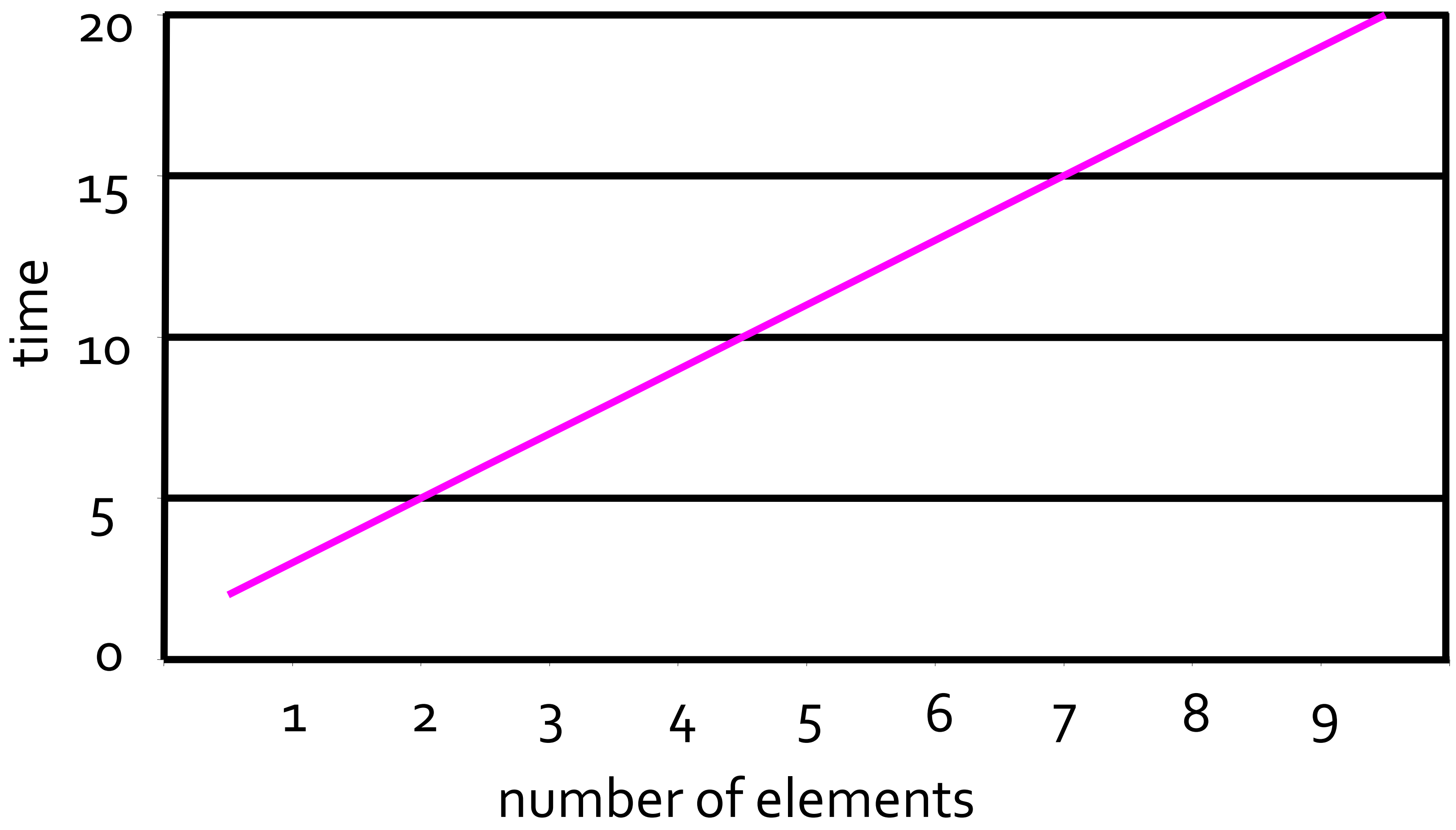
TIME COMPLEXITY – PART 1

Algorithm Behavior

- If an algorithm works with varying amounts of data each time it runs, we would normally expect that
  - When working with a large amount of data (in an array, for example), the algorithm would take longer to complete execution
  - When working with a small amount of data, the algorithm would complete execution more quickly

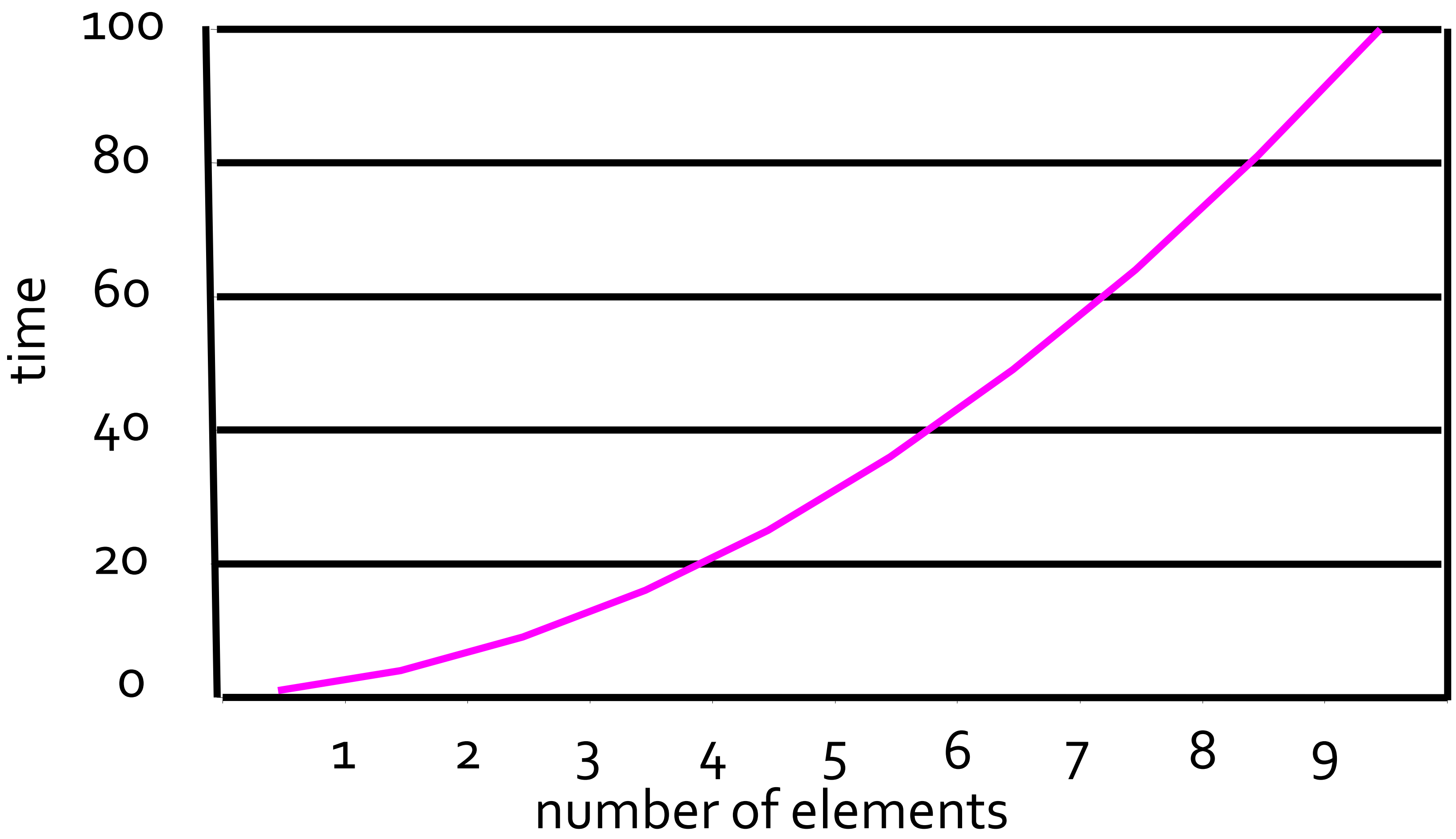
Example: Possible Algorithm Behavior



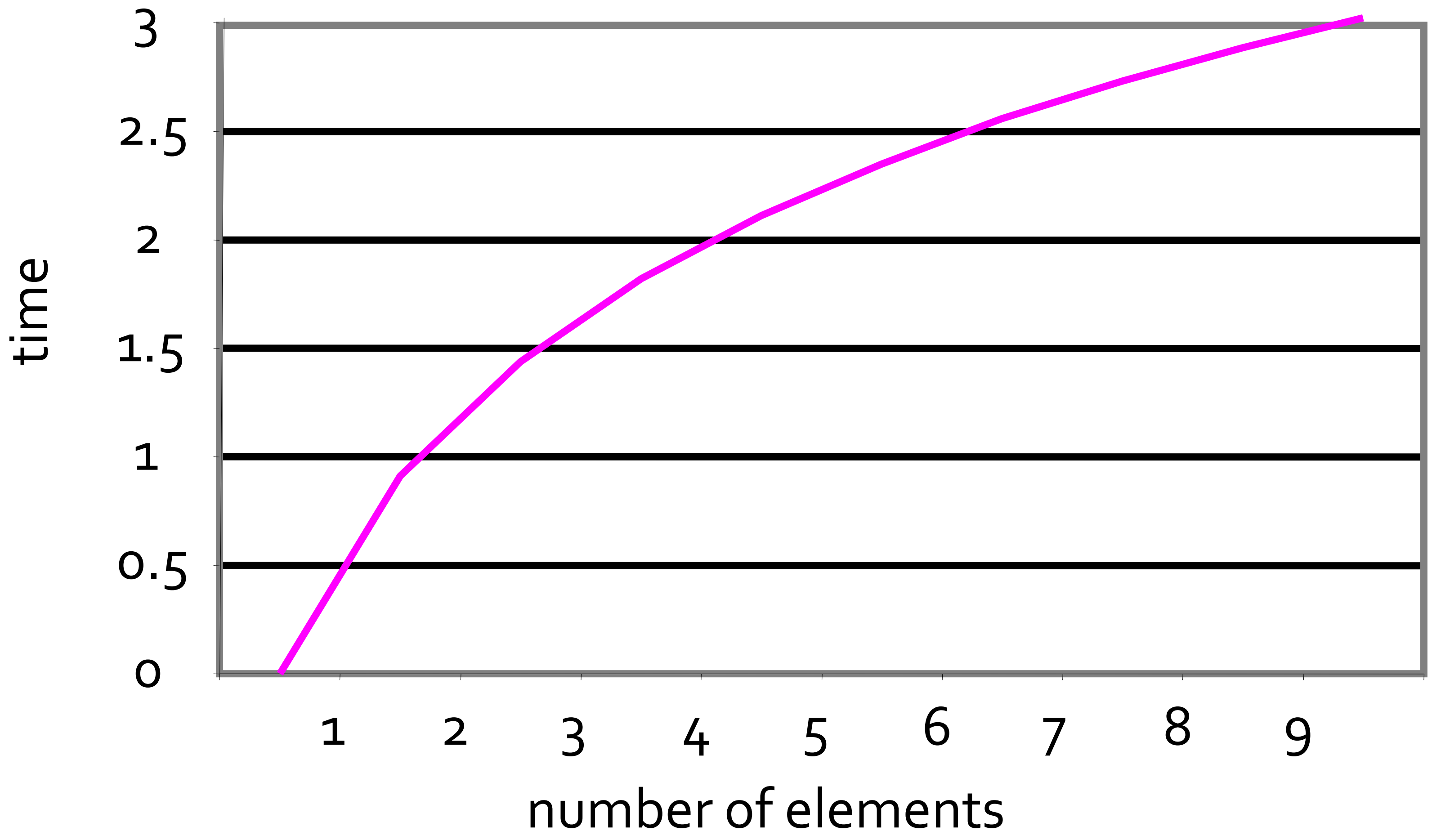
Different Behaviors

- However, an algorithm's behavior, as the number of elements is varied, does not always produce a nice straight line…

Different Behaviors (cont.)



Different Behaviors (cont.)



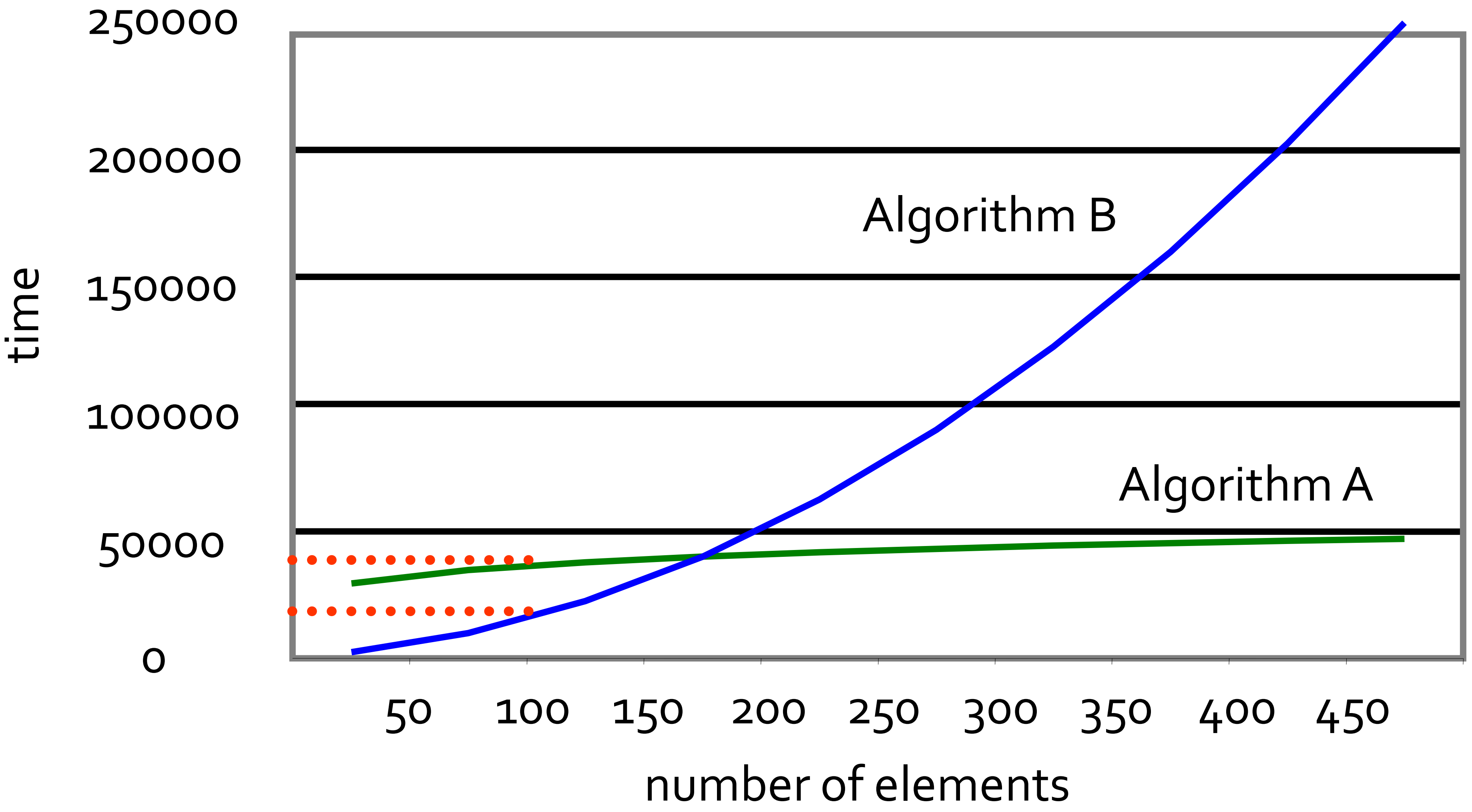
Time Complexities

- The different behaviors that you see actually represent different ***time complexities***
- Time complexities are used to help make an intelligent decision about which algorithm to use, when two different algorithms accomplish the same task

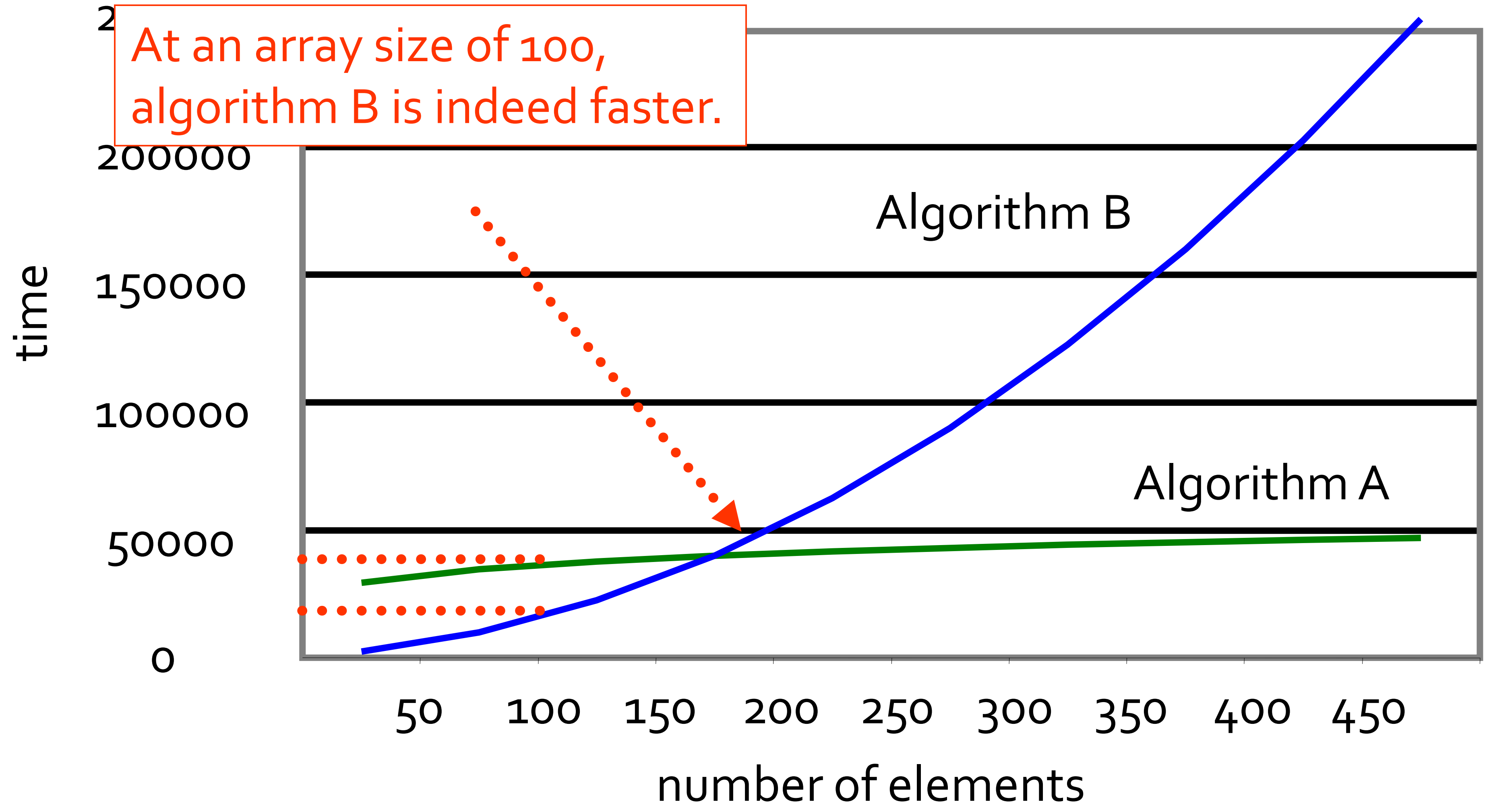
An Example

- A decision needs to be made about which algorithm to use, algorithm A or algorithm B
- They both accomplish the same task, but they use radically different methods to achieve the same result
  - One mountain top, but many ways up
- They are timed, using millions of trials automated on a computer, and an array size of 100
- Algorithm B turned out to be faster.

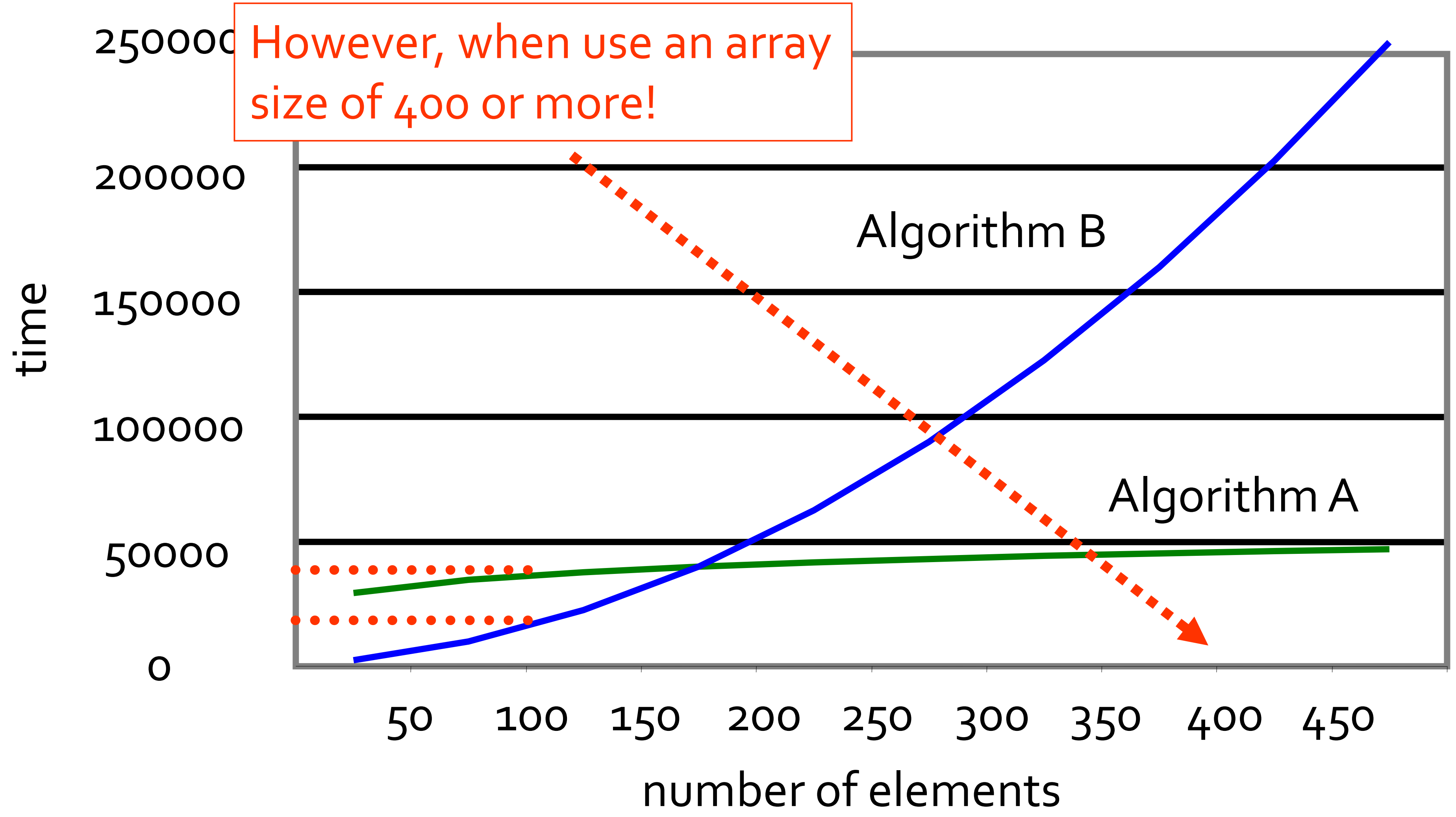
The Reality


250000
200000
150000
100000
50000
0
time
Algorithm B
Algorithm A
50
100
150
200
250
300
350
400
450
number of elements

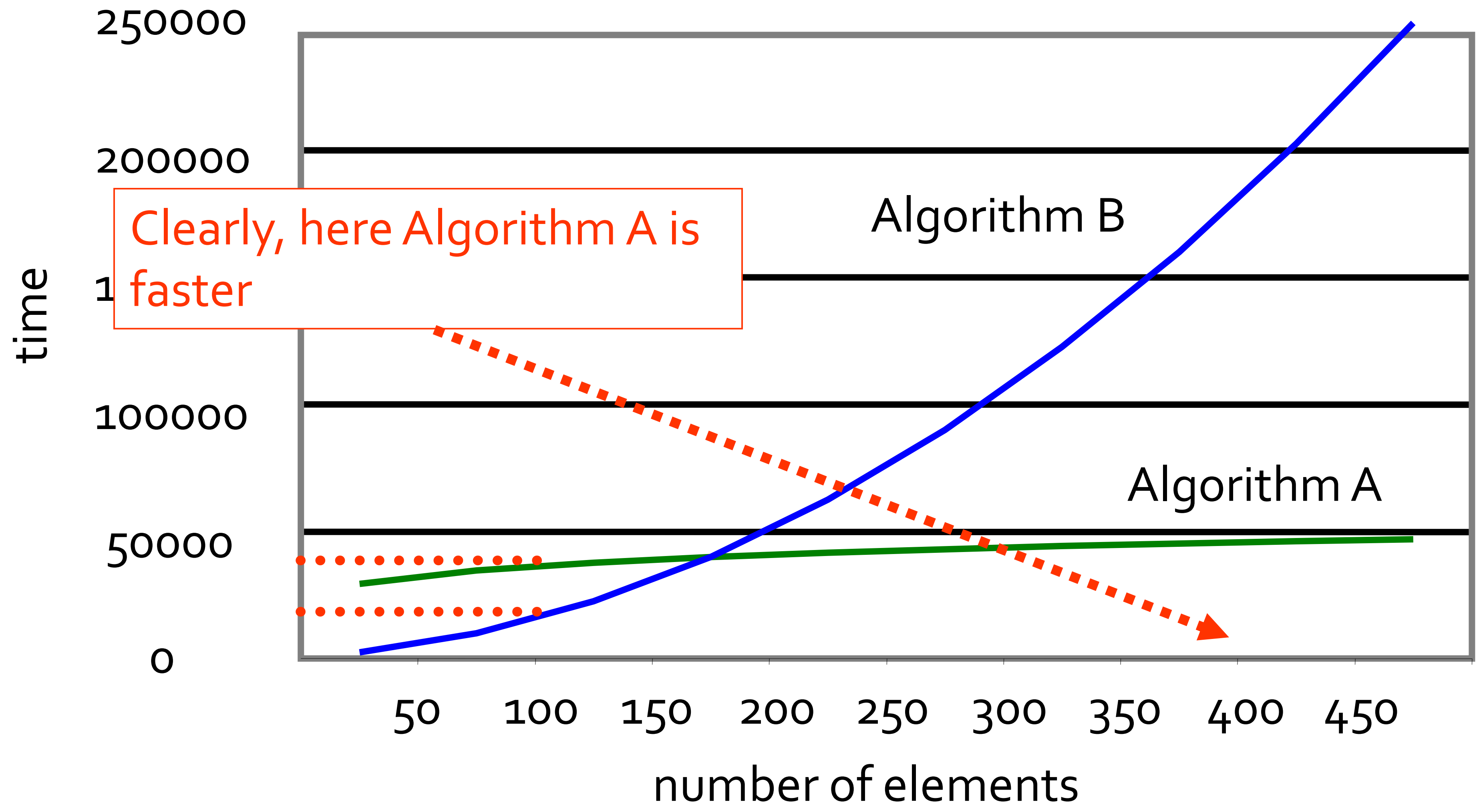
The Reality (cont.)



The Reality (cont.)



The Reality (cont.)



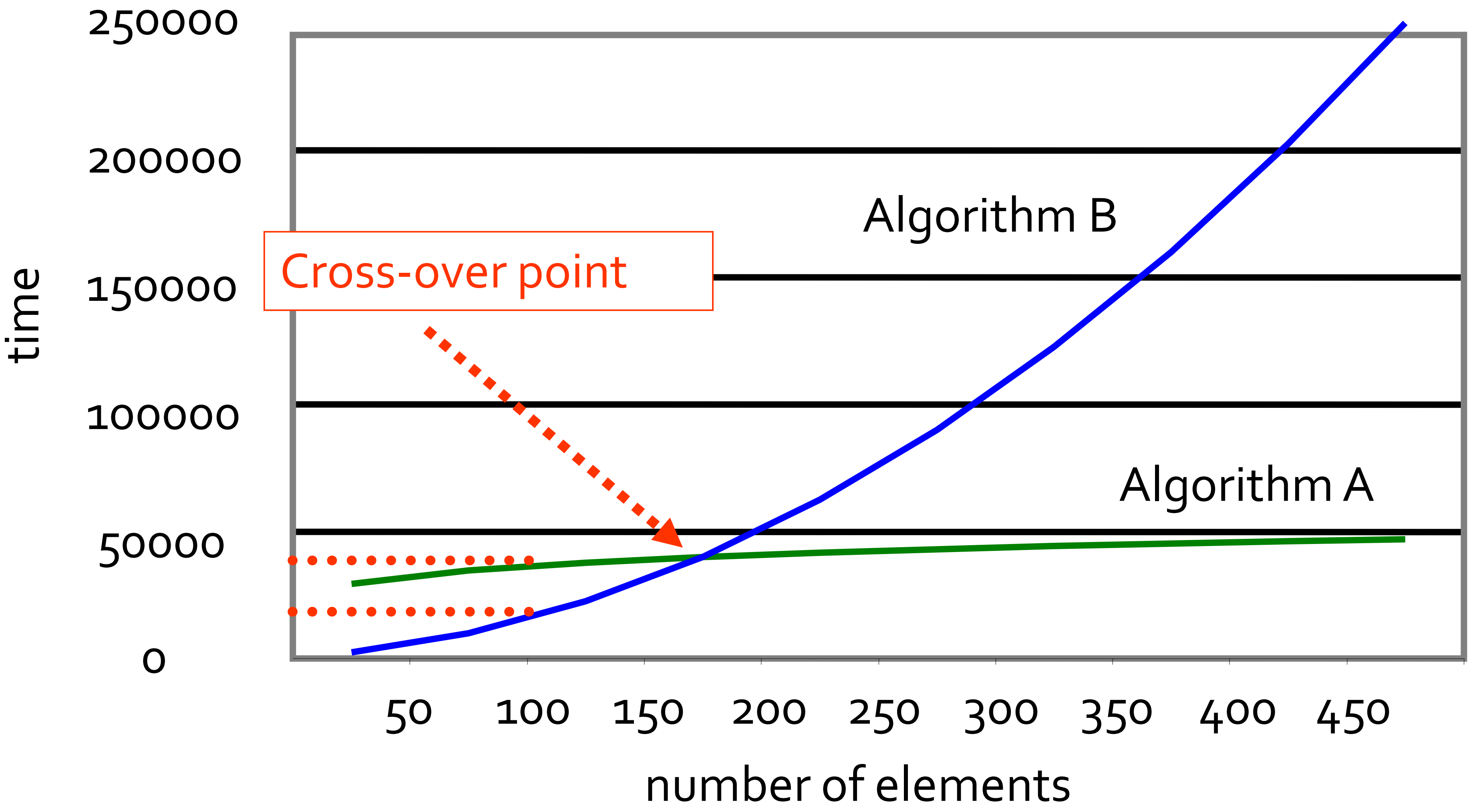
The Reality (cont.)

- By knowing time complexities, we can make an informed decision about how the algorithms compare

Cross-over Point

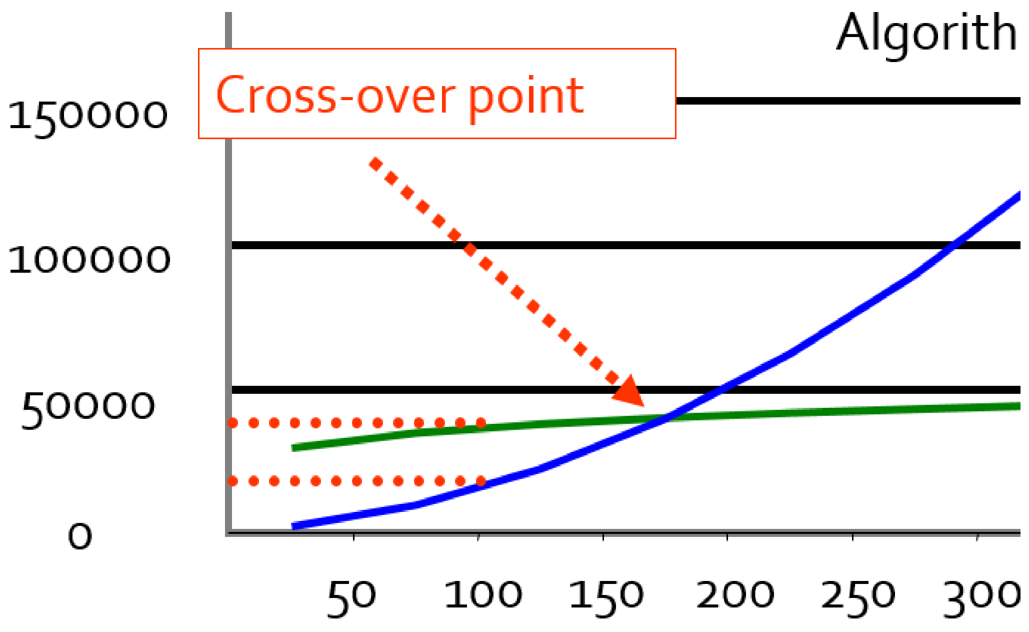
- The graphs of two algorithms with different time complexities often have a ***cross-over point***…

Cross-over Point (cont.)



Cross-over Point (cont.)

- Computer scientists often ignore the left of the cross-over point – the number of elements is low here, so execution time of almost any algorithm is expected to be fast
- To the right of the cross-over point, where the number of elements is high, is where people notice the difference in algorithm execution
- ***Asymptotic running time*** – running time of an algorithm as the number of elements approaches infinity



Algorithm Behavior Components

- ***n*** is used to stand for the number of elements
  - In reality, it can be anything that has a pronounced effect on execution time of an algorithm as it is varied
  - n is often referred to as the ***problem size***
- The amount of time an algorithm takes to execute is represented by the number of instructions that are executed

How Many Instructions?

```
sum = 0;
i = 0;
while ( i < 3 ) {
        sum += A[ i ];
        i++;
}
```

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < 3 ) {
        sum += A[ i ];
        i++;
}
```

3 instructions to this point

3 +

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < 3 ) {
        sum += A[ i ];
        i++;
}
```

The loop has 3 instructions, each executed 3 times

3 +

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < 3 ) {
        sum += A[ i ];
        i++;
}
```

The loop has 3 instructions, each executed 3 times

3 + 3*(3) = 12 instructions

Functions of n

The number of instructions can often be written as a function of n:

```
sum = 0;
i = 0;
while ( i < numElements ) {
        sum += A[ i ];
        i++;
}
```

In this example, numElements is n

The number of instructions is:
$3 + n(3) = 3n + 3$, or

$f(n) = 3n + 3$

How Many Instructions?

```
sum = 0;
i = 0;
while ( i < n ) {
        j = 0;
        while ( j < n ) {
                sum += i * j;
                j++;
                }
        i++;
        }
```

How Many Instructions? (cont.)

```
1   sum = 0;
2   i = 0;
3   while ( i < n ) {
4       j = 0;
5       while ( j < n ) {
6           sum += i * j;
7           j++;
8           }
9       i++;
10      }
```

Let's look at the inner part first – line 4 - 9

How Many Instructions? (cont.)

```
1   sum = 0;
2   i = 0;
3   while ( i < n ) {
4           j = 0;
5           while ( j < n ) {
6                   sum += i * j;
7                   j++;
8                   }
9           i++;
10          }
```

Line 4, 5 = 2 instructions

Interate n time with 3 instructions , giving us 2 + n(3)

When line 9 is executed, we have a total 3n + 3 instructions for line 4 - 9

How Many Instructions? (cont.)

```
1   sum = 0;
2   i = 0;
3   while ( i < n ) {
4           j = 0;
5           while ( j < n ) {
6                   sum += i * j;
7                   j++;
8                   }
9           i++;
10          }
```

$2 + n(3) + 1 = 3n + 3$ instructions

Add the condition on line 3

How Many Instructions? (cont.)

```
1   sum = 0;
2   i = 0;
3   while ( i < n ) {
4           j = 0;
5           while ( j < n ) {
6                   sum += i * j;
7                   j++;
8                   }
9           i++;
10          }
```

2 + n(3) + 1 = 3n + 3 instructions

Add the condition on line 3

Yields 3n + 4 instructions

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < n ) {
        j = 0;
        while ( j < n ) {
                sum += i * j;
                j++;
                }
        i++;
        }
```

3n + 4 instructions are executed n times (the outer while loop)

Giving

n(3n + 4) instructions

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < n ) {
        j = 0;
        while ( j < n ) {
                sum += i * j;
                j++;
        }
        i++;
}
```

Add 3 more initial instructions

$n(3n + 4) + 3$ instructions

How Many Instructions? (cont.)

```
sum = 0;
i = 0;
while ( i < n ) {
      j = 0;
      while ( j < n ) {
            sum += i * j;
            j++;
      }
      i++;
}
```

Add 3 more initial instructions

n(3n + 4) + 3 instructions

= $3n^2 + 4n + 3$ instructions

Our Functions

- The two functions in the last two examples were
  $f(n) = 3n + 3$
  $g(n) = 3n^2 + 4n + 3$

- These two functions have different shapes when graphed

Determining a Time Complexity

- If a function for the number of instructions has at least one term with **n**, we can determine the time complexity by:
  1. Removing the **least significant** terms from the function
  2. Removing the **coefficient** of the remaining term

Example

- For $g(n) = 3n^2 + 4n + 3$

$$3n^2 + 4n + 3$$

Example

- For $g(n) = 3n^2 + 4n + 3$

$$3n^2 + \underline{4n + 3}$$

Remove the least significant terms

Example

- For $g(n) = 3n^2 + 4n + 3$

$3n^2$

Example

- For $g(n) = 3n^2 + 4n + 3$

$$\underline{3}n^2$$

Remove the coefficient of the remaining term.

Example

- For $g(n) = 3n^2 + 4n + 3$

n^2

Example

- For g(n) = $3n^2 + 4n + 3$

$\underline{n^2}$

The time complexity that g(n) belongs to

A Time Complexity is a Set

- A time complexity is a set of functions
- O(n) has an infinite number of functions that belong to it:

Examples:

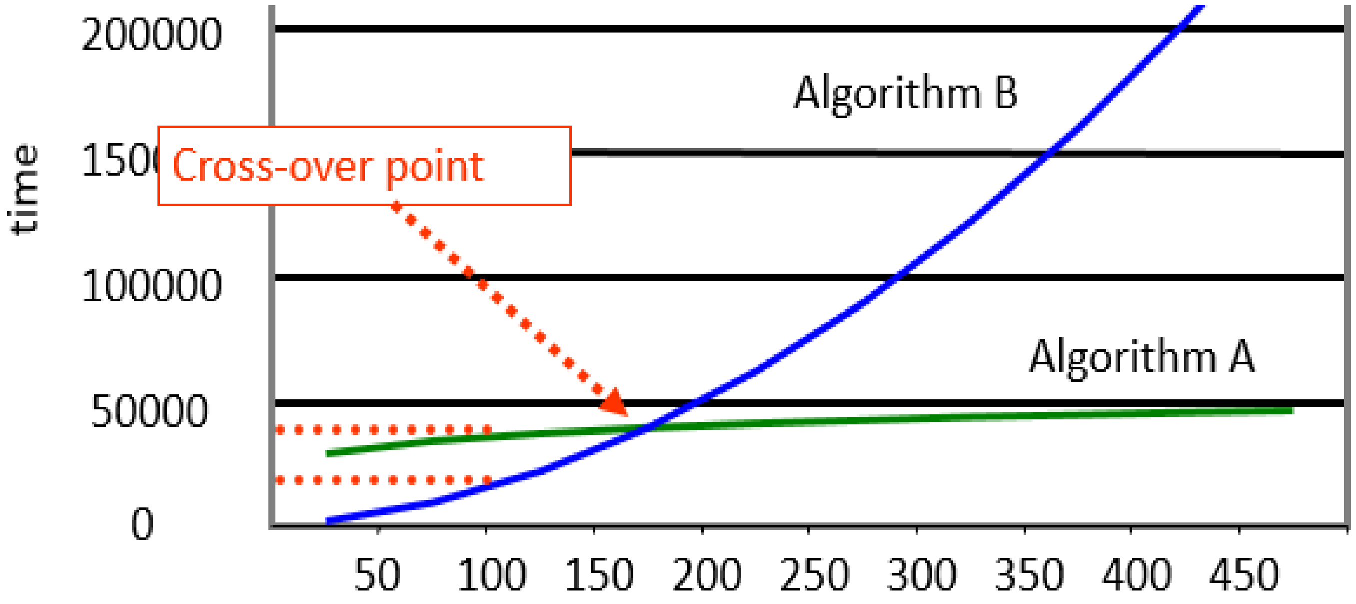
$3n + 4$, $10n + 1$, $5n$, $2n + 100$, n, $\frac{1}{2}n + 25$, $n - 2$, etc.

Big O Notation for Time Complexities

- **Big O** notation is used for time complexities
  - A time complexity of n^2 is written $O(n^2)$
- Using **Big O** notation, we know that what we see is a time complexity and not the number of instructions

How Different Time Complexities Compare

- If two functions belong to two different time complexities, then to the right of the cross-over point:
  - One will be faster than the other
  - As **n** element is increased further, more benefit will be gained from the faster function



Example

n	**f(*n*) = $n^2 + n + 2$ (number of instructions)**	**g(*n*) = 4*n* + 2 (number of instructions)**	**How many times faster g(*n*) is over f(*n*) -- [f(*n*) / g(*n*)]**
10	112	42	2.7
100	10102	402	25.1
1000	1001002	4002	250.1

Constant Time Complexity

- The constant time complexity, written O(1), is the best possible time complexity
- As n increases, there is generally no effect on the number of instructions executed
  - Example: the algorithm which dequeues from the linked list implementation of a queue

Example

The algorithm below finds itemToFind in a sorted array:

```
i = 0;
found = false;
while ( ( i < size ) && itemToFind > A[ i ] )
        i++;

if ( itemToFind == A[ i ] )
        found = true;
```

This algorithm can run in O(1) time, if itemToFind is the first item in the array.

Example (cont.)

The algorithm below finds itemToFind in a sorted array:

```
i = 0;
found = false;
while ( ( i < size ) && itemToFind > A[ i ] )
        i++;

if ( itemToFind == A[ i ] )
        found = true;
```

It can also run in O(n) time, if itemToFind is the last item in the array.

We can say it runs in O(n) time.

Logarithmic Equations

- A logarithmic equation is just another way of writing an exponential equation…

Logarithmic and Exponential Equations

$$\log_3 9 = 2$$

$$3^2 = 9$$

Logarithmic and Exponential Equations (cont.)

- The two equations mean exactly the same thing, but are just different ways of writing it
- To convert between the equations, remember two simple rules:
  - The base of the logarithm is also the base for the exponent
  - The result of a logarithm is an exponent

$$\log_3 9 = 2$$

$$3^2 = 9$$

Example

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

The base is a base for an exponent

2

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

The result of a logarithm is an exponent.

2^x

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

There is only one place left for the 8 to go.

2^x

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

There is only one place left for the 8 to go.

$2^x = 8$

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

x is 3 here

$2^x = 8$

Example (cont.)

- Solve $\log_2 8$ by converting to exponential form
- $\log_2 8 = x$

so the result of the logarithm is 3

$2^x = 8$

log

- In computer science, log is used as a logarithm with base 2
- $\log 32 = x$
- $2^x = 32$
- So $x = 5$

The Logarithmic Time Complexity

- O(log n) is the next best thing to O(1) in the most common time complexities
- One way an algorithm can achieve a logarithmic time complexity is by reducing the problem size by one half on each iteration, until the problem size becomes 1
- If we had an initial problem size of one billion elements, and the problem size is reduced by half each time through a loop, only 30 iterations would be required to get down to a problem size of 1.

The End of Part 1