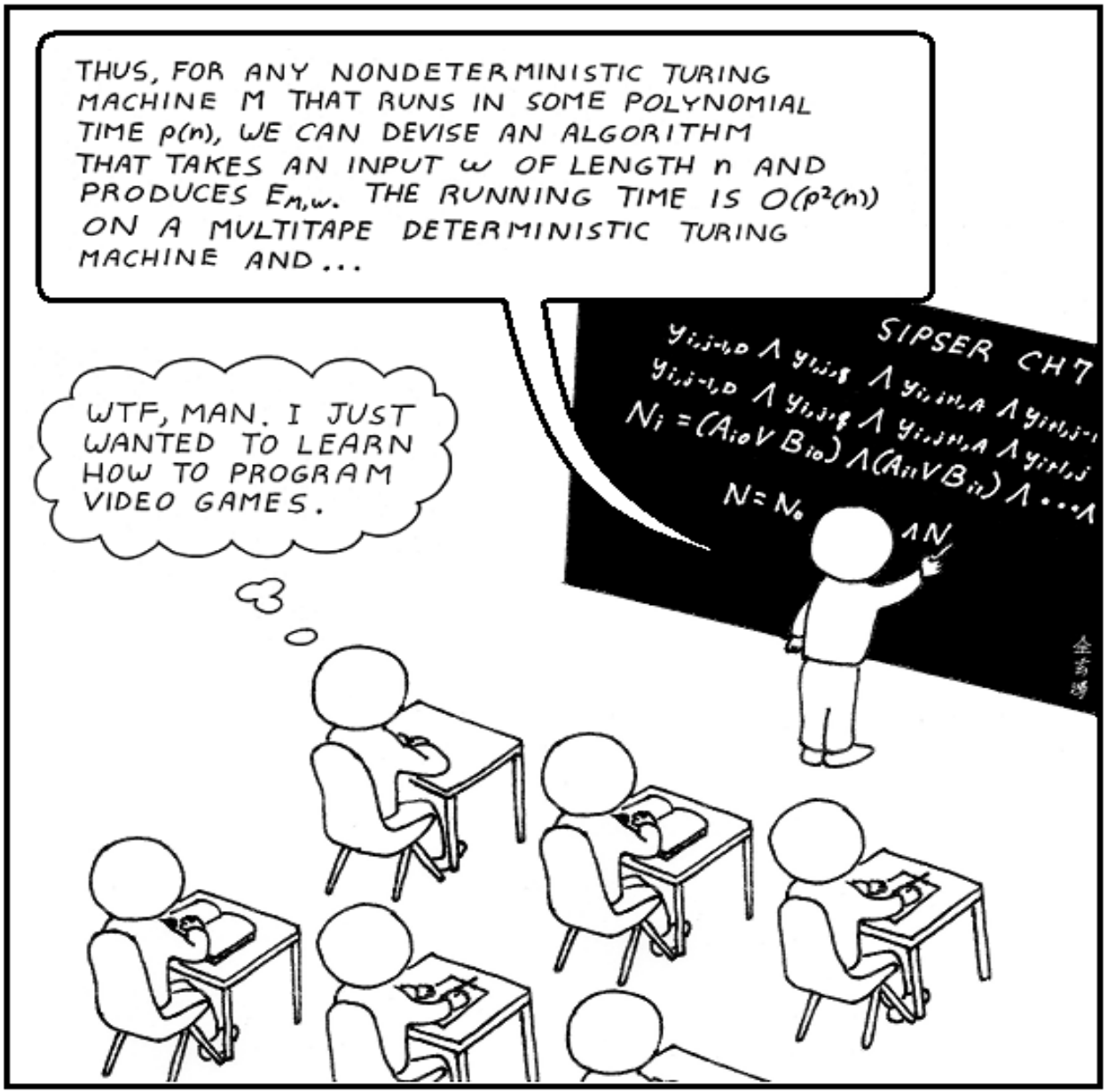
Algorithm Analysis



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

Efficiency

- Computer Scientists don't just write programs.
- They also *analyze* them.
- How efficient is a program?
  - How much time does it take program to complete?
  - How much memory does a program use?
  - How do these change as the amount of data changes?

Efficiency

- Computer Scientists don't just write programs.
- They also *analyze* them.
- How efficient is a program?
  - How much time does it take program to complete?
  - How much memory does a program use?
  - How do these change as the amount of data changes?

Question 2

- What is output by the following code?

```
int total = 0;
for(int i = 0; i < 13; i++)
    for(int j = 0; j < 11; j++)
        total += 2;
System.out.println( total );
```

A. 24

B. 120

C. 143

D. 286

E. 338

Question 3

- What is output when method `sample` is called?

```
public static void sample(int n, int m) {
      int total = 0;
      for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                  total += 5;
      System.out.println( total );
}
```

A. 5

B. n * m

C. n * m * 5

D. n^m

E. $(n * m)^5$

Example

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

- How many statements are executed by method `total` as a function of `values.length`?
- Let n = `values.length`
  - ***n*** is commonly used as a variable that denotes the amount of data

Counting Statements

```
int x; // one statement
x = 12; // one statement
int y = z * x + 3 % 5 * x / i; // 1
x++; // one statement
boolean p = x < y && y % 2 == 0 ||
                    z >= y * x; // 1
int[] list = new int[100]; // 100
list[0] = x * x + y * y; // 1
```

Counting Up Statements

- `int result = 0;` *1*
- `int i = 0;` *1*
- `i < values.length;` *n + 1*
- `i++;` *n*
- `result += values[i];` *n*
- `return total;` *+ 1*

- T(n) = *3n + 4*
- T(n) is the number of executable statements in method `total` as function of `values.length` = n

Another Simplification

- When determining complexity of an algorithm we want to simplify things
  - hide some details to make comparisons easier
- Like assigning your grade for course
  - At the end of CS221, your transcript won't list all the details of your performance in the course
  - it won't list scores on all assignments, quizzes, and tests
  - simply a letter grade, B- or A or D+
- So we focus on the dominant term from the function and ignore the coefficient

Big O

- The most common method and notation for discussing the execution time of algorithms is *Big O*, also spoken *Order*
- 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

Formal Definition of Big O

- T(n) is O(g(n)) if there are positive constants c and n_0 such that $T(n) \leq c*g(n)$ 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
  - g(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
- g(n) is the function that bounds the growth rate
  - may be upper or lower bound
- T(n) may not necessarily equal g(n)
  - constants and lesser terms ignored because it is a *bounding function*

Big-Oh Rules

- If T(n) is a sum of several terms:
  - Keep the one with the largest growth rate
  - Omit all others
  - If $T(n) = n^2 + n + 1$, then $T(n) = O(n^2)$
- If T(n) is a product of several factors:
  - Omit the constants terms in the product that do not depend on n
  - If T(n) = 3n, then T(n) = O(n).

Typical 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

O(1) – Constant Time

- ***O(1)*** describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
Boolean isFirstElementNull(String[] elements)
{
      return elements[0] == null;
}
```

O(log n) – Logarithmic Time

- Iterative halving of data sets
  - Binary Search
    - produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase.
    - doubling the size of the input data set has little effect on its growth.
    - after a single iteration, the data set will be halved and therefore on a par with an input data set half the size.
  - Algorithms like Binary Search extremely efficient when dealing with large data sets.

Binary Search

```
int binary_search(int A[], int key, int min, int max) {
        // test if array is empty
        if (max < min)
        // set is empty, so return value showing not found
                return KEY_NOT_FOUND;
        else
        {
                // calculate midpoint to cut set in half
                int mid = midpoint(min, max);
                // three-way comparison
                if (A[mid] > key)
                    // key is in lower subset
                    return binary_search(A, key, min, mid - 1);
                else if (A[mid] < key)
                    // key is in upper subset
                    return binary_search(A, key, mid + 1, max);
                else
                    // key has been found
                    return mid;
        }
}
```

O(n) – Linear Time

- ***O(n)*** describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
Boolean containsValue(String[] elements, string value, int size)
{
            for(int i = 0; i < size; i++)
            {
                  if (element == value)
                        return true;
            }
            return false;
}
```

$O(n^2)$ – Quadratic Time

- ***O(n²)*** represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- This is common with algorithms that involve nested iterations over the data set.
- Deeper nested iterations will result in ***O(n³)***, ***O(n⁴)*** etc.

Quadratic Time Example

```
public void printGrid(int n)
{
      for ( int i = 0 ; i < n; i++ )
      {
            // PRINT a row
            for ( int i = 0 ; i < n; i++ )
            {
                  System.out.print( "*" ) ;
            }
            // PRINT newline
            System.out.println( " ") ;
      }
}
```

$O(2^n)$ – Exponential Time

- ***$O(2^n)$*** denotes an algorithm whose growth doubles with each addition to the input data set.

```
public int Fibonacci(int number)
{
    if (number <= 1)
          return number;
    else
          return Fibonacci(number - 2) +
                              Fibonacci(number - 1);
}
```

Question 4

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

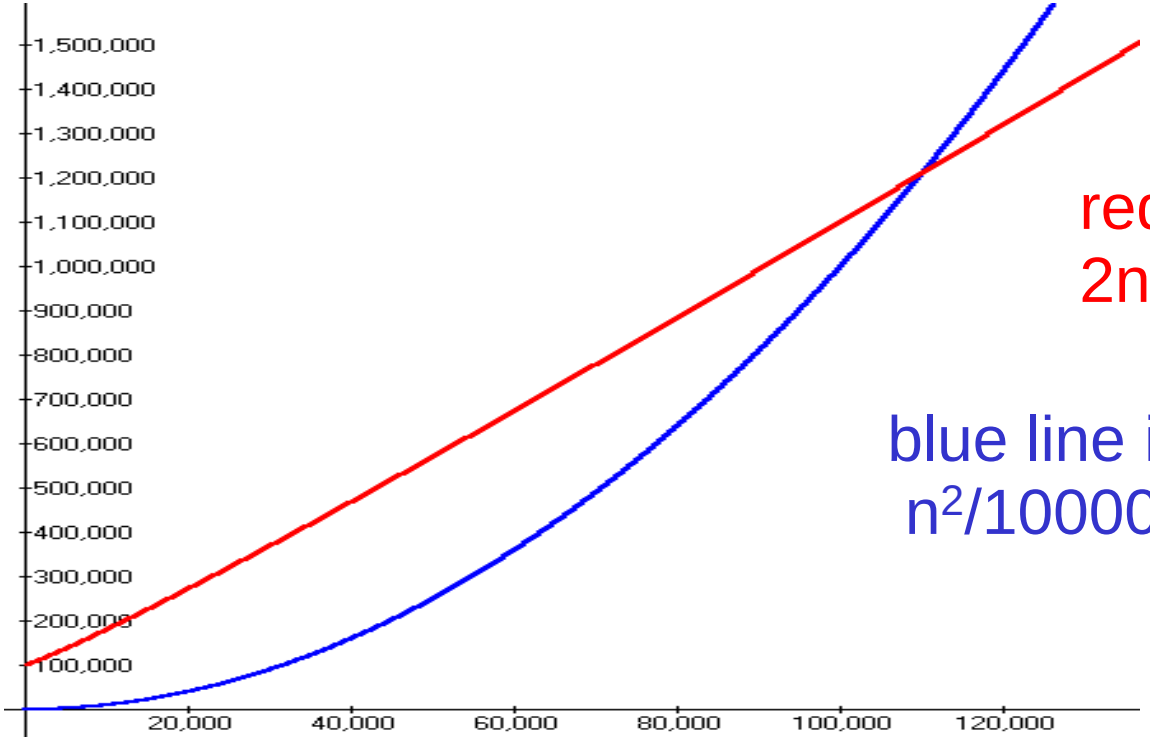
Example of Dominance

- 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(n²)***?
- 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
$2n \log_{10} n + 100000$

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
- Algorithm A solves the problem correctly and is ***O(n²)***
- Algorithm B solves the same problem correctly and is ***O(n log(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

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++;
```

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(n log n)

D. $O(n^2)$ E. O(n!)

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;
}
```

First, assume *numRows* = *n* and *numCols* = *n* (a square Matrix).

Then assume *numRows* = *numCols* = *n.*

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

A. O(1) B. O(n) C. O(n log n)

D. $O(n^2)$ E. O(n!)

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

- Big O could be used to specify how much space is needed for a particular algorithm
  - in other words how many variables are needed
- Often there is a *time – space tradeoff*
  - can often take less time if willing to use more memory
  - can often use less memory if willing to take longer
  - truly beautiful solutions take less time and space

The biggest difference between time and space is that you can't reuse time. - Merrick Furst

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

Another Example

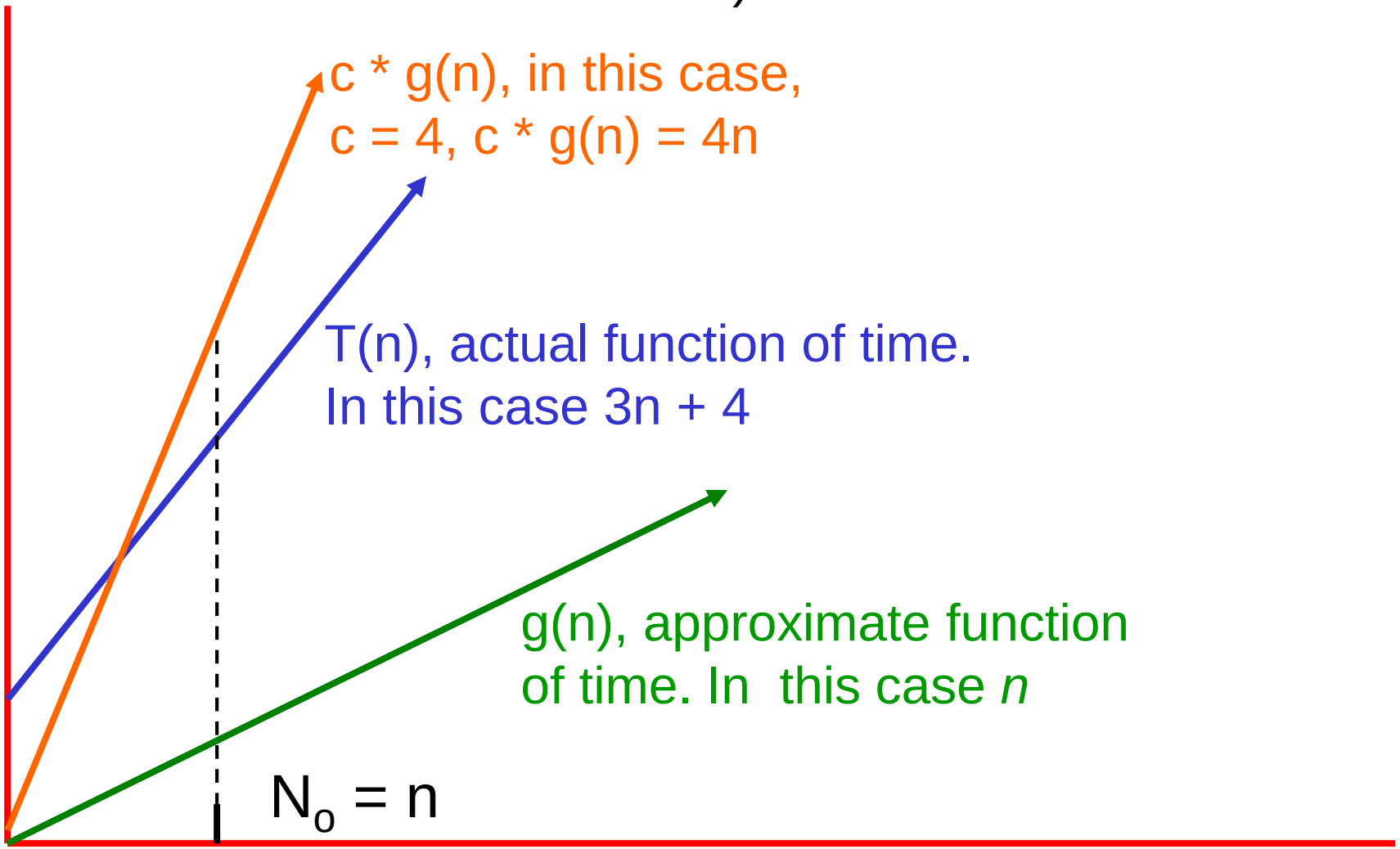
```
public double minimum(double[] values)
{  int n = values.length;
   double minValue = values[0];
   for(int i = 1; i < n; i++)
      if(values[i] < minValue)
         minValue = values[i];
   return minValue;
}
```

- T(n)? g(n)? Big O? Best case? Worst Case? Average Case?
- If no other information, assume asking worst case

Showing O(n) is Correct

- Recall the formal definition of Big O
  - T(n) is O(g(n)) if there are positive constants c and n_0 such that $T(n) \leq c*g(n)$ when $n > n_0$
- Recall method `total`, $T(n) = 3n + 4$
  - show method `total` is O(n).
  - g(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. (simplified to number of executable statements)



horizontal axis: n, number of elements in data set

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