Computer Science

Advanced Computer Science Topics

Mike Scott

Contest Director

For all coaches and contestants.

University Interscholastic League

Topics

- Interesting Examples of CS
  - intersection control
  - robot soccer
  - suspended particle explosions
- Algorithm Analysis and Big O
- Anything you want to cover

A Brief Look at Computer Science

- The UIL CS contest emphasizes programming
- Most introductory CS classes, both at the high school and college level, teach programming
- … and yet, computer science and computer programming are not the same thing!
- So what is Computer Science?

What is Computer Science?

- Poorly named in the first place.
- It is not so much about the computer as it is about *Computation.*
- *"Computer Science is more the study of managing and processing information than it is the study of computers."*

  -Owen Astrachan, Duke University

So why Study Programming?

- Generally the first thing that is studied in Chemistry is stoichiometry.
  - Why? It is a skill necessary in order to study more advanced topics in Chemistry
- The same is true of programming and computer science.

What do Computer Scientists do?

- Computer Scientists solve problems
  - creation of algorithms
- Three examples
  - Kurt Dresner, Intersection Control
  - Austin Villa, Robot Soccer
  - Okan Arikan, playing with fire

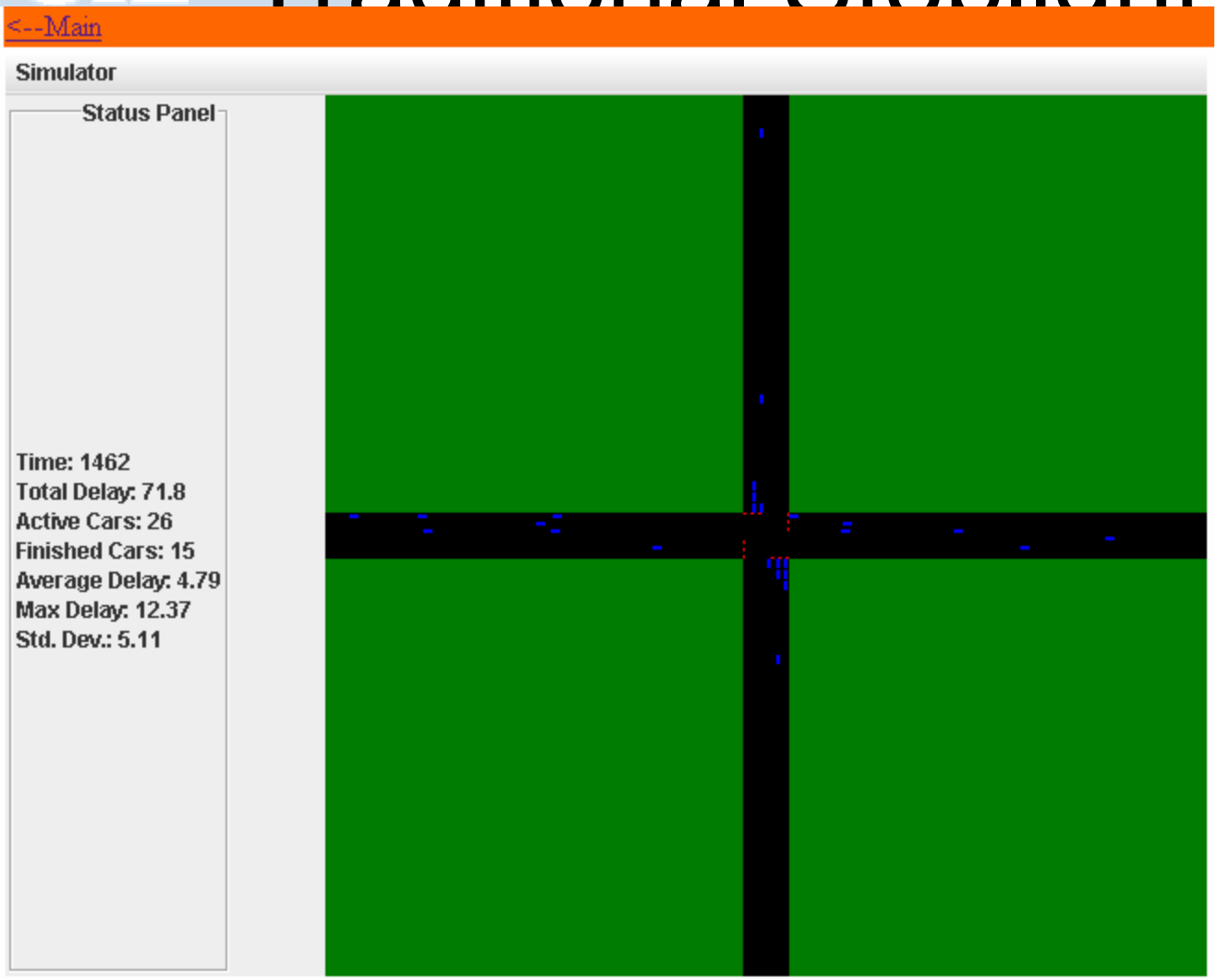
What do Computer Scientists do?

- Computer Scientists solve problems
  - creation of algorithms
- Three examples
  - Kurt Dresner, Intersection Control
  - Austin Villa, Robot Soccer
  - Okan Arikan, playing with fire

Kurt Dresner – Intersection Control

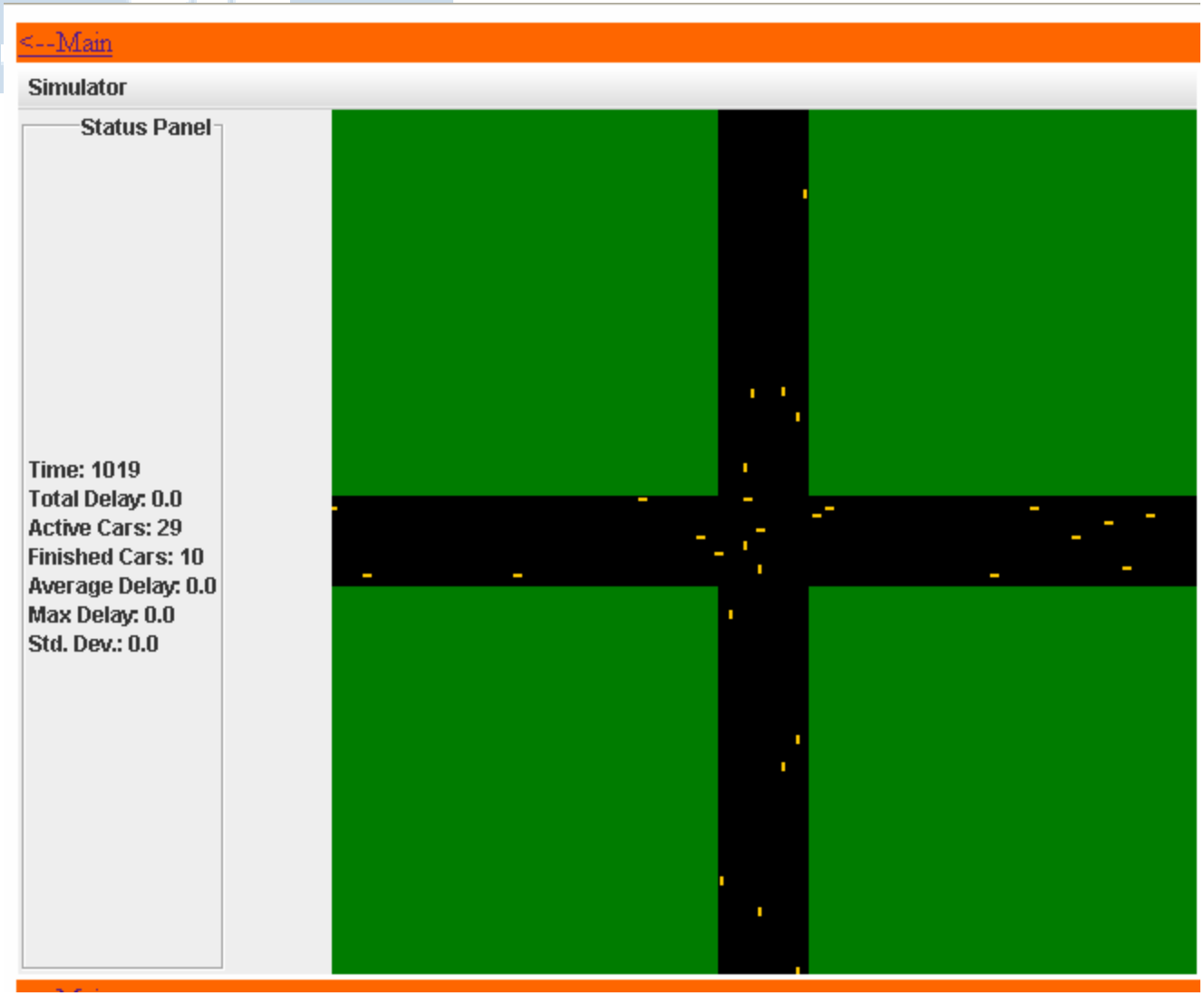
- PhD student in UTCS department
- area of interest artificial intelligence
- ***Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism***
  - how will intersections work if and when cars are autonomous?

Traditional Stoplight



stop sign

Reservation System



3 lanes

6 lanes

Austin Villa – Robot Soccer

- Multiple Autonomous Agents
- Get a bunch of Sony Aibo robots to play soccer
- Problems:
  - *vision* (is that the ball?)
  - *localization* (Where am I?)
  - *locomotion* (I want to be there!)
  - *coordination* (I am open! pass me the ball!)
- Video

Okan Arikan – Playing with Fire

- There are some things in computer graphics that are "hard"
  - fire, water, hair, smoke
  - "Animating Suspended Particle Explosions"
  - "Pushing People Around"

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

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

University Interscholastic League

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

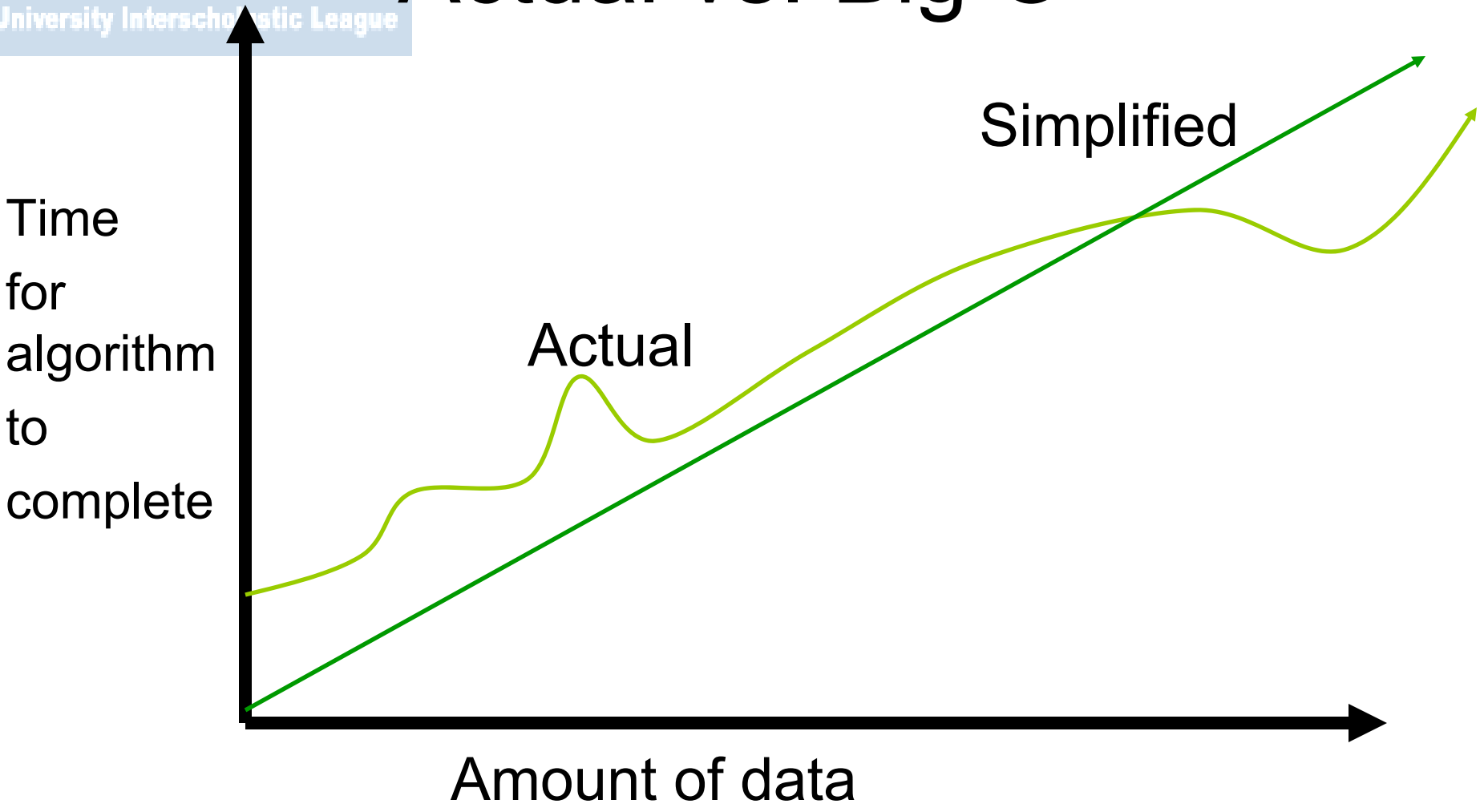
Typical Functions Big O Functions

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.
- 4N2 + 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 funciton.
  - 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

University Interscholastic League

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*

University Interscholastic League

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
- 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

- 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)
```

Convenience Loops

```
// mat is a 2d array of booleans
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++;
```

This piece of code turn out to be constant time not O(N^2).

It is Not Just Counting Loops

```
// 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++;
```

Dealing with Other Methods

```
// Card.Clubs = 2, Card.Spades = 4
// Card.TWO = 0. Card.ACE = 12
for(int suit = Card.CLUBS; suit <= Card.SPADES; suit++)
{    for(int value = Card.TWO; value <= Card.ACE; value++)
     {    myCards[cardNum] = new Card(value, suit);
          cardNum++;
     }
}
```

How many statement is

```
myCards[cardNum] = new Card(value, suit);
```

???

Dealing with other methods

- What do I do about the method call `Card(value, suit)` ?
- Long way
  - go to that method or constructor and count statements
- Short way
  - substitute the simplified Big O function for that method.
  - if Card(int, int) is constant time, O(1), simply count that as 1 statement.

Loops That Work on a Data Set

- Loops like the previous slide are *fairly* rare
- Normally loop operates on a data set which can vary is size
  - 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 total;
}
```

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

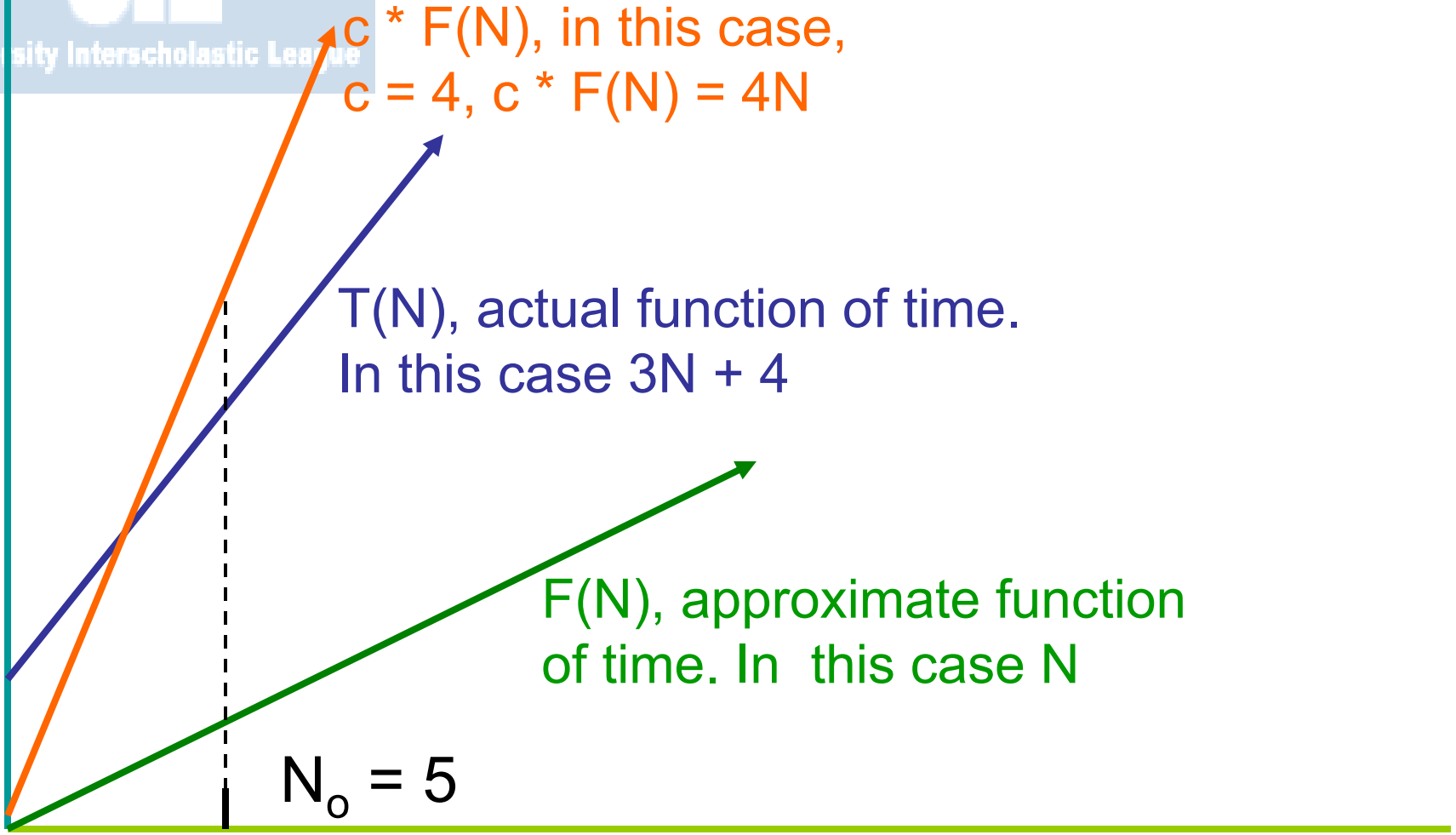
Counting Up Statements

- `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 $T(N) \leq cF(N)$
- 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

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)
  - 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;
}
```

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: 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

Nested Loops

```
public Matrix add(Matrix rhs)
{  Matrix sum = new Matrix(numRows(), numCols(), 0);
   for(int row = 0; row < numRows(); row++)
      for(int col = 0; col < numCols(); col++)
         sum.myMatrix[row][col] = myMatrix[row][col]
            + rhs.myMatrix[row][col];
   return sum;
}
```

- Number of executable statements, T(N)?
- Appropriate F(N)?
- Big O?

Another Nested Loops Example

```
public void selectionSort(double[] data)
{  int n = data.length;
   int min;
   double temp;
   for(int i = 0; i < n; i++)
   {  min = i;
      for(int j = i+1; j < n; j++)
         if(data[j] < data[min])
            min = j;
      temp = data[i];
      data[i] = data[min];
      data[min] = temp;
   }// end of outer loop, i
}
```

- Number of statements executed, T(N)?

Some helpful mathematics

- 1 + 2 + 3 + 4 + … + N
  - $N(N+1)/2 = N^2/2 + N/2$ is $O(N^2)$
- N + N + N + …. + N (total of N times)
  - $N*N = N^2$ which is $O(N^2)$
- $1 + 2 + 4 + \ldots + 2^N$
  - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

One More Example

```
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?

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 $g(N) < f(N)$

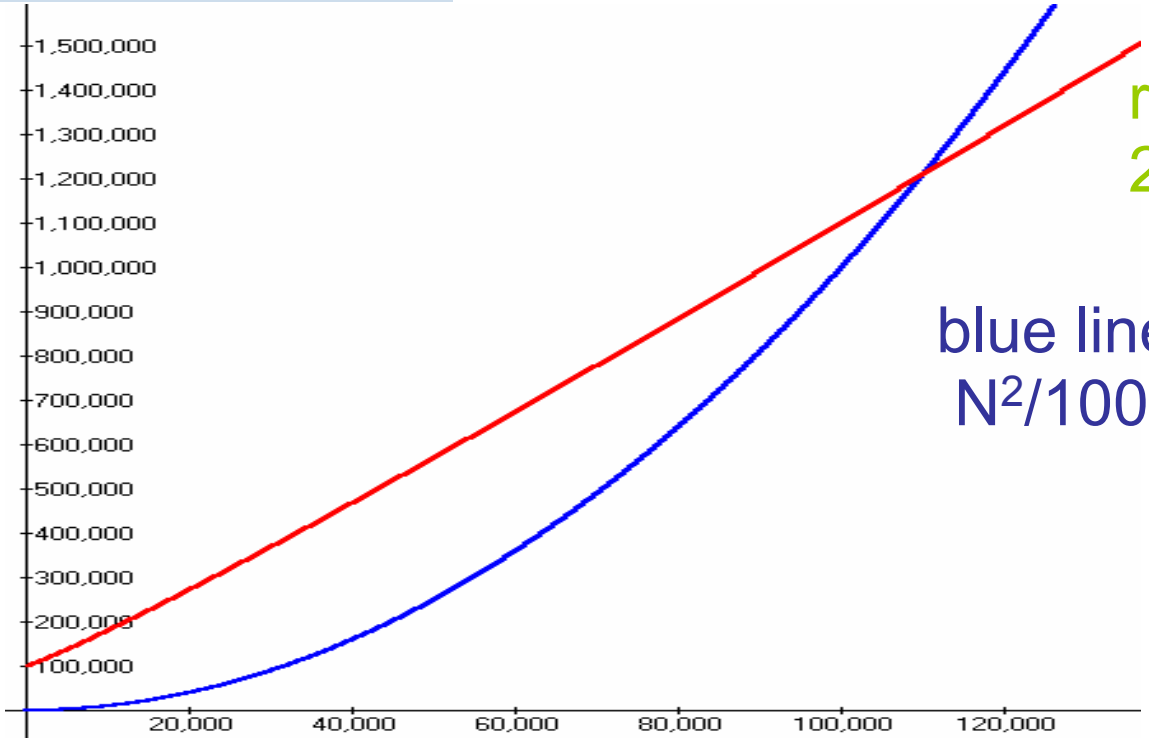
Example of Dominance

- 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



red line is
2Nlog10 N + 100000

blue line is
$N^2/10000$

- 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 to be solved
- 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	3.2×10^{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
$\sqrt{\ N}$	3.2×10^{-7} seconds
log N	1.2×10^{-8} seconds

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 …?

10^9 instructions/sec, runtimes

N	*O(log N)*	*O(N)*	*O(N log N)*	*O(N^2)*
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

When to Teach Big O?

- In a second programming course (like APCS AB) curriculum do it early!
- A valuable tool for reasoning about data structures and which implementation is better for certain operations
- Don't memorize things!
  - ArrayList `add(int index, Object x)` is O(N) where N is the number of elements in the ArrayList
  - If you implement an array based list and write similar code you will learn and remember WHY it is O(N)

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))	$T(N) \leq F(N)$
Big Ω	T(N) = Ω(F(N))	$T(N) \geq F(N)$
Big θ	T(N) = θ(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

University Interscholastic League

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