

# CSCI 132:

# Basic Data Structures and Algorithms

Growth Rates

Reese Pearsall & Iliana Castillon  
Fall 2024

# Announcements

Lab 6 due **Thursday** at 11:59 pm  
No class on Friday (work day)

Program 2 due next Friday



Midterm Exam is next week

- Study guide will be posted soon

No lab next week

As computer scientists, we write many **algorithms**

We want to be able to describe how well our algorithms perform on a variety of inputs

As computer scientists, we write many **algorithms**

We want to be able to describe how well our algorithms perform on a variety of inputs



Consider an algorithm that will **make a cake**

How could measure the effectiveness and performance of our cake making algorithm?



## Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*

What is the **total time needed** to make the cake?

- Prep time
- Combining ingredients
- Baking
- Cooling
- Icing

The **time** an algorithm takes to finish is important. We generally want our algorithms to run as fast as possible



# Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*

*How well does our algorithm work on a variety of problems?*



Suppose we needed to make different types of cake, How well would our algorithm do?

**Generalizability and Reliability**



Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*

*How well does our cake taste?*



We want our algorithm to output the best cake possible

Does our algorithm always yield the **optimal** result ?





Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*

*Does our algorithm actually make a cake?*



Does our algorithm *actually* do what we say it does?

Our algorithm needs to be **valid**





Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*



Suppose we needed to make a lot of the same cake

How well does our algorithm **scale**?



Consider an algorithm that will **make a cake**

*What are some ways we could measure the performance and effectiveness of our algorithm ?*



How much kitchen **space** is needed to make the cake?

What if the cake needed is really big?



# Algorithm Analysis

## Performance & Efficiency

1. Running time of Algorithm
2. Space needed to run the algorithm

## Algorithm Correctness

- Validity
- Optimality
- Generalizability



# Algorithm Analysis

## Performance & Efficiency

1. Running time of Algorithm
2. Space needed to run the algorithm

**Important:** How well does the algorithm do as the problem gets bigger? (Scalability)

## Algorithm Correctness

- Validity
- Optimality
- Generalizability

The **growth rate** of the algorithm looks at how much more resource an algorithm needs (time or space) as the input size increases

*(In this class, we will be focusing on **time**)*

# Running time issues

The easiest way I could prove the running time of an algorithm is by starting a stopwatch when the algorithm starts, and stop when algorithm finishes

This is not a very good way to accurately show the running time **because ...**





# Running time issues

The easiest way I could prove the running time of an algorithm is by starting a stopwatch when the algorithm starts, and stop when algorithm finishes



This is not a very good way to accurately show the running time **because ...**



What if this is my computer?

# Running time issues

**Issue:** The time needed to run an algorithm varies depending on the hardware of computer that is running the algorithm



# Running time issues

**Issue:** The time needed to run an algorithm varies depending on the hardware of computer that is running the algorithm

Instead of focusing on the actual time needed to run an algorithm (seconds), we will look at the **number of steps/instructions in the algorithm** that need be executed *as the input grows!*

The **growth rate** of the algorithm looks at how much more resource an algorithm needs (time or space) as the input size increases

???

???

We will look at 4 growth rates today

???

???

The remaining two will be covered later this semester

???

???

## Algorithm #1: Finding the Maximum Value in an Array

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```

This algorithm will find the largest value in some **N** sized array

## Algorithm #1: Finding the Maximum Value in an Array

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```

This algorithm will find the largest value in some **N** sized array

This code checks each spot in the array



## Algorithm #1: Finding the Maximum Value in an Array

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```

This algorithm will find the largest value in some **N** sized array

This code checks each spot in the array

Let's look at how many times this instruction is executed as the array size grows

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```



Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	9999

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	9999
1	

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	9999
1	1

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	9999
1	1
1000000000	

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```



# Algorithm #1: Finding the Maximum Value in an Array

Array Size      Number of Spots Checked

10	10
100	100
533	533
1000	1000
9999	9999
1	1
10000000000	10000000000

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

Algorithm #1: Finding the Maximum Value in an Array

<u>Array Size</u>	<u>Number of Spots Checked</u>
10	10
100	100
533	533
1000	1000
9999	9999
1	1
1000000000	1000000000

```
public int find_max_value(int[] array) {
    int largest_so_far = -1;
    for(int i = 0; i < array.length; i++) {
        if ( array[i] > largest_so_far ) {
            largest_so_far = array[i];
        }
    }
    return largest_so_far;
}
```

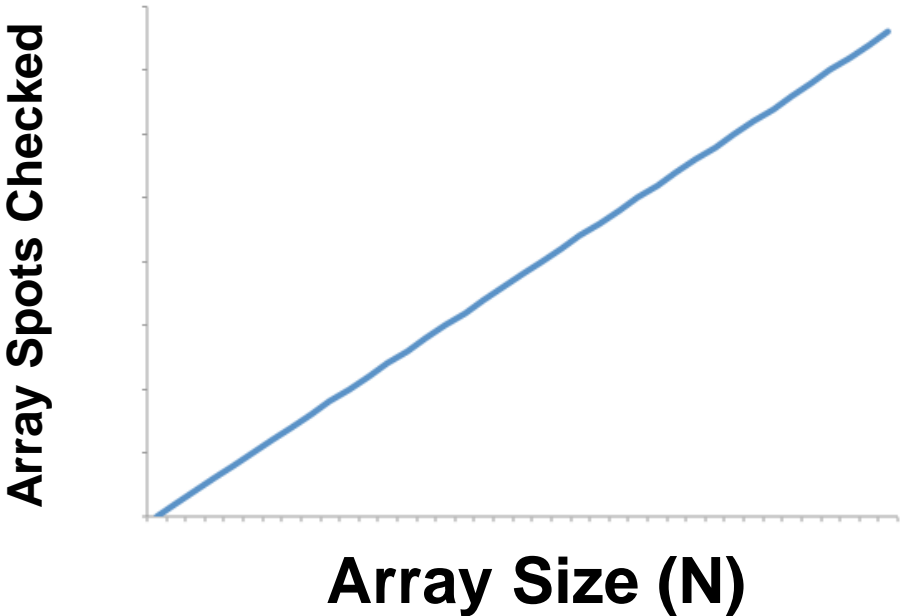
What if we graphed these points?

(10,10) , (100,100), (533,533), (1000,100) ....

# Algorithm #1: Finding the Maximum Value in an Array

<u>Array Size</u>	<u>Number of Spots Checked</u>
10	10
100	100
533	533
1000	1000
9999	9999
1	1
1000000000	1000000000

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```

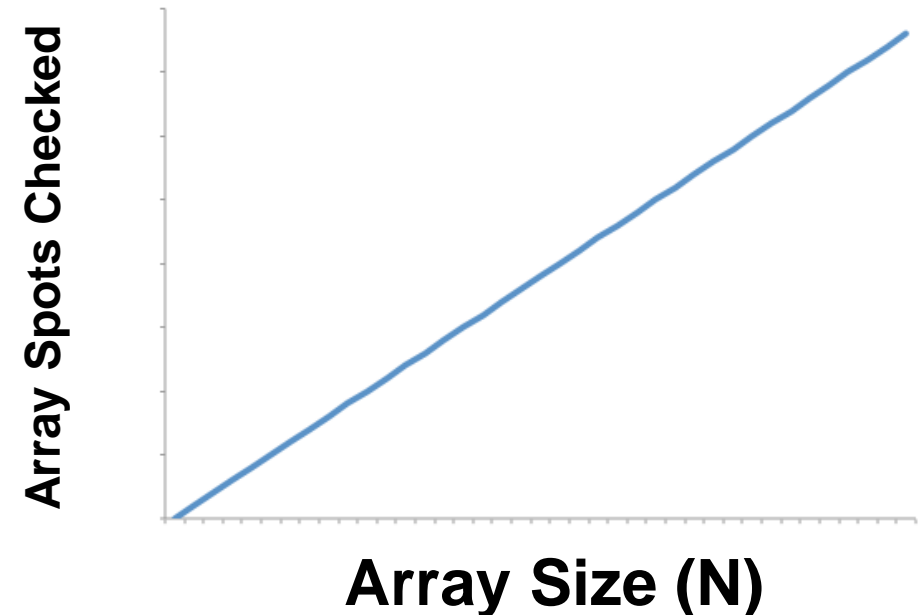


## Algorithm #1: Finding the Maximum Value in an Array

The growth rate of this algorithm is **linear**

A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other.

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```



## Algorithm #1: Finding the Maximum Value in an Array

The growth rate of this algorithm is **linear**

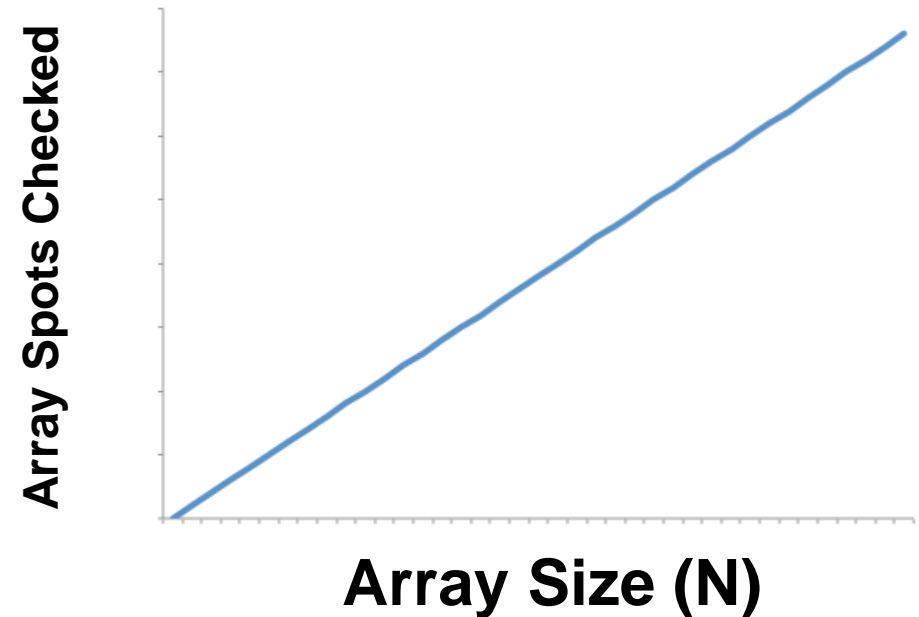
A linear growth rate is a growth rate where the resource needs and the amount of data is directly proportional to each other.

The driving factor of this algorithm is the size of the array

As N increases, the number of steps executed in this algorithm linearly increases

$$F(x) = N$$

```
public int find_max_value(int[] array) {  
    int largest_so_far = -1;  
    for(int i = 0; i < array.length; i++) {  
        if ( array[i] > largest_so_far ) {  
            largest_so_far = array[i];  
        }  
    }  
    return largest_so_far;  
}
```



## Algorithm #2: Printing out an N x N 2D Array

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

Given an N x N 2D array, print out its contents

## Algorithm #2: Printing out an N x N 2D Array

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

Given an N x N 2D array, print out its contents

N = 3

000

000

000

N = 7

0000000

0000000

0000000

0000000

0000000

0000000

0000000

# Algorithm #2: Printing out an N x N 2D Array

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

Given an N x N 2D array, print out its contents

Let's look at how many times this operation is executed as N increases

N = 3

000  
000  
000

N = 7

0000000  
0000000  
0000000  
0000000  
0000000  
0000000  
0000000



# Algorithm #2: Printing out an N x N 2D Array

**N**                  Number of Array Spots Printed out


```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                  Number of Array Spots Printed out

1	

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                  Number of Array Spots Printed out

1	1

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```

# Algorithm #2: Printing out an N x N 2D Array

N	Number of Array Spots Printed out
1	1
2	4

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

# Algorithm #2: Printing out an N x N 2D Array

N	Number of Array Spots Printed out
1	1
2	4
3	

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                      Number of Array Spots Printed out

1	1
2	4
3	9

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                      Number of Array Spots Printed out

1	1
2	4
3	9
4	

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                      Number of Array Spots Printed out

1	1
2	4
3	9
4	16

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```



# Algorithm #2: Printing out an N x N 2D Array

**N**                      Number of Array Spots Printed out

1	1
2	4
3	9
4	16
5	25

```
public void print2Darray(int[][] array) {
    for(int[] x: array) {
        for(int y: x) {
            System.out.print(y);
        }
        System.out.println();
    }
}
```

# Algorithm #2: Printing out an N x N 2D Array

N	Number of Array Spots Printed out
1	1
2	4
3	9
4	16
5	25
...	...
100	10000

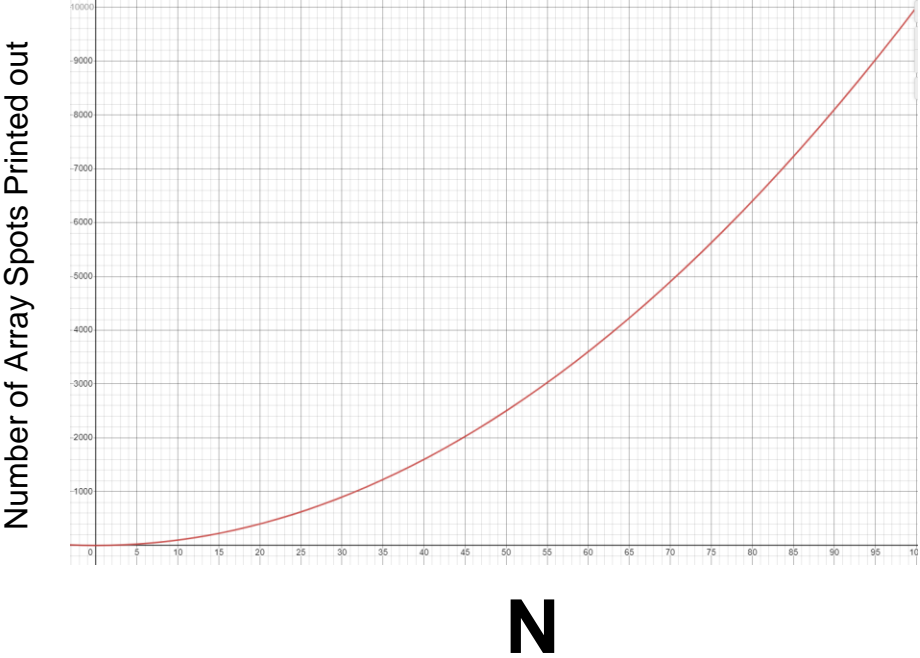
```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

# Algorithm #2: Printing out an N x N 2D Array

**N**                      Number of Array Spots Printed out

1	1
2	4
3	9
4	16
5	25
...	...
100	10000

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```

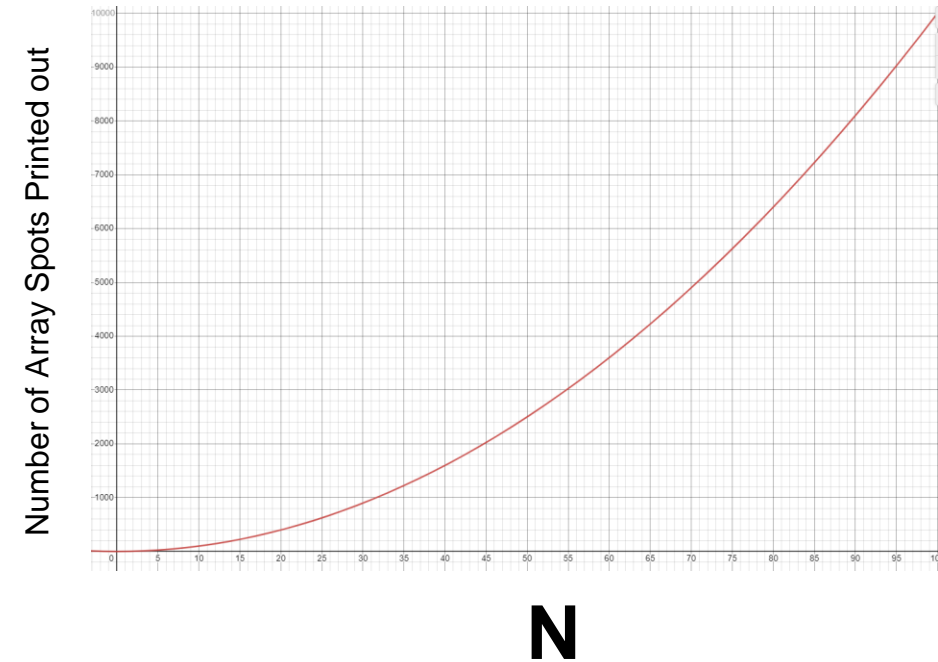


## Algorithm #2: Printing out an N x N 2D Array

The growth rate of this algorithm is **quadratic**

A quadratic growth rate is a growth rate where the resource needs and the amount of data is proportional to the *square* of a function

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```



## Algorithm #2: Printing out an N x N 2D Array

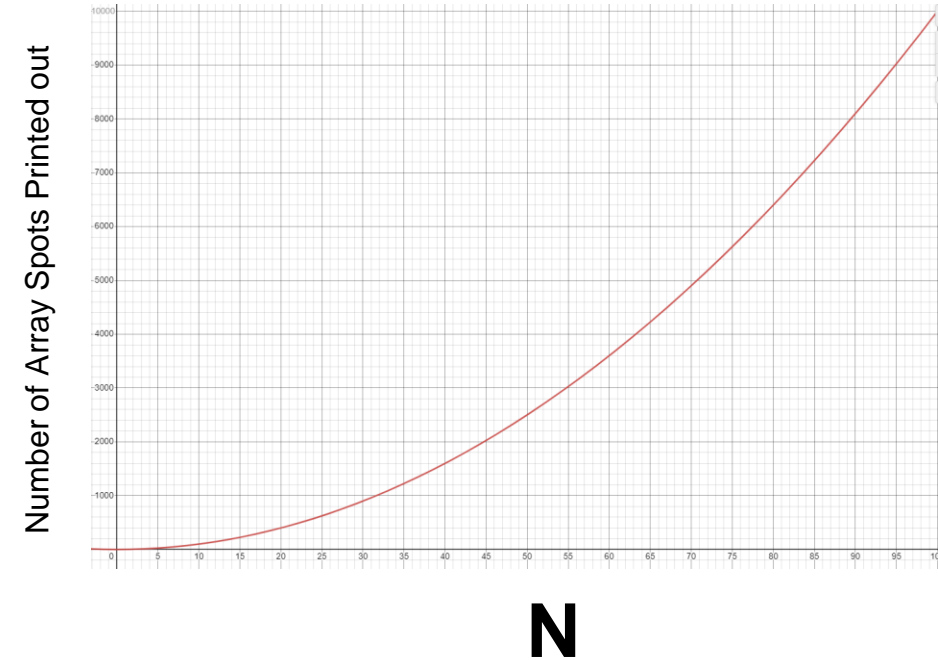
The growth rate of this algorithm is **quadratic**

A quadratic growth rate is a growth rate where the resource needs and the amount of data is proportional to the *square* of a function

$$F(x) = X^2$$

We have a for loop inside of a for loop, so as N increases, the number of times the inside for loop executes =  $N * N$

```
public void print2Darray(int[][] array) {  
    for(int[] x: array) {  
        for(int y: x) {  
            System.out.print(y);  
        }  
        System.out.println();  
    }  
}
```



### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

Given a singly linked list (with at least one node), this algorithm adds a new node to the front of the LL

### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

Given a singly linked list (with at least one node), this algorithm adds a new node to the front of the LL



### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

Given a singly linked list (with at least one node), this algorithm adds a new node to the front of the LL

This algorithm consists of two operations. Let's look at how many times these operations are executed as the Linked List size increases

# Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

**N**                      **# of operations executed**


N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2

N= # of nodes in LL



Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	

N= # of nodes in LL

### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2

N= # of nodes in LL

# Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2
777	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2
777	2

N= # of nodes in LL

# Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2
777	2
1000000000	

N= # of nodes in LL

Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2
777	2
1000000000	2

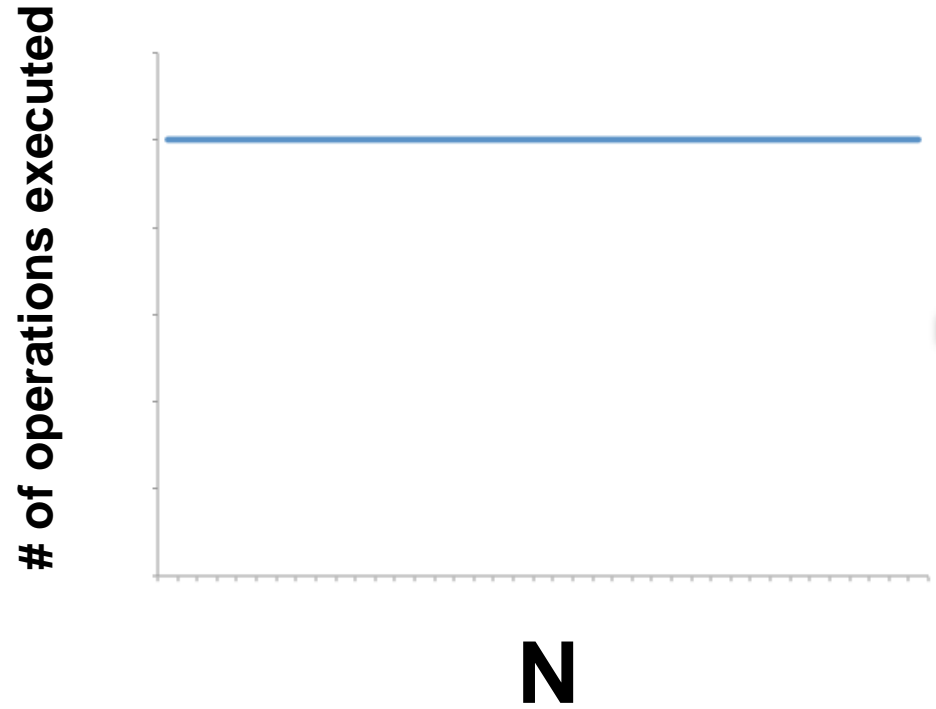
N= # of nodes in LL



Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

N	# of operations executed
1	2
2	2
3	2
4	2
100	2
777	2
1000000000	2



N= # of nodes in LL

### Algorithm #3: Adding a node to front of linked list

```
public void addToFront(Node newNode) {  
    newNode.setNext(this.head);  
    this.head = newNode;  
}
```

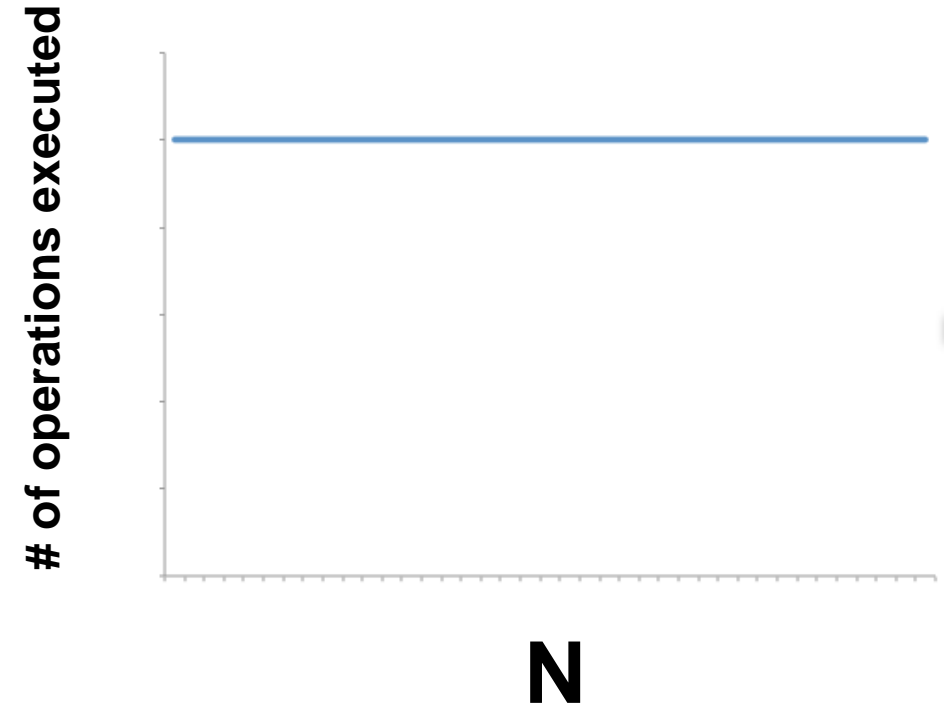
The growth rate of this algorithm is **constant**

A **constant growth rate** is one where the resource need does not grow as  $N$  increases.

$$F(x) = 1$$

The number of instructions executed for adding something to LL of size 1 is the same as adding to a LL of size 1000000000000

ie. As  $N$  increases, the number of steps our algorithm performs is constant



## Algorithm #4: Generating all possible binary combinations of length N

## Algorithm #4: Generating all possible binary combinations of length N

**Input:** 2

**Output:**

0 0

0 1

1 0

1 1

# Algorithm #4: Generating all possible binary combinations of length N

**Input: 2**

**Output:**

0 0

0 1

1 0

1 1

**Input: 3**

**Output:**

0 0 0

0 0 1

0 1 0

0 1 1

1 0 0

1 0 1

1 1 0

1 1 1

Algorithm #4: Generating all possible binary combinations of length N

**Input: 2**  
**Output:**  
0 0  
0 1  
1 0  
1 1

**Input: 3**  
**Output:**  
0 0 0  
0 0 1  
0 1 0  
0 1 1  
1 0 0  
1 0 1  
1 1 0  
1 1 1

**Input: 4**  
**Output**  
0000  
0001  
0010  
0011  
0100  
0101  
0110  
0111  
...  
1111

## Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

Don't worry too much about the specifics of the algorithm. Let's look at the amount a string that get generated as N increases

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {
    if (i == n) {
        printTheArray(arr, n);
        return;
    }
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}
```

**N**                      **# Of binary digits generated**

1	

**N = length of binary digits**



# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {
    if (i == n) {
        printTheArray(arr, n);
        return;
    }
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}
```

N	# Of binary digits generated
1	2

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

**N**                      **# Of binary digits generated**

1	2
2	4
3	

**N = length of binary digits**

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4
3	8

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {
    if (i == n) {
        printTheArray(arr, n);
        return;
    }
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16
5	

N = length of binary digits



# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16
5	32

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {
    if (i == n) {
        printTheArray(arr, n);
        return;
    }
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16
5	32
8	256

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {
    if (i == n) {
        printTheArray(arr, n);
        return;
    }
    arr[i] = 0;
    generateAllBinaryStrings(n, arr, i + 1);

    arr[i] = 1;
    generateAllBinaryStrings(n, arr, i + 1);
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16
5	32
8	256
16	65536

N = length of binary digits

# Algorithm #4: Generating all possible binary combinations of length N

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

N	# Of binary digits generated
1	2
2	4
3	8
4	16
5	32
8	256
16	65536

# Of binary digits generated



N = length of binary digits

## Algorithm #4: Generating all possible binary combinations of length N

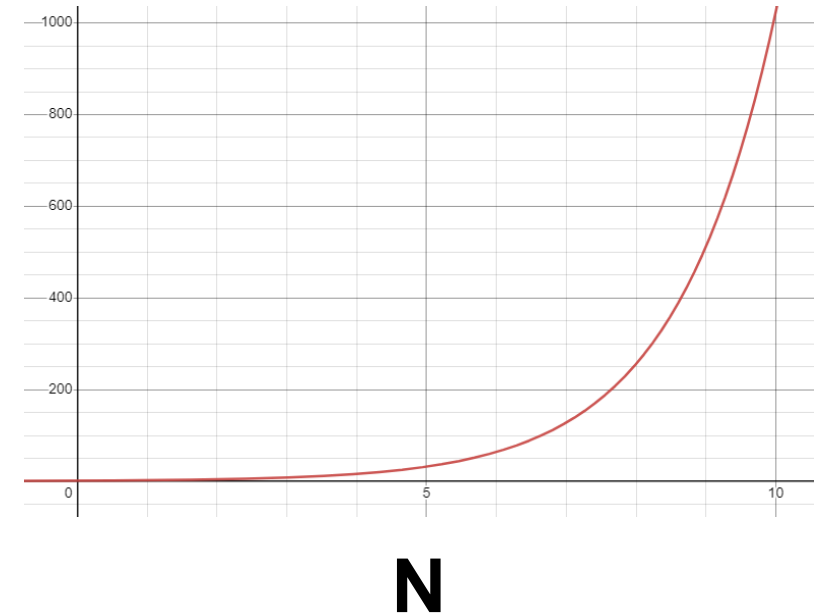
The growth rate of this algorithm is **exponential**

An **exponential growth rate** is one where the resource needed begins to double or increase very drastically as N increases

$$F(x) = B ^ x$$

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

# Of binary digits generated



## Algorithm #4: Generating all possible binary combinations of length N

The growth rate of this algorithm is **exponential**

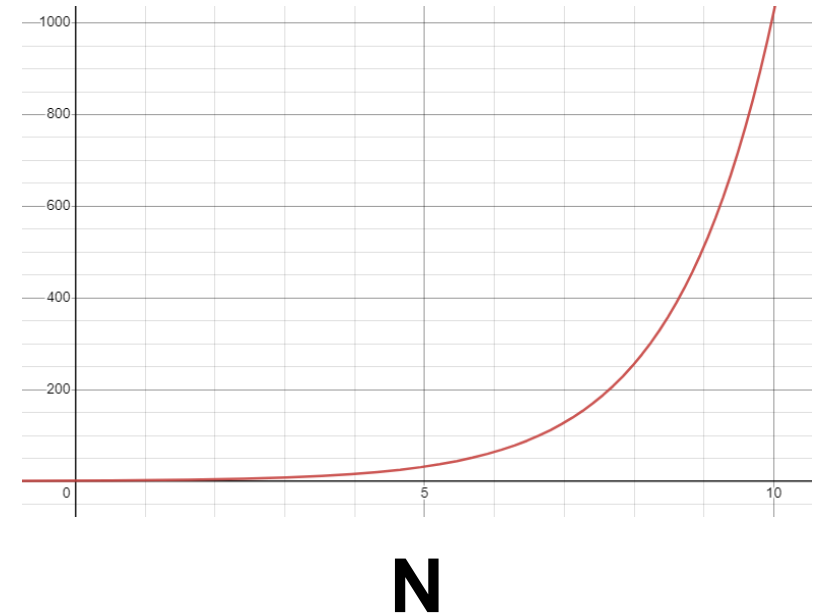
An **exponential growth rate** is one where the resource needed begins to double or increase very drastically as N increases

$$F(x) = B ^ x$$

An algorithm that has an exponential growth rate is generally perceived as **inefficient**. When N gets big, sometimes the algorithm won't finish for years

```
static void generateAllBinaryStrings(int n, int arr[], int i) {  
    if (i == n) {  
        printTheArray(arr, n);  
        return;  
    }  
    arr[i] = 0;  
    generateAllBinaryStrings(n, arr, i + 1);  
  
    arr[i] = 1;  
    generateAllBinaryStrings(n, arr, i + 1);  
}
```

# Of binary digits generated



The **growth rate** of the algorithm looks at how much more resource an algorithm needs (time or space) as the input size increases

**Constant**

???

???

**Quadratic**

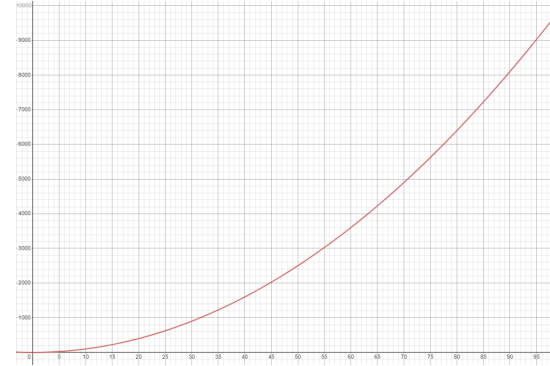
**Linear**

**Exponential**

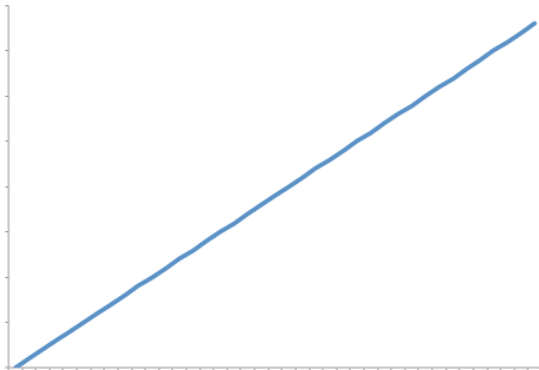
## Constant



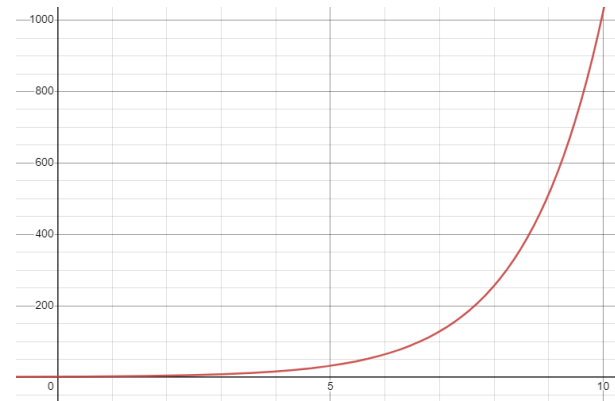
## Quadratic



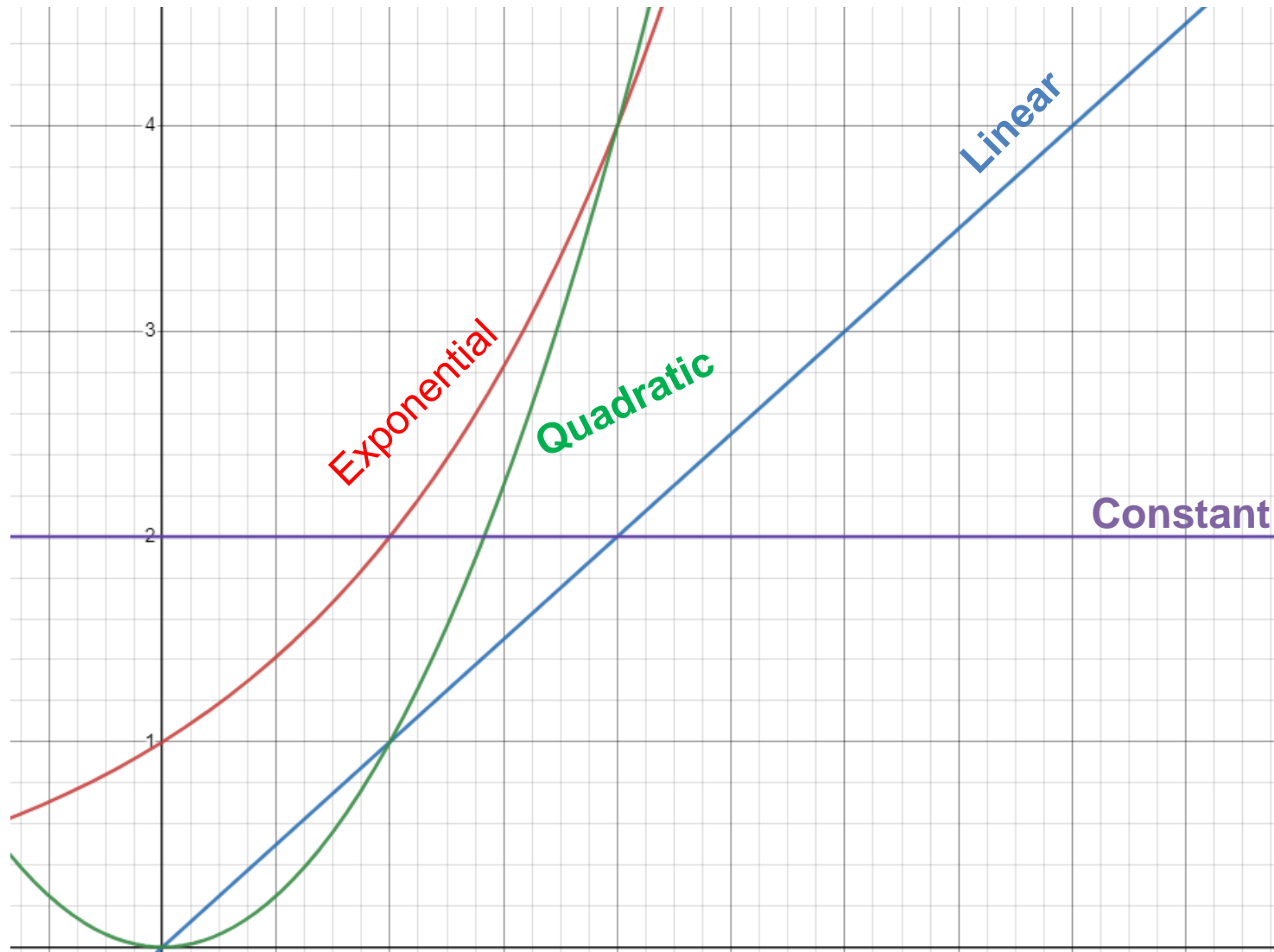
## Linear

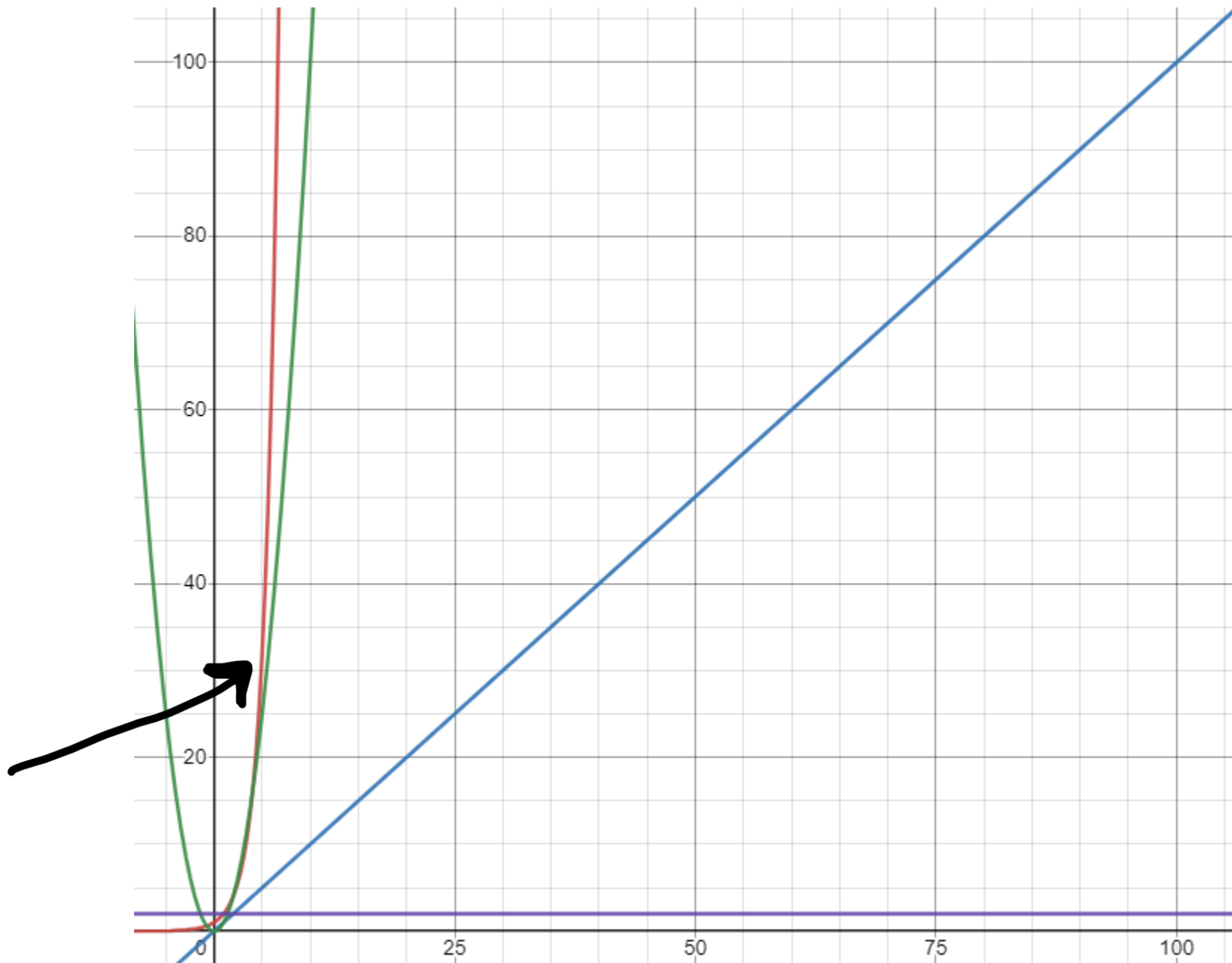


## Exponential









After a certain point, exponential growth rates goes crazy and beings to grow much faster than the other growth rates

## Algorithm #5: Generating prime numbers up to N

```
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        if(n%i==0)return false;
    }
    return true;
}

public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
}
```

This algorithm generates prime numbers up to N

## Algorithm #5: Generating prime numbers up to N

```
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        if(n%i==0)return false;
    }
    return true;
}

public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
}
```

This algorithm generates prime numbers up to N

Check all possible values up to N

# Algorithm #5: Generating prime numbers up to N

```
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        if(n%i==0)return false;
    }
    return true;
}

public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
}
```

This algorithm generates prime numbers up to N

Check every possible factor of a value

Check all possible values up to N

## Algorithm #5: Generating prime numbers up to N

```
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        if(n%i==0)return false;
    }
    return true;
}

public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
}
```

This algorithm generates prime numbers up to N

How many factors do we check given N?

## Algorithm #5: Generating prime numbers up to N

```
static int counter= 0;
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        counter++;
        if(n%i==0)return false;
    }
    return true;
}

public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
    System.out.println();
    System.out.println(counter);
}
```

Let's add a counter!

# Algorithm #5: Generating prime numbers up to N

```
static int counter= 0;
//function to check if a given number is prime
static boolean isPrime(int n)
{
    if(n==1||n==0) return false;
    for(int i=2; i<=n-1; i++){
        counter++;
        if(n%i==0)return false;
    }
    return true;
}

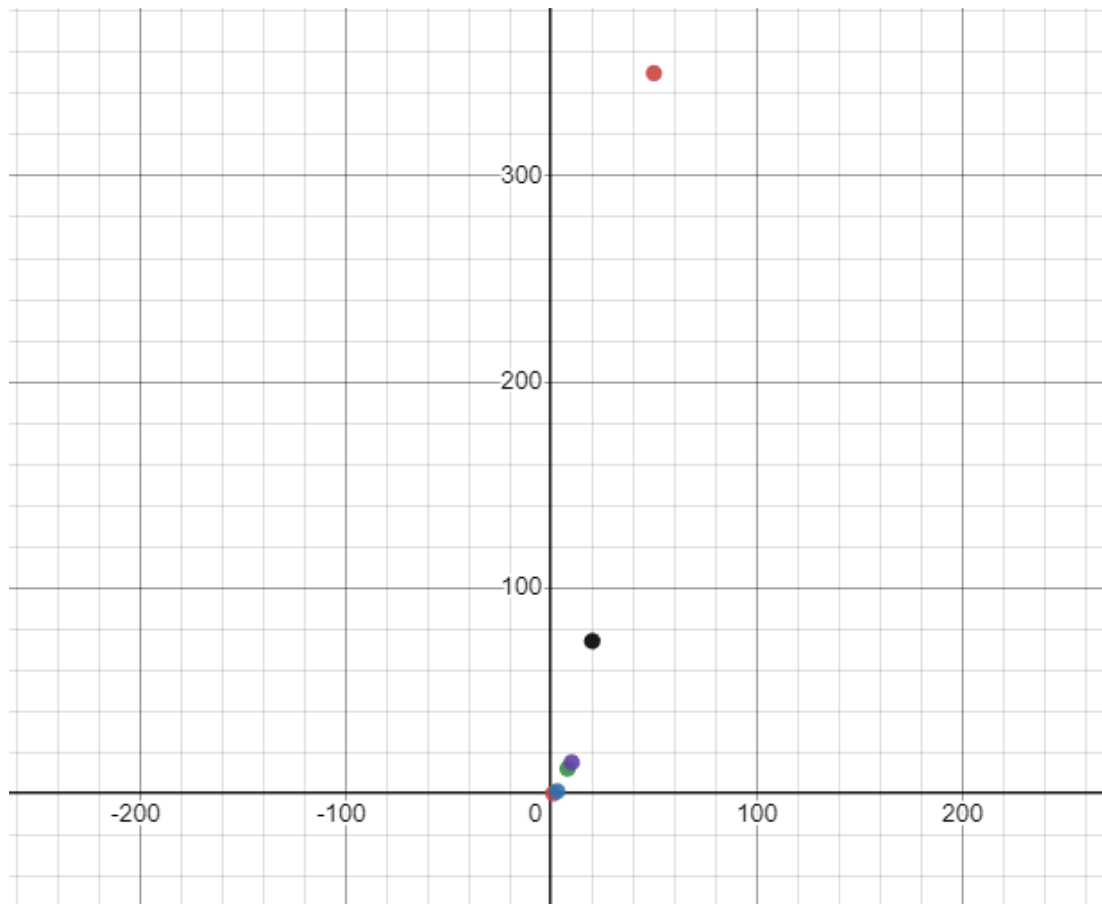
public static void main (String[] args)
{
    int N = 50;
    for(int i=1; i<=N; i++){
        if(isPrime(i)) {
            System.out.print(i + " ");
        }
    }
    System.out.println();
    System.out.println(counter);
}
```

Let's add a counter!

N	Counter
1	0
3	1
8	12
10	15
20	74
50	350



Algorithm #5: Generating prime numbers up to N



This looks to be **quadratic**

## Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {  
    int n = 4; //0 - 9999  
    int upperbound = 1;  
    for(int i = 0; i < n; i++) {  
        upperbound *= 10;  
    }  
    for(int i = 0; i < upperbound; i++ ) {  
        guess(i); //guess a password  
    }  
}
```

## Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {  
    int n = 4; //0 - 9999  
    int upperbound = 1;  
    for(int i = 0; i < n; i++) {  
        upperbound *= 10;  
    }  
    for(int i = 0; i < upperbound; i++ ) {  
        guess(i); //guess a password  
    }  
}
```

This for loop grows **linearly**

Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {  
    int n = 4; //0 - 9999  
    int upperbound = 1;  
    for(int i = 0; i < n; i++) {  
        upperbound *= 10;  
    }  
    for(int i = 0; i < upperbound; i++ ) {  
        guess(i); //guess a password  
    }  
}
```

This for loop grows **linearly**

???


# Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {
    int n = 4; //0 - 9999
    int upperbound = 1;
    for(int i = 0; i < n; i++) {
        upperbound *= 10;
    }
    for(int i = 0; i < upperbound; i++ ) {
        guess(i); //guess a password
    }
}
```

N	# Passcodes Guessed
1	10
2	100
3	1000
4	10000
5	100000
6	1000000


# Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {
    int n = 4; //0 - 9999
    int upperbound = 1;
    for(int i = 0; i < n; i++) {
        upperbound *= 10;
    }
    for(int i = 0; i < upperbound; i++ ) {
        guess(i); //guess a password
    }
}
```

N	# Passcodes Guessed
1	10
2	100
3	1000
4	10000
5	100000
6	1000000
10	

# Algorithm #6: Brute Forcing an N digit numeric passcode

```
public static void main(String[] args) {  
    int n = 4; //0 - 9999  
    int upperbound = 1;  
    for(int i = 0; i < n; i++) {  
        upperbound *= 10;  
    }  
    for(int i = 0; i < upperbound; i++ ) {  
        guess(i); //guess a password  
    }  
}
```

N	# Passcodes Guessed
1	10
2	100
3	1000
4	10000
5	100000
6	1000000
10	

**$y = 10^x$**       Exponential

The **growth rate** of the algorithm looks at how much more resource an algorithm needs (time or space) as the input size increases

**Constant**

???

???

**Quadratic**

**Linear**

**Exponential**