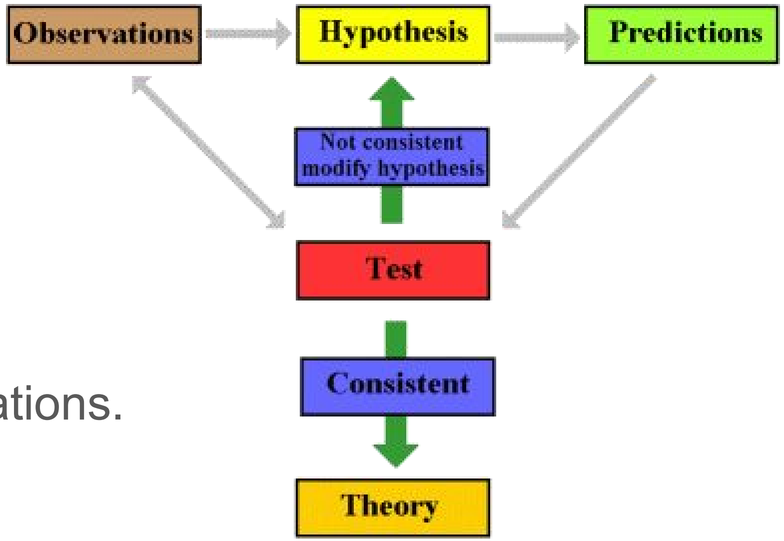
Why analyse algorithms

- Programmers need to develop working solutions to problem
- Algorithm analysis helps developers to write programs that:
  - provide an optimal working solution
  - predict resources and time necessary to execute a program
  - give guarantees regarding performance.
- Helps to avoid performance problems
  - Clients get poor performance because programmer didn't understand or investigate performance characteristics of program.

Scientific Method

Approach that scientists use to understand the natural world

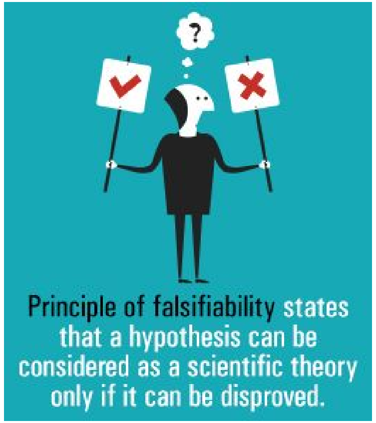
- Observe some feature of the natural world, generally with precise measurements.
- Hypothesise a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.



Scientific Method

Key features of the scientific method:

- Experiments must be reproducible
  - so that you can convince others.
- Your hypothesis must be falsifiable
  - “No amount of experimentation can ever prove me right; a single experiment can prove me wrong”
- Are these scientific hypothesis?:
  - “There is life on other planets”
  - “Two objects will hit the ground at the same time when dropped from the same height(excluding air resisitance)”



Observations

- We can make quantitative measurements of the running time of our programs.
  - Easy compared to other sciences (don’t need to build a hadron collider)
- Answers a core question: How long will my program take?
- Initial observation, the problem size:
  - The problem size can be the size of input or value of input)
  - Most of the time, programming running time is insensitive to the input itself, but IS SENSITIVE to the size of the input.

Observations: Example

ThreeSum: Given N distinct integers, how many triples sum to exactly zero:

```
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 = In.readInts(args[0]);
  System.out.println.println(count(a));
}
}
```

Observation: Example

- How do we measure running time
  - Manual (e.g. stopwatch)
  - Use JUnit(look at running times of methods)
  - Automatic (build it into the program). Can use the Stopwatch() class.

```
public static void main(String[] args)
{
int[] a = In.readInts(args[0]);
Stopwatch   stopwatch = new Stopwatch();
System.out.println(ThreeSum.count(a));
double  time = stopwatch.elapsedTime();
System.out.println("elapsed time " + time);
}
```

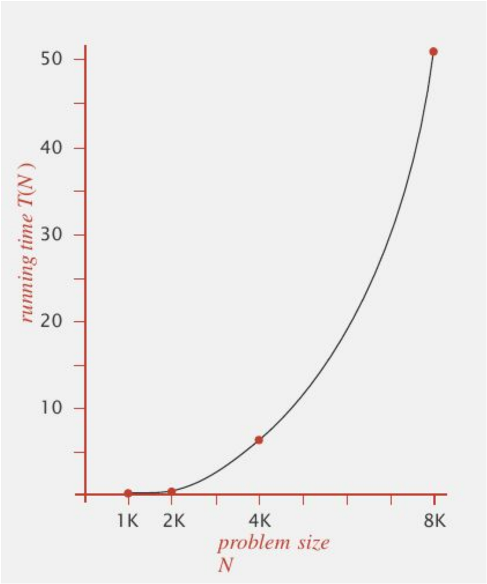
Observation: Empirical Analysis

Running for different size input (N):

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	?

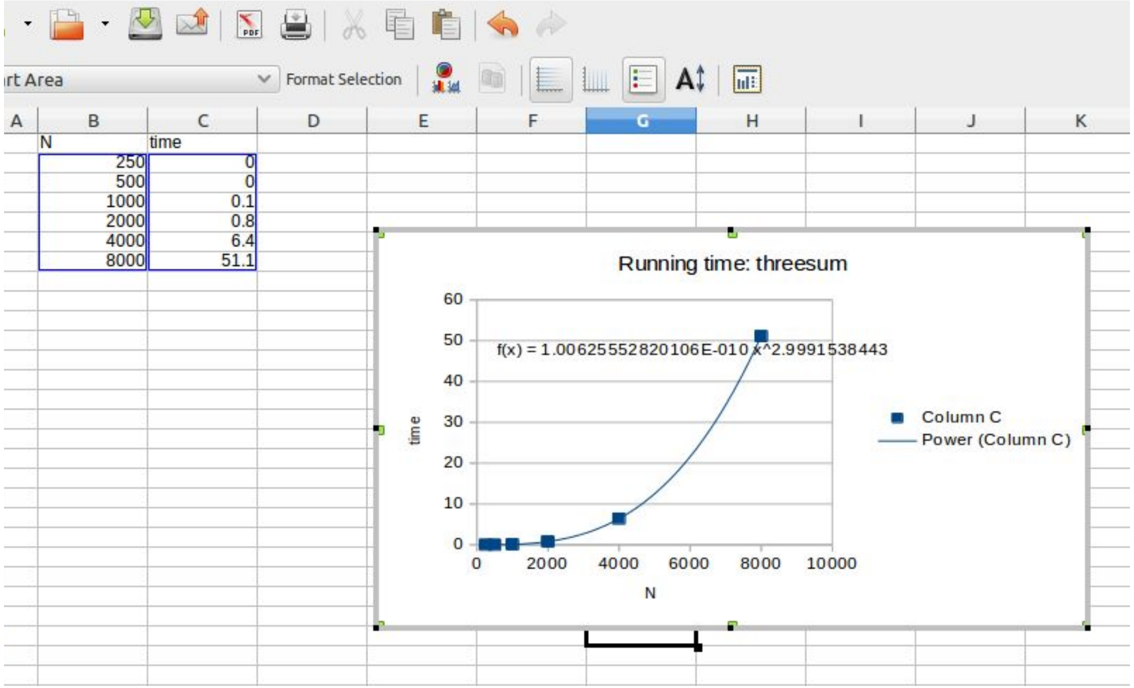
Observation: Data Analysis

- Plot the running time T(N) against input size (N)
- How can we predict values for 16K
  - get an equation for the trendline in the graph
  - Equation can be used to calculate how long will my program take, as a function of the input size.
- One approach:
  - use a tool that can “fit” an equation to the trendline.
  - use the equation to predict other values



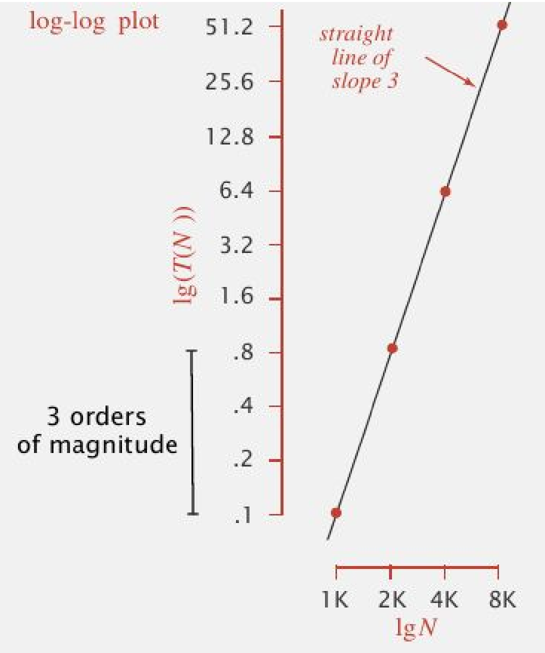
Observation: Data analysis with Spreadsheet

- Chart data as X-Y plot
- Insert Trendline
- More info here: http://www.cpp.edu/~seskandari/documents/Curve_Fitting_William_Lee.pdf
- Use equation for trendline to predict future values:
- Aproximating eqn: $T(N) = 1.006x10^{-10} N^3$



Observation: Data Analysis using logs

- Log-log plot: Plot running time T (N) vs. input size N using log-log scale.
- Get straight line with slope of 3:
  - eqn. of straight line is y=mx + c
  - for this graph: lg(T (N)) = b lg N + c
  - b=2.99, c=-33.2103
- $T(N)=aN^b$, where $a=2^c$ using power law https://en.wikipedia.org/wiki/Power_law
- Now we can make a Hypothesis for running time
  - Running time is approx. $2^{-33.21}N^3$
    $T(N)=1.006 \times 10^{-10} N^3$
- Same as previous slide...



Prediction and Validation

- Hypothesis: Running time is $1.006 \times 10^{-10} N^3$ where N is the size of the input
- Predictions:
  - 51 seconds for N=8000
  - 408.1 seconds for N=16000
- Observations:

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1
16,000	410.8

Hypothesis validated!

What effects the Running Time

- System independent effects:
  - Algorithm
  - Input Data
- System dependent effects
  - hardware: processor, memory
  - software: compiler, garbage collection etc.
  - System: operating system, network, other apps…

- System independent effects determine the exponent in eqn.
- Both System independent and dependent effects determine the constant
- Difficult to get precise measurement but easier to obtain measurements
  - no animals were harmed in this experiment!
  - Can run large number of experiments.

Mathematical Models for Algorithms

Mathematical Models for Algorithms

- Example: 1-Sum
  - How many instructions are performed in the code:

```
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 $2N$

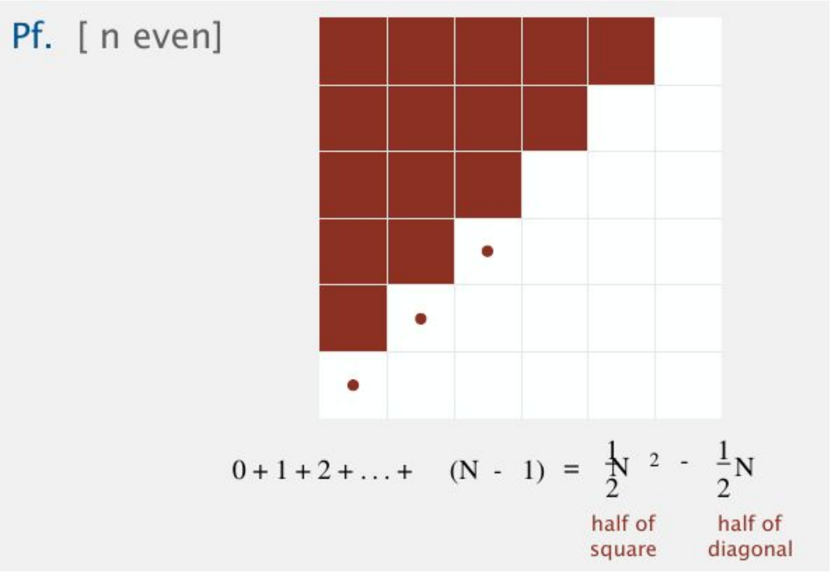
Mathematical Models for Algorithms

Example: 2-sum

- How many instructions as a function of input size

```
1. int count = 0;
2. for (int i = 0; i < N; i++)
3.    for (int j = i+1; j < N; j++)
4.       if (a[i] + a[j] == 0)
5.          count++;
```

- Line 4 is executed (N-1)+(N-2)+(N-3)+... +2+1+0 times



Mathematical Models for Algorithms

Example: 2-sum

```
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++;
```

- NEED TO SIMPLIFY!!!

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$
equal to compare	$\frac{1}{2} N (N - 1)$
array access	$N (N - 1)$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$

tedious to count exactly

Mathematical Cost Models: Simplify

"...we shall therefore only attempt to count the number of multiplications and recordings. " — Alan Turing

- Identify a basic operation
  - usually the operation that executes the most number of times
  - Can ignore other operations
- In 2-sum, the array accesses in the "if" statement is a good choice:

```
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++;
```

Mathematical Cost Model: Simplicity

Time efficiency can analysed by determining the number of repetitions of the ***basic operation*** as a function of ***input size.*** For big input sizes, N:

$$T(N) \approx c_{op}C(N)$$

Running Time

Execution time of basic operation

Number of times basic operation is executed

Mathematical Cost Model: Simplify

Use “Tilda Notation”

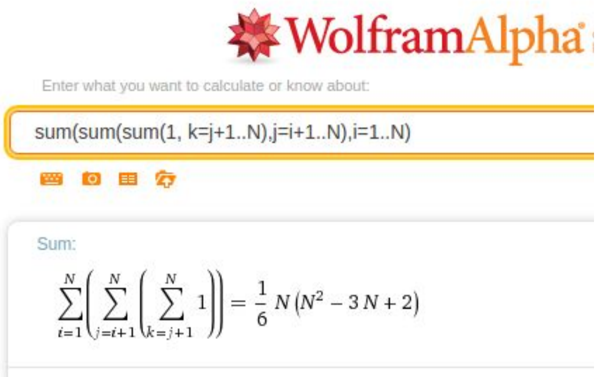
- Estimate Number of Times Basic Operation is executed and use Higher Order term:
- For 2-Sum example:
  - Basic Operation runs N(N-1)

$C(N) = N^2 - N \sim N^2$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$
equal to compare	$\frac{1}{2} N(N - 1)$
array access	$N(N - 1)$ ← cost model = ar
increment	$\frac{1}{2} N(N - 1)$ to $N(N - 1)$

Mathematical Cost Model

3-Sum Example:




```
1int N = a.length;
2 int count = 0;
3 for (int i = 0; i < N; i++)
4   for (int j = i+1; j < N; j++)
5     for (int k = j+1; k < N; k++)
6       if (a[i] + a[j] + a[k] == 0)
7          count++;
 return count;
Basic Operation (line 6: “touches the array 3 times)
```

Number of times Line 6 executes: N(N-1)(N-2)/6 ~ $N^3/6$ (Can calculate using discrete maths or online tool: http://www.wolframalpha.com/)

Number of times array accessed C(N) ~ $N^3/2$

What does this tell us about how the algorithm running time grows as you increase size?

$T(N) = c_{op}C(N) = c_{op}N^3/3$

Mathematical Cost Model: Summary

Develop a Mathematical model using the following steps

■ Develop an input model, including a definition of the problem size(e.g. size of array)

■ Identify the inner loop.

■ Define a cost model that includes the “basic operation” in the inner loop.

■ Determine the frequency of execution of the basic operation for the given input.

Doing so might require mathematical analysis...

Order of Growth Classification

Common Order of Growth classifications

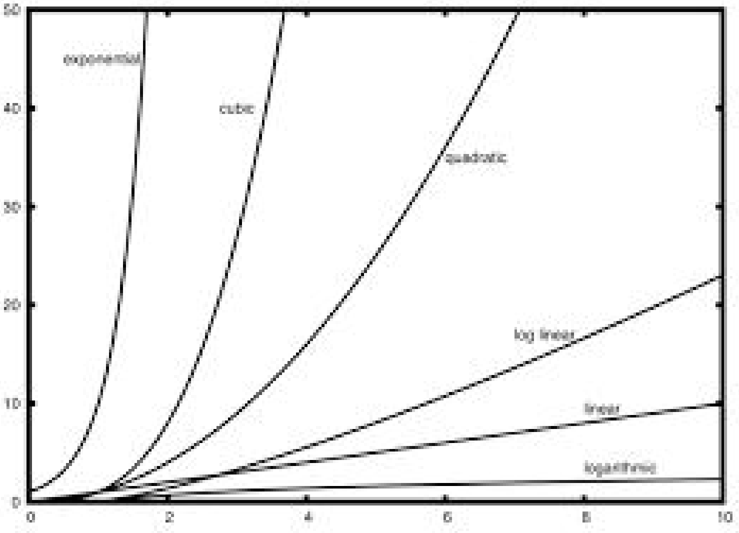
- If $f(N) \sim cg(N)$ for some constant c>0 then the Order of Growth of f(n) is g(n).
  - example Threesum:
    $C(N) \sim 1/2N^3$ so order of growth is N^3

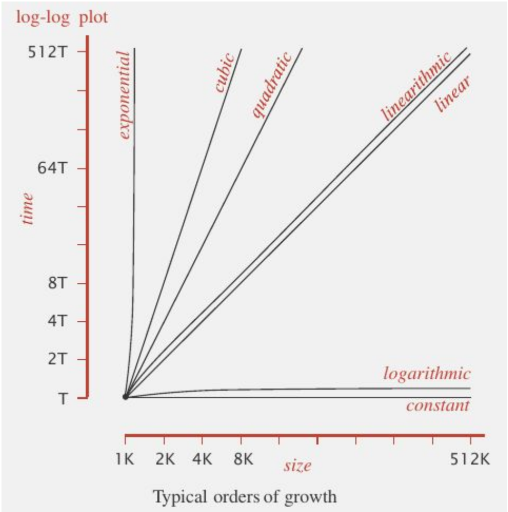
```
int count = 0;

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++;
```

Common order-of-growth classifications

- Most algorithms can be classified using the following functions of their input size:
  1, log N, N, NlogN,N^2 ,N^3, and 2^N





Common order-of-growth classifications

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	$T(N)$

Demo - Binary Search

- Problem: Given a SORTED array and a key, find index of the key in the array?
- Solution: Use suitable search algorithm, Binary Search
  - Compare key against middle
  - Smaller:- go left
  - Larger:- go right
  - Equal:- return location

```java
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;
}
```

Example - Binary Search Analysis

- What's the basic operation
  - 1st key comparison - runs every time
- C(N) is the basic operation count in a sub-array of size <=N
- C(N) is less than or equal to the number of key comparisons to search left or right half of the array, C(N/2) + 1.
- This is a **recurrence relation.**

C(N) <= C(N/2)+1 for N>1 and C(1)=1

Assume N is a power of 2: $N=2^x$

$C(N) \le C(N/2) + 1$

$\le C(N/4) + 1 + 1$

$\le C(N/8) + 1 + 1 + 1$

……..

$\le C(N/N) + 1 + 1 + \ldots + 1$ (i.e. $\le C(N/2^x) + 1 + x$)

$= 1 + \lg N$

Apply recurrence to 1st term

- Binary Search is Logarithmic

Example - 2Sum

- See the 2Sum algorithm, determine the number of pairs of integers that sum to 0.
- 2Sum solved in quadratic time (N^2)
- Possible improvement
    a. sort array a (MergeSort: NlogN)
    b. for each nimber a[i] search for -a[i] (Binary Search: NlogN)
- Overall Running time: NLogN

```
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++;
```

```
Arrays.sort(a);
 int N = a.length;
int cnt = 0;
for (int i = 0; i < N; i++)
 if (BinarySearch.rank(-a[i], a) >
 i)
 cnt++;
return cnt;
```

Example - 3Sum improvement

Algorithm:

1. Sort Array a[]
2. For each pair of numbers a[i] and a[j] binary search for -(a[i]+a[j])

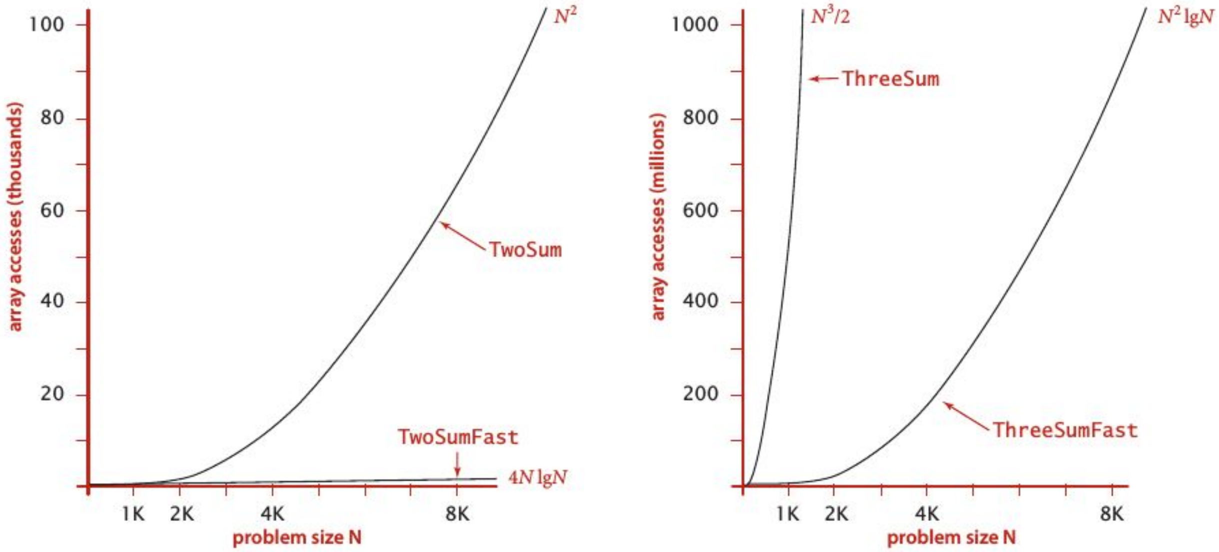
Analysis

1. Sort is N^2 (Insertion Sort)
2. Binary search is $N^2 LogN$

```java
Arrays.sort(a);
 int N = a.length;
 int cnt = 0;
for (int i = 0; i < N; i++)
for (int j = i+1; j < N; j++)
 if (BinarySearch.rank(-a[i]-a[j], a) > j)
 cnt++;
return cnt;
```

Example: 2Sum and 3Sum Comparisons

Typically, better order of growth means faster running times



Costs of algorithms to solve the 2-sum and 3-sum problems

Algorithm Theory

Analysis Types

- Best Case
  - Lower bound on cost
  - Determined by “easiest input”.
- Worst Case
  - Upper bound on cost
  - Provides a worst case guarantee
- Average Case
  - Expected cost of random input
  - Predictor for performance

Common Notation in Algorithm Theory

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

Theory of Algorithms

Goals.

- Establish "difficulty" of a problem.

- Develop "optimal" algorithms.

Approach.

- Eliminate variability in input model: focus on the worst case.

- Establish Upper bound and Lower bound.

Upper bound is performance guarantee

Lower bound. proof that no algorithm can do better.

In-Class Example

```
private static int maxValue(char[] chars) {
    int max = chars[0];
    for (int ktr = 1; ktr < chars.length; ktr++) {
        if (chars[ktr] > max) {
            max = chars[ktr];
        }
    }
    return max;
}
```

- Input Size?
- Basic Operation?
- Best Case?
- Worst Case?
- Average Case?
- Classification

In-Class Example

- Input Size?
- Basic Operation?
- Best Case?
- Worst Case?

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//            and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

In-Class Example

- Input Size?
- Basic Operation?
- Best Case?
- Worst Case?
- Average Case?

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
  $count \leftarrow count + 1$
  $n \leftarrow \lfloor n/2 \rfloor$
return $count$