Topic Number 8
Algorithm Analysis

"bit twiddling: 1. (pejorative) An exercise in tuning (see *tune*) in which incredible amounts of time and effort go to produce little noticeable improvement, often with the result that the code becomes incomprehensible."

- The Hackers Dictionary, version 4.4.7

Is This Algorithm Fast?

- Problem: given a problem, how fast does this code solve that problem?
- Could try to measure the time it takes, but that is subject to lots of errors
  - multitasking operating system
  - speed of computer
  - language solution is written in

Attendance Question 1

- "My program finds all the primes between 2 and 1,000,000,000 in 1.37 seconds."
  - how good is this solution?

A. Good

B. Bad

C. It depends

Grading Algorithms

- What we need is some way to grade algorithms and their representation via computer programs for efficiency
  - both time and space efficiency are concerns
  - are examples simply deal with time, not space
- The grades used to characterize the algorithm and code should be independent of platform, language, and compiler
  - We will look at Java examples as opposed to pseudocode algorithms

Big O

- The most common method and notation for discussing the execution time of algorithms is "Big O"
- Big O is the *asymptotic execution time* of the algorithm
- Big O is an upper bounds
- It is a mathematical tool
- Hide a lot of unimportant details by assigning a simple grade (function) to algorithms

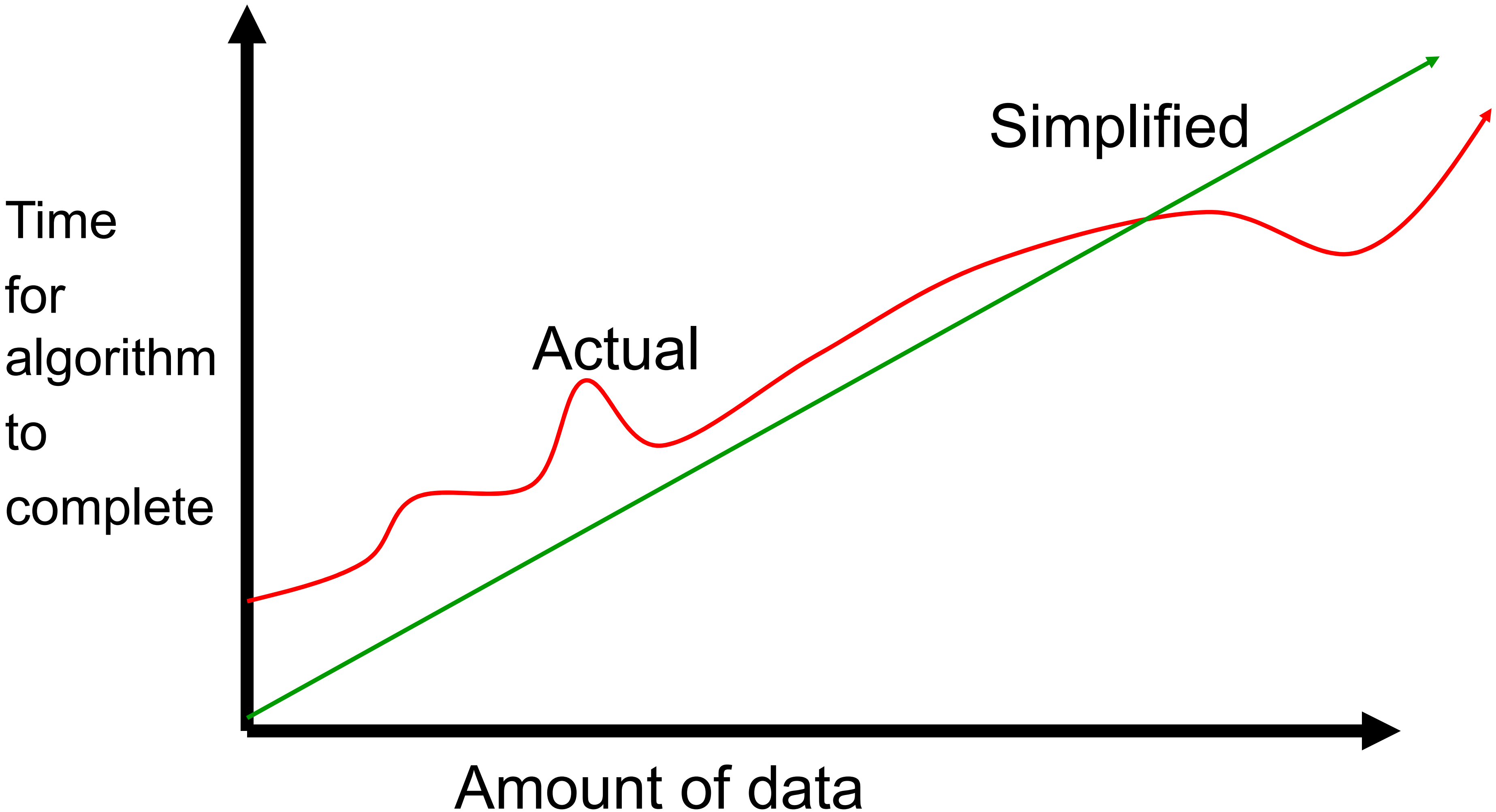
Typical Big O Functions – "Grades"

Function	Common Name
N!	factorial
2^N	Exponential
N^d, d > 3	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
N log N	N log N
N	Linear
$\sqrt{N}$	Root - n
log N	Logarithmic
1	Constant

Big O Functions

- N is the size of the data set.
- The functions do not include less dominant terms and do not include any coefficients.
- $4N^2 + 10N – 100$ is not a valid F(N).
  - It would simply be $O(N^2)$
- It is possible to have two independent variables in the Big O function.
  - example O(M + log N)
  - M and N are sizes of two different, but interacting data sets

Actual vs. Big O



Formal Definition of Big O

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm. (The typical Big functions table)
  - c and N_0 are constants

What it Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Yuck

- How do you apply the definition?
- Hard to measure time without running programs and that is full of inaccuracies
- Amount of time to complete should be directly proportional to the number of statements executed for a given amount of data
- Count up statements in a program or method or algorithm as a function of the amount of data
  - This is one technique
- Traditionally the amount of data is signified by the variable N

Counting Statements in Code

- So what constitutes a statement?
- Can't I rewrite code and get a different answer, that is a different number of statements?
- Yes, but the beauty of Big O is, in the end you get the same answer
  - remember, it is a simplification

Assumptions in For Counting Statements

- Once found accessing the value of a primitive is constant time. This is one statement:

```
x = y; //one statement
```

- mathematical operations and comparisons in boolean expressions are all constant time.

```
x = y * 5 + z % 3; // one statement
```

- if statement constant time if test and maximum time for each alternative are constants

```
if( iMySuit == DIAMONDS || iMySuit == HEARTS )
  return RED;
else
  return BLACK;
// 2 statements (boolean expression + 1 return)
```

Counting Statements in Loops Attendenance Question 2

- Counting statements in loops often requires a bit of informal mathematical *induction*
- What is output by the following code?

```
int total = 0;
for(int i = 0; i < 2; i++)
    total += 5;
System.out.println( total );
```

A. 2 B. 5 C. 10 D. 15 E. 20

Attendances Question 3

- What is output by the following code?

```
int total = 0;
// assume limit is an int >= 0
for(int i = 0; i < limit; i++)
    total += 5;
System.out.println( total );
```

A. 0

B. limit

C. limit * 5

D. limit * limit

E. $limit^5$

Counting Statements in Nested Loops Attendance Question 4

- What is output by the following code?

```
int total = 0;
for(int i = 0; i < 2; i++)
    for(int j = 0; j < 2; j++)
        total += 5;
System.out.println( total );
```

A. 0
B. 10
C. 20
D. 30
E. 40

Attendance Question 5

- What is output by the following code?

```
int total = 0;
// assume limit is an int >= 0
for(int i = 0; i < limit; i++)
    for(int j = 0; j < limit; j++)
        total += 5;
System.out.println( total );
```

A. 5

B. limit * limit

C. limit * limit * 5

D. 0

E. limit^5

Loops That Work on a Data Set

- The number of executions of the loop depends on the length of the array, values.

```
public int total(int[] values)
{   int result = 0;
    for(int i = 0; i < values.length; i++)
        result += values[i];
    return result;
}
```

- How many many statements are executed by the above method
- N = values.length. What is T(N)? F(N)?

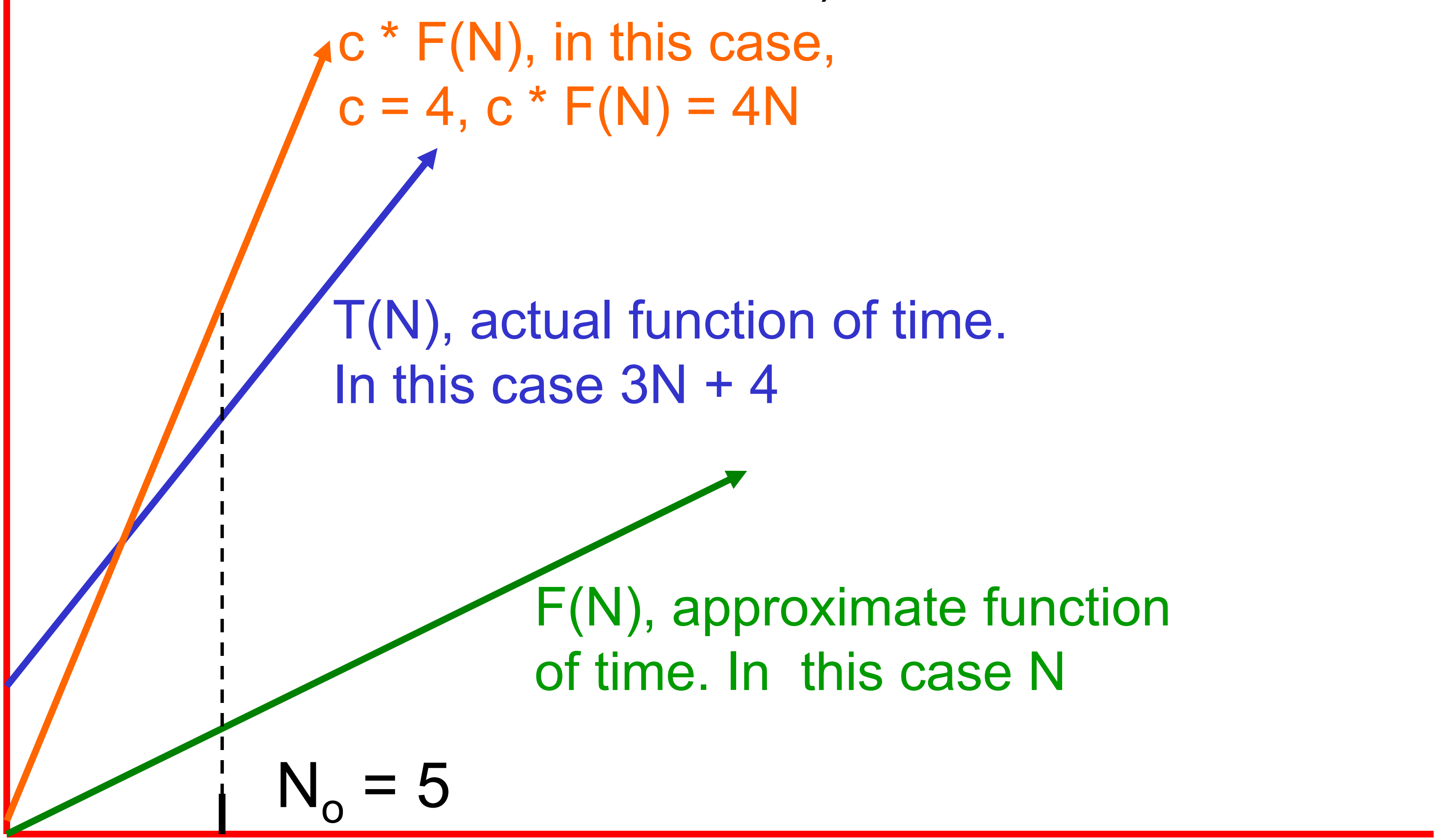
Counting Up Statements

- `int result = 0;` 1 time
- `int i = 0;` 1 time
- `i < values.length`; N + 1 times
- `i++` N times
- `result += values[i];` N times
- `return total;` 1 time
- T(N) = 3N + 4
- F(N) = N
- Big O = O(N)

Showing O(N) is Correct

- Recall the formal definition of Big O
  - T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N > N_0$
- In our case given T(N) = 3N + 4, prove the method is O(N).
  - F(N) is N
- We need to choose constants c and N_0
- how about c = 4, $N_0 = 5$?

vertical axis: time for algorithm to complete. (approximate with number of executable statements)



horizontal axis: N, number of elements in data set

Attendance Question 6

- Which of the following is true?

A. Method `total` is $O(N)$

B. Method `total` is $O(N^2)$

C. Method `total` is $O(N!)$

D. Method `total` is $O(N^N)$

E. All of the above are true

Just Count Loops, Right?

```
// assume mat is a 2d array of booleans
// assume mat is square with N rows,
// and N columns

int numThings = 0;
for(int r = row - 1; r <= row + 1; r++)
    for(int c = col - 1; c <= col + 1; c++)
        if( mat[r][c] )
            numThings++;
```

What is the order of the above code?

A. O(1) B. O(N) C. $O(N^2)$ D. $O(N^3)$ E. $O(N^{1/2})$

It is Not Just Counting Loops

```
// Second example from previous slide could be
// rewritten as follows:
int numThings = 0;
if( mat[r-1][c-1] ) numThings++;
if( mat[r-1][c] ) numThings++;
if( mat[r-1][c+1] ) numThings++;
if( mat[r][c-1] ) numThings++;
if( mat[r][c] ) numThings++;
if( mat[r][c+1] ) numThings++;
if( mat[r+1][c-1] ) numThings++;
if( mat[r+1][c] ) numThings++;
if( mat[r+1][c+1] ) numThings++;
```

Sidetrack, the logarithm

- Thanks to Dr. Math
- $3^2 = 9$
- likewise $\log_3 9 = 2$
  - "The log to the base 3 of 9 is 2."
- The way to think about log is:
  - "the log to the base x of y is the number you can raise x to to get y."
  - Say to yourself "The log is the exponent." (and say it over and over until you believe it.)
  - In CS we work with base 2 logs, a lot
- $\log_2 32 = ?$ $\log_2 8 = ?$ $\log_2 1024 = ?$ $\log_{10} 1000 = ?$

When Do Logarithms Occur

- Algorithms have a logarithmic term when they use a divide and conquer technique
- the data set keeps getting divided by 2

```
public int foo(int n)
{   // pre n > 0
    int total = 0;
    while( n > 0 )
    {   n = n / 2;
        total++;
    }
    return total;
}
```

- What is the order of the above code?

A. O(1) B. O(logN) C. O(N)

D. O(Nlog N) E. $O(N^2)$

Dealing with other methods

- What do I do about method calls?

```
double sum = 0.0;
for(int i = 0; i < n; i++)
    sum += Math.sqrt(i);
```

- Long way
  – go to that method or constructor and count statements
- Short way
  – substitute the simplified Big O function for that method.
  – if Math.sqrt is constant time, O(1), simply count `sum += Math.sqrt(i);` as one statement.

Dealing With Other Methods

```
public int foo(int[] list){
    int total = 0;
    for(int i = 0; i < list.length; i++){
        total += countDups(list[i], list);
    }
    return total;
}
// method countDups is O(N) where N is the
// length of the array it is passed
```

What is the Big O of `foo`?

A. O(1) B. O(N) C. O(NlogN)

D. O(N^2) E. O(N!)

Quantifiers on Big O

- It is often useful to discuss different cases for an algorithm
- Best Case: what is the best we can hope for?
  - least interesting
- Average Case (a.k.a. expected running time): what usually happens with the algorithm?
- Worst Case: what is the worst we can expect of the algorithm?
  - very interesting to compare this to the average case

Best, Average, Worst Case

- To Determine the best, average, and worst case Big O we must make assumptions about the data set
- Best case -> what are the properties of the data set that will lead to the fewest number of executable statements (steps in the algorithm)
- Worst case -> what are the properties of the data set that will lead to the largest number of executable statements
- Average case -> Usually this means assuming the data is randomly distributed
  - or if I ran the algorithm a large number of times with different sets of data what would the average amount of work be for those runs?

Another Example

```
public double minimum(double[] values)
{   int n = values.length;
    double minValue = values[0];
    for(int i = 1; i < n; i++)
        if(values[i] < minValue)
            minValue = values[i];
    return minValue;
}
```

- T(N)? F(N)? Big O? Best case? Worst Case? Average Case?
- If no other information, assume asking average case

Independent Loops

```
// from the Matrix class
public void scale(int factor){
    for(int r = 0; r < numRows(); r++)
        for(int c = 0; c < numCols(); c++)
            iCells[r][c] *= factor;
}
```

Assume an `numRows()` = N and `numCols()` = N.

In other words, a square Matrix.

What is the T(N)? What is the Big O?

A. O(1)
B. O(N)
C. O(NlogN)
D. O(N^2)
E. O(N!)

Significant Improvement – Algorithm with Smaller Big O function

- Problem: Given an array of ints replace any element equal to 0 with the maximum value to the right of that element.

Given:

`[0, 9, 0, 8, 0, 0, 7, 1, -1, 0, 1, 0]`

Becomes:

[<u>**9**</u>, 9, <u>**8**</u>, 8, <u>**7**</u>, <u>**7**</u>, 7, 1, -1, <u>**1**</u>, 1, 0]

Replace Zeros – Typical Solution

```
public void replace0s(int[] data){
   int max;
   for(int i = 0; i < data.length -1; i++){
      if( data[i] == 0 ){
         max = 0;
         for(int j = i+1; j<data.length;j++)
            max = Math.max(max, data[j]);
         data[i] = max;
      }
   }
}
```

Assume most values are zeros.

Example of a **dependent loops**.

Replace Zeros – Alternate Solution

```
public void replace0s(int[] data){
  int max =
     Math.max(0, data[data.length - 1]);
  int start = data.length - 2;
  for(int i = start; i >= 0; i--){
    if( data[i] == 0 )
          data[i] = max;
     else
          max = Math.max(max, data[i]);
  }
}
```

Big O of this approach?

A. O(1)

B. O(N)

C. O(NlogN)

D. $O(N^2)$

E. O(N!)

A Caveat

- What is the Big O of this statement in Java?

```
int[] list = new int[n];
```

A. O(1) B. O(N) C. O(NlogN)
D. $O(N^2)$ E. O(N!)

- Why?

Summing Executable Statements

- If an algorithms execution time is $N^2 + N$ the it is said to have $O(N^2)$ execution time not $O(N^2 + N)$
- When adding algorithmic complexities the larger value dominates
- formally a function $f(N)$ dominates a function $g(N)$ if there exists a constant value n_0 such that for all values $N > N_0$ it is the case that $g(N) < f(N)$

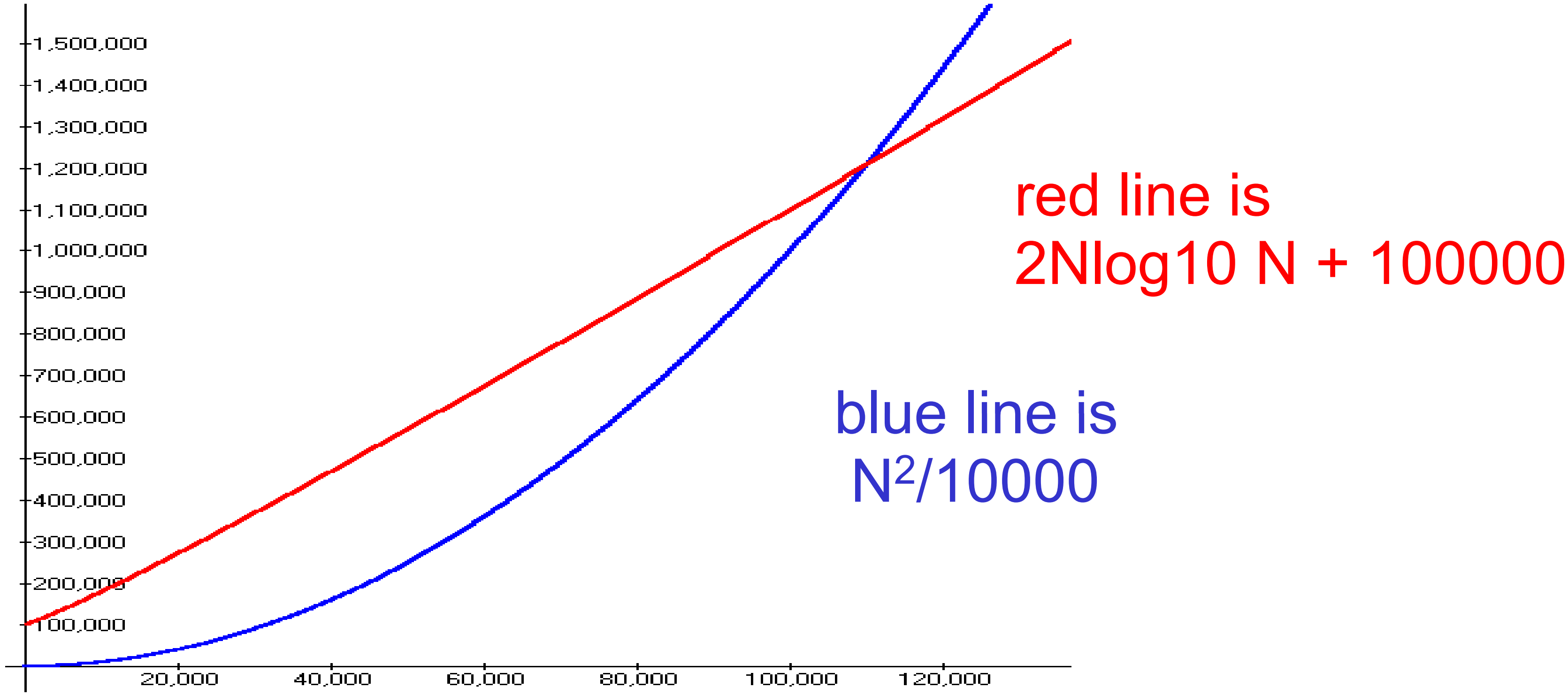
Example of Dominance

- Look at an extreme example. Assume the actual number as a function of the amount of data is:

  $$N^2/10000 + 2N\log_{10} N + 100000$$

- Is it plausible to say the N^2 term dominates even though it is divided by 10000 and that the algorithm is $O(N^2)$?
- What if we separate the equation into $(N^2/10000)$ and $(2N \log_{10} N + 100000)$ and graph the results.

Summing Execution Times



- For large values of N the N^2 term dominates so the algorithm is $O(N^2)$
- When does it make sense to use a computer?

Comparing Grades

- Assume we have a problem
- Algorithm A solves the problem correctly and is $O(N^2)$
- Algorithm B solves the same problem correctly and is $O(N \log_2 N)$
- Which algorithm is faster?
- One of the assumptions of Big O is that the data set is large.
- The "grades" should be accurate tools if this is true

Running Times

- Assume N = 100,000 and processor speed is 1,000,000,000 operations per second

Function	Running Time
2^N	3.2 x 10^{30086} years
N^4	3171 years
N^3	11.6 days
N^2	10 seconds
$N\sqrt{N}$	0.032 seconds
N log N	0.0017 seconds
N	0.0001 seconds
$\sqrt{N}$	3.2 x 10^{-7} seconds
log N	1.2 x 10^{-8} seconds

Theory to Practice OR Dykstra says: "Pictures are for the Weak."

	1000	2000	4000	8000	16000	32000	64000	128K
O(N)	2.2×10^{-5}	2.7×10^{-5}	5.4×10^{-5}	4.2×10^{-5}	6.8×10^{-5}	1.2×10^{-4}	2.3×10^{-4}	5.1×10^{-4}
O(NlogN)	8.5×10^{-5}	1.9×10^{-4}	3.7×10^{-4}	4.7×10^{-4}	1.0×10^{-3}	2.1×10^{-3}	4.6×10^{-3}	1.2×10^{-2}
$O(N^{3/2})$	3.5×10^{-5}	6.9×10^{-4}	1.7×10^{-3}	5.0×10^{-3}	1.4×10^{-2}	3.8×10^{-2}	0.11	0.30
$O(N^2)$ ind.	3.4×10^{-3}	1.4×10^{-3}	4.4×10^{-3}	0.22	0.86	3.45	13.79	(55)
$O(N^2)$ dep.	1.8×10^{-3}	7.1×10^{-3}	2.7×10^{-2}	0.11	0.43	1.73	6.90	(27.6)
$O(N^3)$	3.40	27.26	(218)	(1745) 29 min.	(13,957) 233 min	(112k) 31 hrs	(896k) 10 days	(7.2m) 80 days

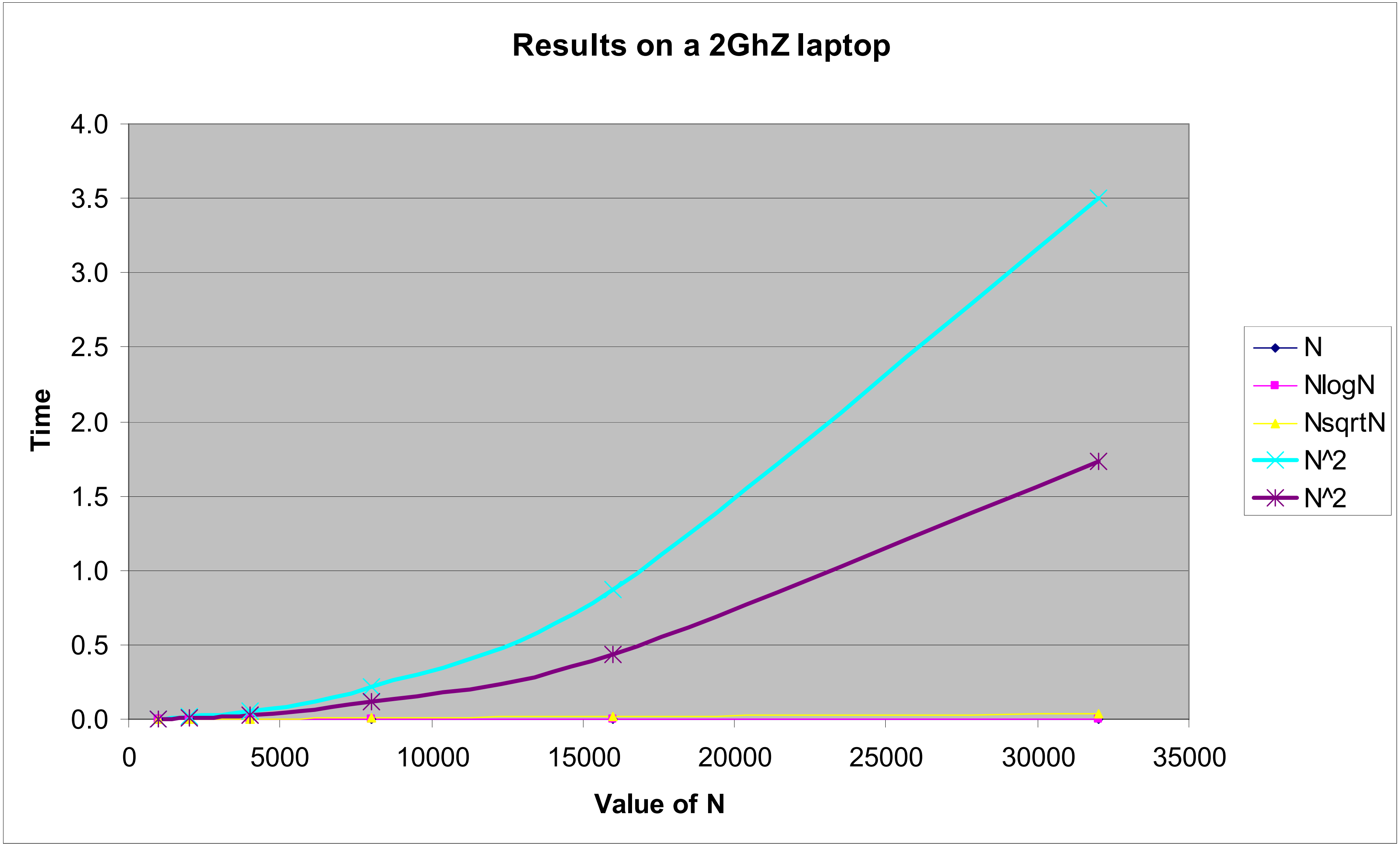
Times in Seconds. Red indicates predicated value.

Change between Data Points

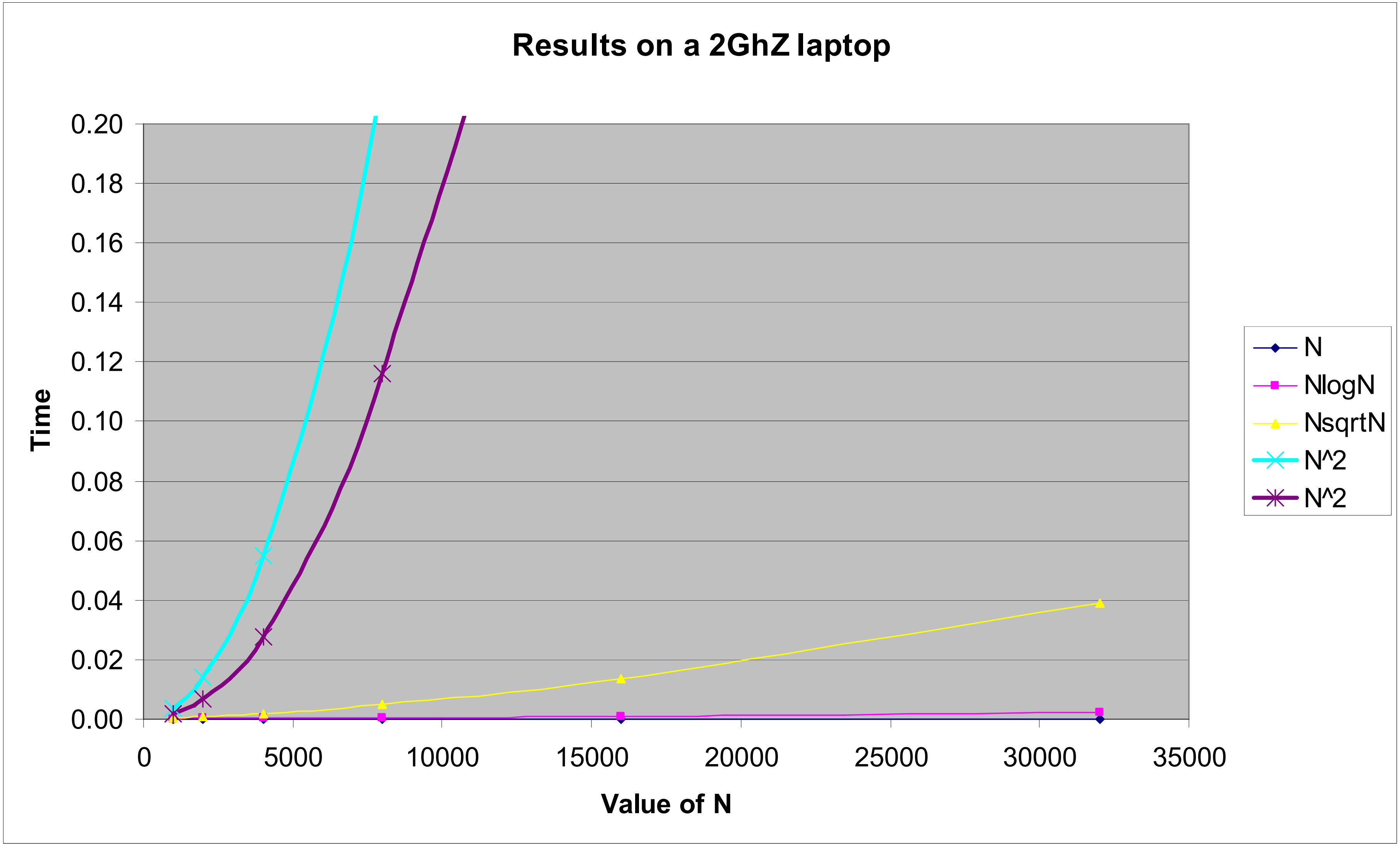
	1000	2000	4000	8000	16000	32000	64000	128K	256k	512k
O(N)	-	1.21	2.02	0.78	1.62	1.76	1.89	2.24	2.11	1.62
O(NlogN)	-	2.18	1.99	1.27	2.13	2.15	2.15	2.71	1.64	2.40
$O(N^{3/2})$	-	1.98	2.48	2.87	2.79	2.76	2.85	2.79	2.82	2.81
$O(N^2)$ ind	-	4.06	3.98	3.94	3.99	4.00	3.99	-	-	-
$O(N^2)$ dep	-	4.00	3.82	3.97	4.00	4.01	3.98	-	-	-
$O(N^3)$	-	8.03	-	-	-	-	-	-	-	-

Value obtained by $Time_x$ / $Time_{x-1}$

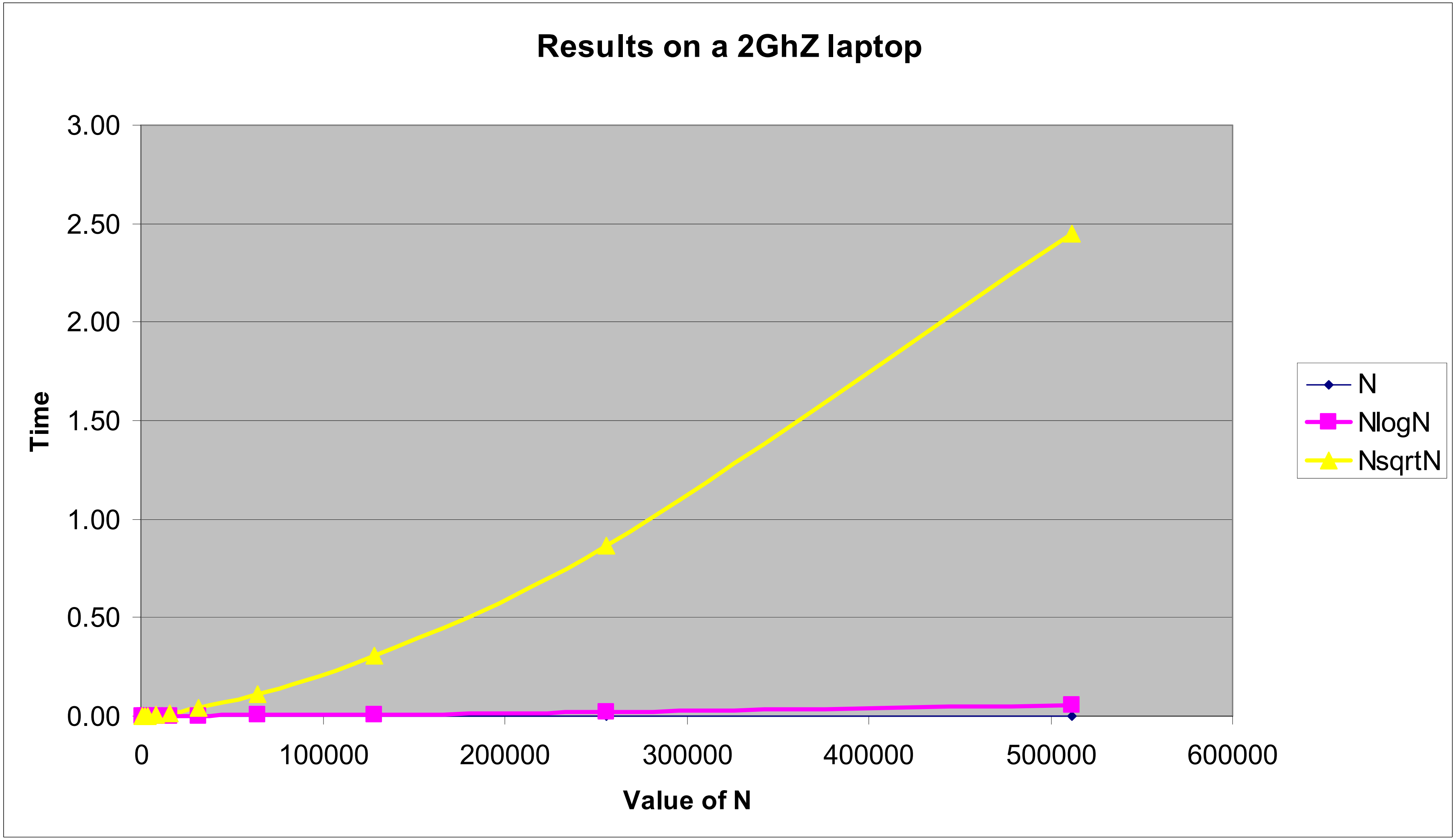
Okay, Pictures



Put a Cap on Time

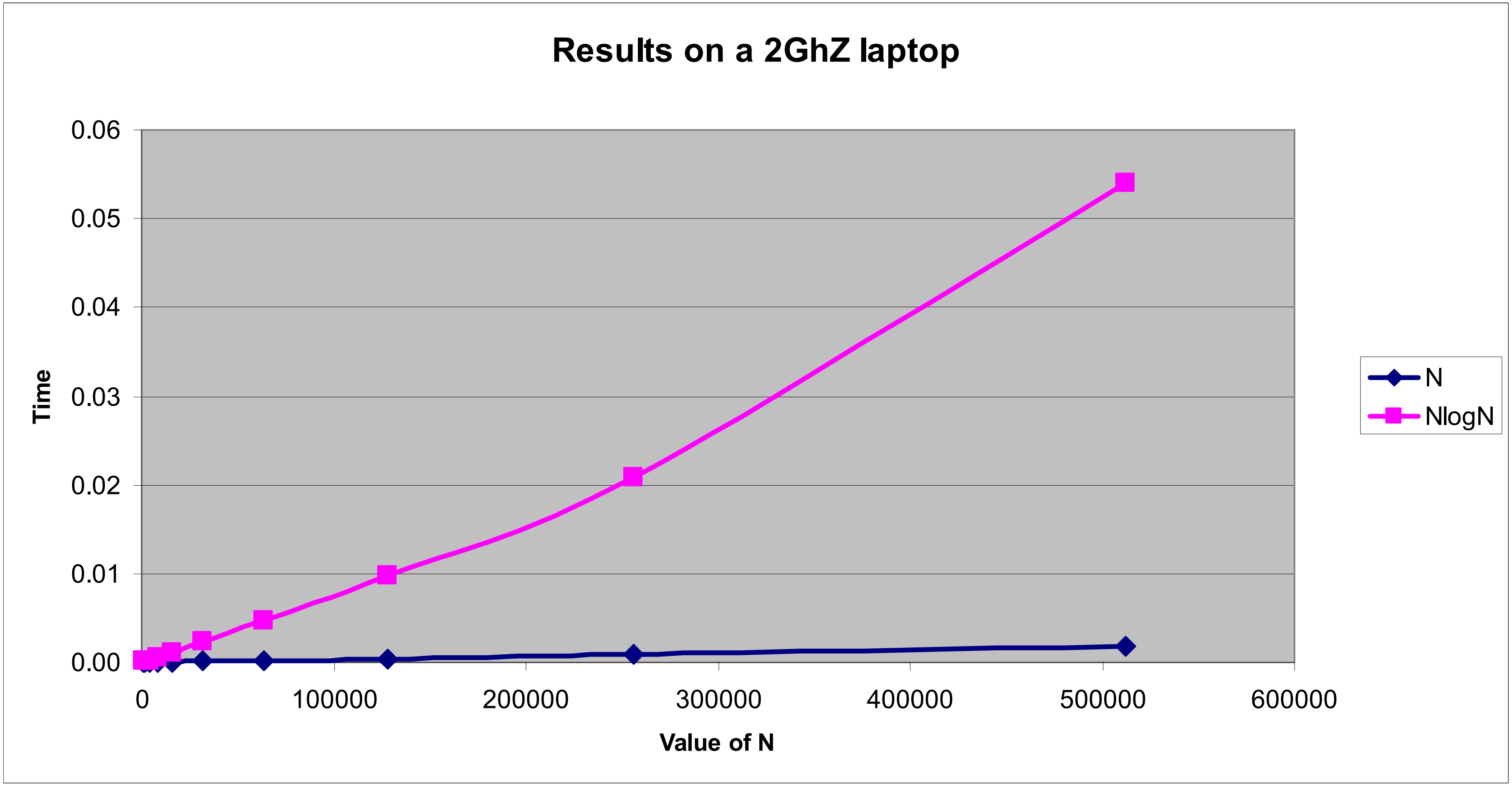


No O(N^2) Data

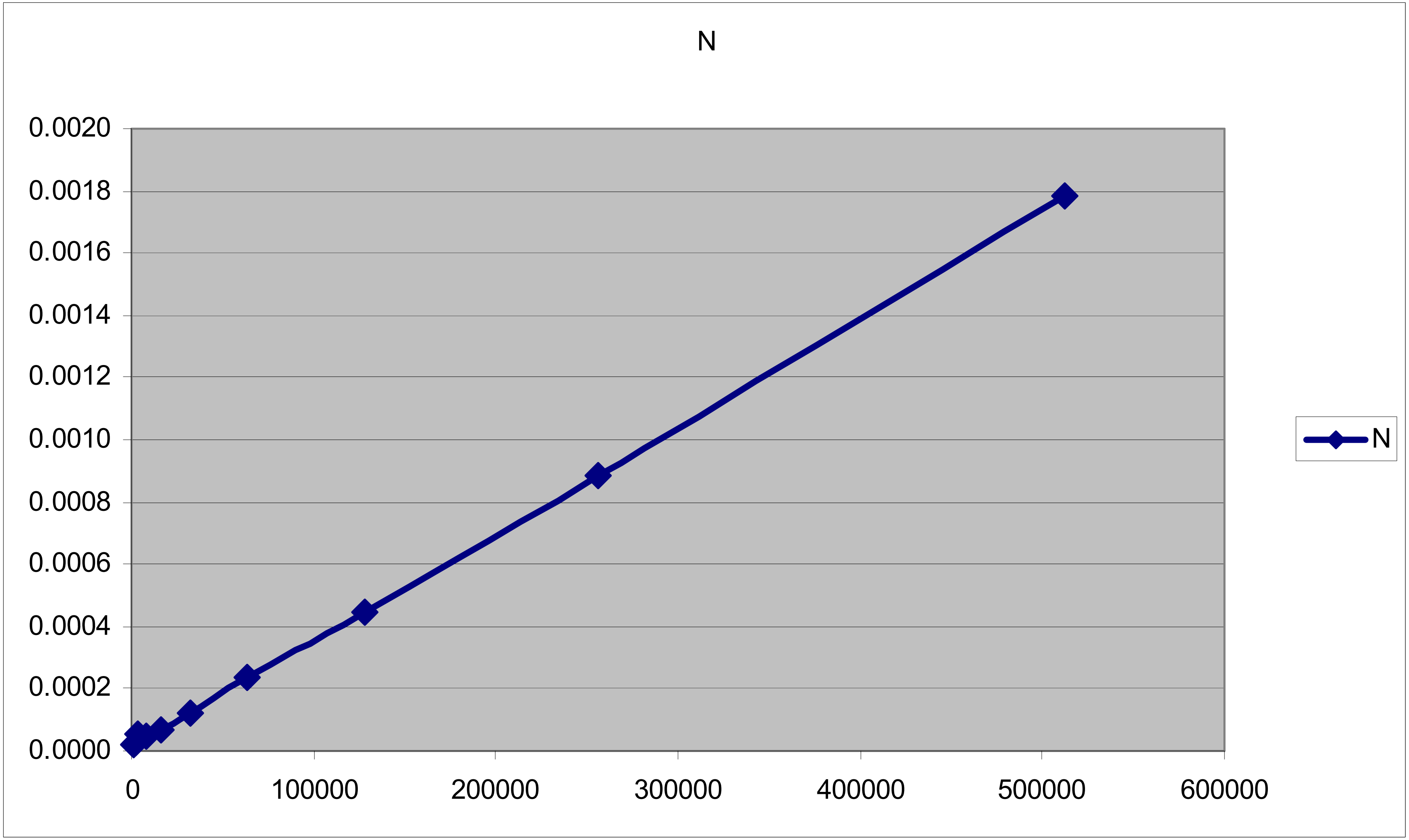


Results on a 2GhZ laptop
3.00
2.50
2.00
1.50
1.00
0.50
0.00
Time
0
100000
200000
300000
400000
500000
600000
Value of N
N
NlogN
NsqrtN

Just O(N) and O(NlogN)



Just O(N)



N
0.0020
0.0018
0.0016
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
0
100000
200000
300000
400000
500000
600000
N

Reasoning about algorithms

- We have an O(N) algorithm,
  - For 5,000 elements takes 3.2 seconds
  - For 10,000 elements takes 6.4 seconds
  - For 15,000 elements takes ….?
  - For 20,000 elements takes ….?

- We have an $O(N^2)$ algorithm
  - For 5,000 elements takes 2.4 seconds
  - For 10,000 elements takes 9.6 seconds
  - For 15,000 elements takes …?
  - For 20,000 elements takes …?

A Useful Proportion

- Since F(N) is characterizes the running time of an algorithm the following proportion should hold true:

$F(N_0) / F(N_1) \sim= time_0 / time_1$

- An algorithm that is $O(N^2)$ takes 3 seconds to run given 10,000 pieces of data.
  - How long do you expect it to take when there are 30,000 pieces of data?
  - common mistake
  - logarithms?

10^9 instructions/sec, runtimes

N	*O(log N)*	*O(N)*	*O(N log N)*	*O(N²)*
10	0.000000003	0.00000001	0.000000033	0.0000001
100	0.000000007	0.00000010	0.000000664	0.0001000
1,000	0.000000010	0.00000100	0.000010000	0.001
10,000	0.000000013	0.00001000	0.000132900	0.1 min
100,000	0.000000017	0.00010000	0.001661000	10 seconds
1,000,000	0.000000020	0.001	0.0199	16.7 minutes
1,000,000,000	0.000000030	1.0 second	30 seconds	31.7 years

Why Use Big O?

- As we build data structures Big O is the tool we will use to decide under what conditions one data structure is better than another
- Think about performance when there is a lot of data.
  - "It worked so well with small data sets..."
  - Joel Spolsky, Schlemiel the painter's Algorithm
- Lots of trade offs
  - some data structures good for certain types of problems, bad for other types
  - often able to trade SPACE for TIME.
  - Faster solution that uses more space
  - Slower solution that uses less space

Big O Space

- Less frequent in early analysis, but just as important are the space requirements.
- Big O could be used to specify how much space is needed for a particular algorithm

Formal Definition of Big O (repeated)

- T(N) is O(F(N)) if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$
  - N is the size of the data set the algorithm works on
  - T(N) is a function that characterizes the *actual* running time of the algorithm
  - F(N) is a function that characterizes an upper bounds on T(N). It is a limit on the running time of the algorithm
  - c and N_0 are constants

More on the Formal Definition

- There is a point N_0 such that for all values of N that are past this point, T(N) is bounded by some multiple of F(N)
- Thus if T(N) of the algorithm is O(N^2) then, ignoring constants, at some point we can *bound* the running time by a quadratic function.
- given a *linear* algorithm it is *technically correct* to say the running time is O(N ^ 2). O(N) is a more precise answer as to the Big O of the linear algorithm
  - thus the caveat "pick the most restrictive function" in Big O type questions.

What it All Means

- T(N) is the actual growth rate of the algorithm
  - can be equated to the number of executable statements in a program or chunk of code
- F(N) is the function that bounds the growth rate
  - may be upper or lower bound
- T(N) may not necessarily equal F(N)
  - constants and lesser terms ignored because it is a *bounding function*

Other Algorithmic Analysis Tools

- *Big Omega* T(N) is Ω(F(N)) if there are positive constants c and N_0 such that T(N) $\geq$ cF(N)) when N $\geq$ N_0
  - Big O is similar to less than or equal, an upper bounds
  - Big Omega is similar to greater than or equal, a lower bound
- *Big Theta* T(N) is θ(F(N)) if and only if T(N) is O(F(N))and T(N) is Ω(F(N)).
  - Big Theta is similar to equals

Relative Rates of Growth

Analysis Type	Mathematical Expression	Relative Rates of Growth
Big O	$T(N) = O(F(N))$	$T(N) \le F(N)$
Big Ω	$T(N) = \Omega(F(N))$	$T(N) \ge F(N)$
Big θ	$T(N) = \theta(F(N))$	$T(N) = F(N)$

"In spite of the additional precision offered by Big Theta, Big O is more commonly used, except by researchers in the algorithms analysis field" - Mark Weiss