CSCI 132:
Basic Data Structures and Algorithms

Time Complexity, Big-O

Reese Pearsall

Spring 2023

*All images are stolen from the internet

Announcements

No Lab next week

Midterm Exam Wednesday

Program 2 due **Friday** 3/10 @ 11:59

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

The **fastest** time he has completed a house in the past

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

The **fastest** time he has completed a house in the past

The **slowest** time he has completed a house in the past

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

The **fastest** time he has completed a house in the past

The **slowest** time he has completed a house in the past

The **average** time it takes him to complete a house

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

(We will also assume they won't break any records for fastest/slowest time)

The **fastest** time he has completed a house in the past

"Best case scenario"

The **slowest** time he has completed a house in the past

"Worst case scenario"

The **average** time it takes him to complete a house

Average

Suppose you are moving across the country. You've contracted a builder to build you a brand-new house. You are trying to plan which date you should put all your belongings in the truck and move to the new house across the country. You ask the builder the following question:

How long will it take to finish building the house?

The builder is unsure exactly when he will be done, but he offers the following answers in an enclosed envelope. You can only pick one.

The **slowest** time he has completed a house in the past

"Worst case scenario"

If we select this option, we are **guaranteed** a date that the house will be finished by

(The house might be empty for a few days, but that's much better than having to stay in a hotel until the house is ready)

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

1.

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

1. Time (seconds, nanoseconds, minutes, days, etc)

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

~~1. Time (seconds, nanoseconds, minutes, days, etc)~~

Practical, but the hardware of the computer greatly affects the time needed

We need a way to measure running time that is independent from the hardware the computer has

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

~~1. Time (seconds, nanoseconds, minutes, days, etc)~~

2. Number of **operations** required to complete algorithm.

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

1. ~~Time (seconds, nanoseconds, minutes, days, etc)~~

2. Number of **operations** required to complete algorithm.

To measure the running time of an algorithm, we will count the number of operations the algorithm performs, and look at how these operations scale *as the input increases*

The **running time** of an algorithm is the time it takes for an algorithm to completely run from start to finish

There are a few ways we can measure running time:

1. ~~Time (seconds, nanoseconds, minutes, days, etc)~~

2. Number of **operations** required to complete algorithm.

To measure the running time of an algorithm, we will count the number of operations the algorithm performs, and look at how these operations scale *as the input increases*

When we describe the running time of an algorithm, we will represent it using **Big-O Notation**

A **primitive operation** is an operation that has a **constant** execution time

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable

```
int N = 3;
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation

```
int N = 3;
a = a + 3 * 12
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values

```
int N = 3;
a = a + 3 * 12
if(n >= i)
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values
- Accessing an element in an array (by index)

```
int N = 3;
a = a + 3 * 12
if(n >= i)
i = arr[3]
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values
- Accessing an element in an array (by index)
- Calling a method

```
int N = 3;
a = a + 3 * 12
if(n >= i)
i = arr[3]
e.print2Darray(array);
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values
- Accessing an element in an array (by index)
- Calling a method
- Returning from a method

```
int N = 3;
a = a + 3 * 12
if(n >= i)
i = arr[3]
e.print2Darray(array);
return
```

A **primitive operation** is an operation that has a **constant** execution time

- Assigning a value to a variable
- Performing an arithmetic operation
- Comparing two numbers/values
- Accessing an element in an array (by index)
- Calling a method
- Returning from a method
- Printing out a value

```
int N = 3;
a = a + 3 * 12
if(n >= i)
i = arr[3]
e.print2Darray(array);
return
System.out.println(“Hi”)
```

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

The number of operations this algorithm executes varies because…

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

The number of operations this algorithm executes varies **S** will be at different locations

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

This is a **primitive operation**, lets count how many times this operation is executed given some input

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	6	3	5	1	8	2	9	7	10

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

s = 5

4	6	3	5	1	8	2	9	7	10

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	**6**	**3**	**5**	**1**	**8**	**2**	**9**	**7**	**10**

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	6	3	5	1	8	2	9	7	10

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	6	3	5	1	8	2	9	7	10

```java
public int find_element_in_array(int[] array, int s) {
     for(int i = 0; i < array.length; i++) {
          if(array[i] == s) {
               return i;
          }
     }
     return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	6	3	5	1	8	2	9	7	10

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 5

4	6	3	5	1	8	2	9	7	10

4 operations (5 operations including the return)

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = ???

4	**6**	**3**	**5**	**1**	**8**	**2**	**9**	**7**	**10**

What is the **best-case scenario** for this algorithm (when would this have the shortest running time) ?

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 4

4	6	3	5	1	8	2	9	7	10

What is the **best-case scenario** for this algorithm (when would this have the shortest running time) ?

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = ?

4	6	3	5	1	8	2	9	7	10

What is the **worst-case scenario** for this algorithm (when would this have the longest running time) ?

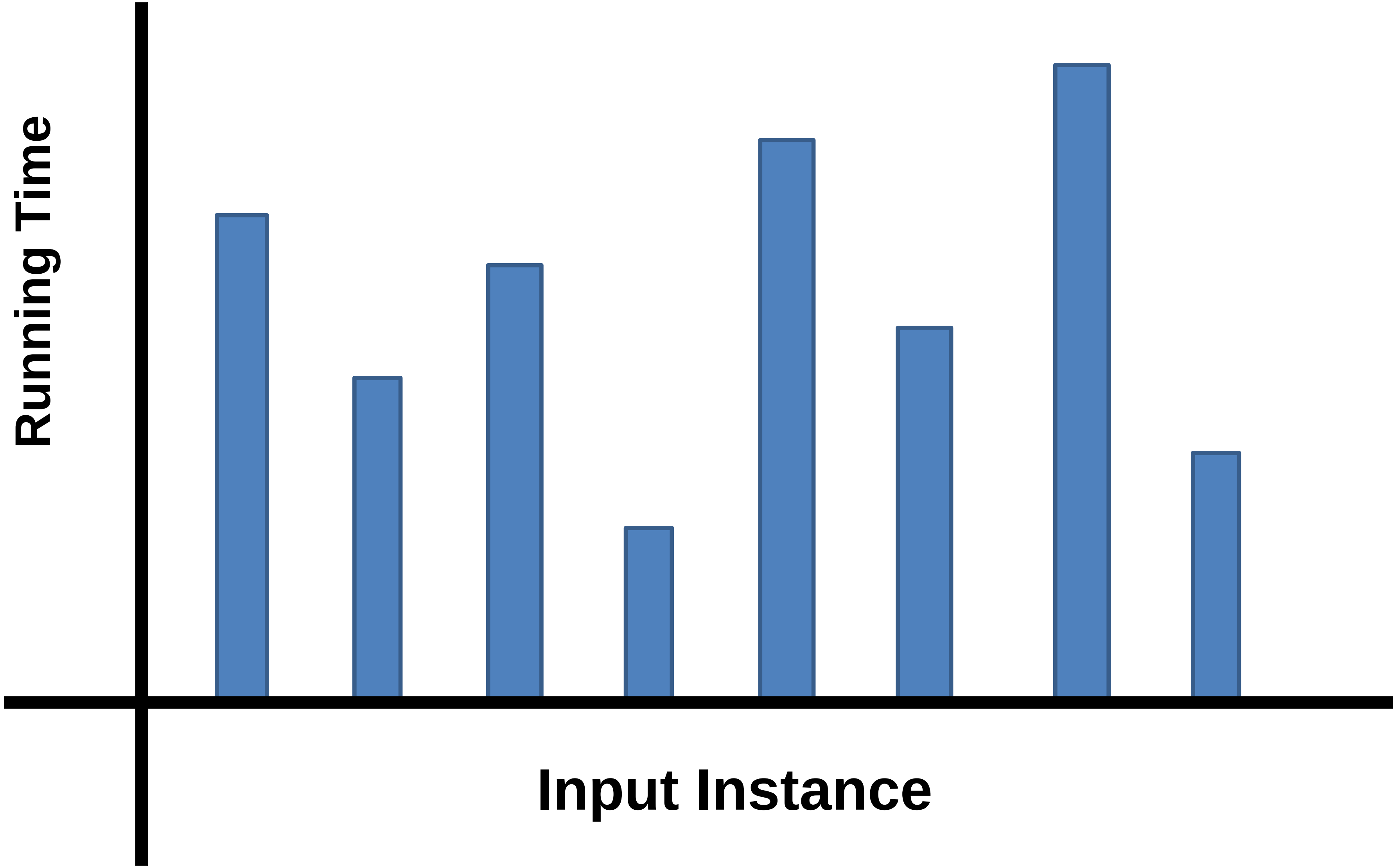
```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

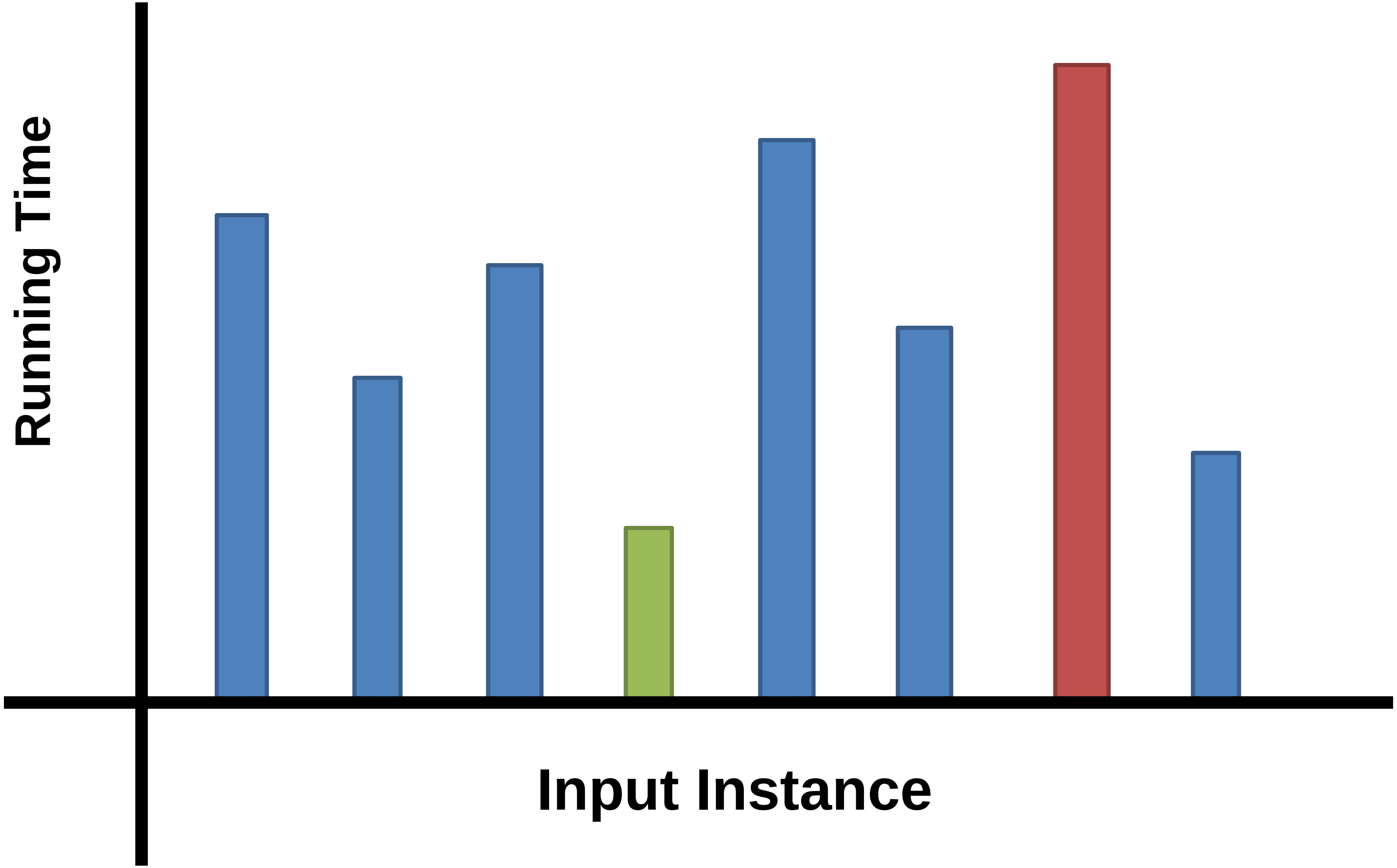
This algorithm finds the location (index) of an integer **S** in an array of size **N**

S = 11

4	**6**	**3**	**5**	**1**	**8**	**2**	**9**	**7**	**10**

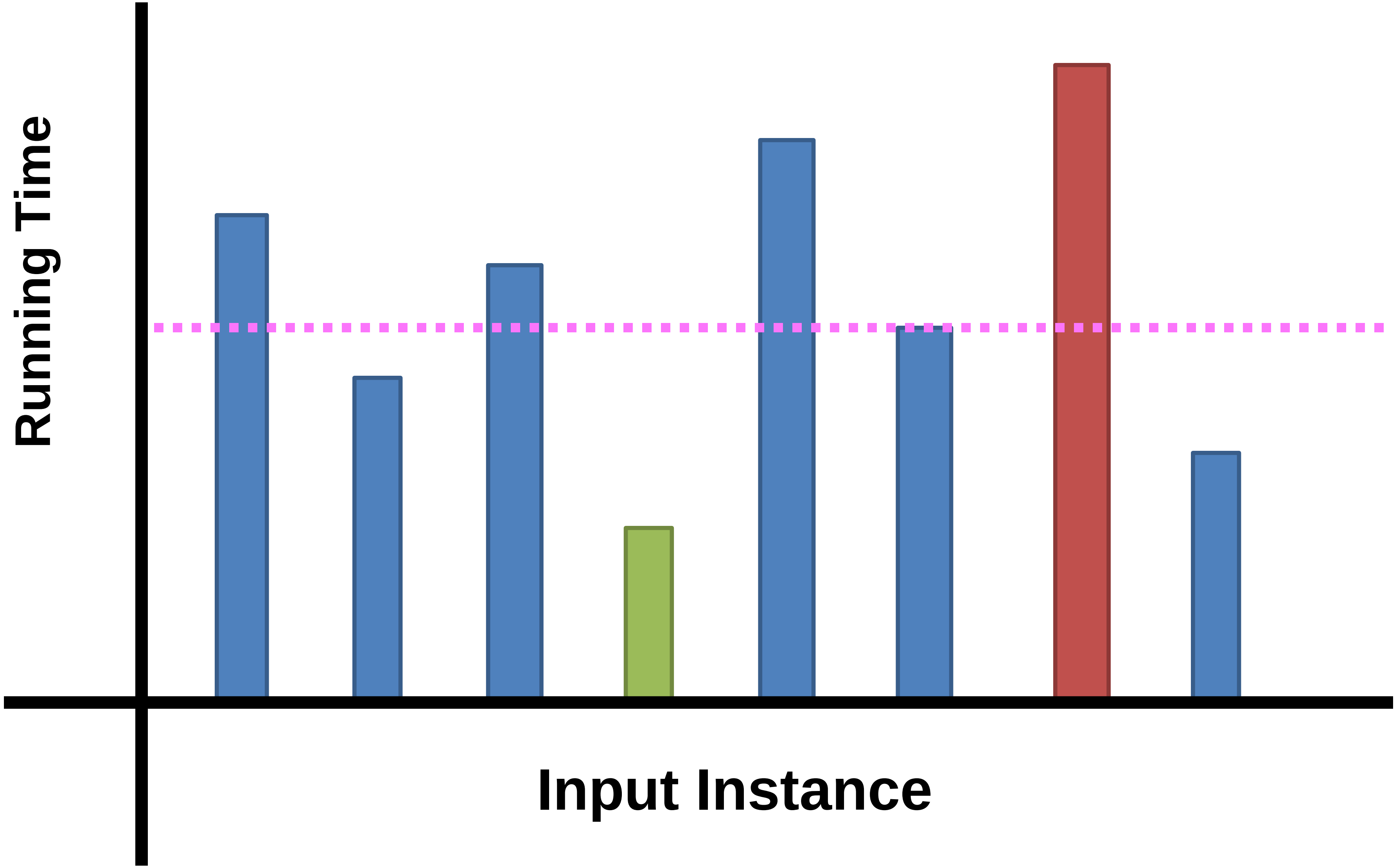
What is the **worst-case scenario** for this algorithm (when would this have the longest running time) ?


Running Time
Input Instance



Best Case Scenario

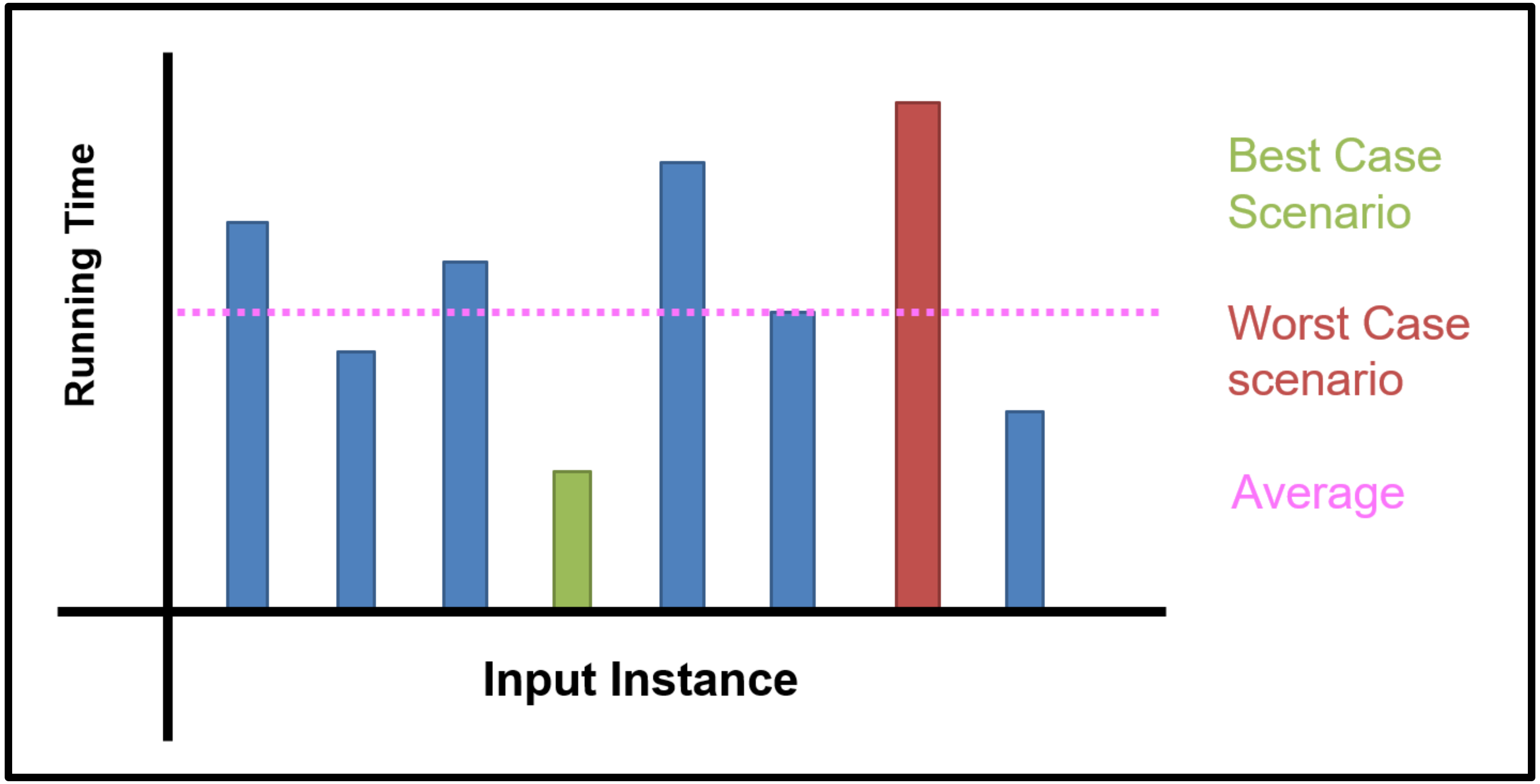
Worst Case scenario



Best Case Scenario

Worst Case scenario

Average



In computer science (and this class in particular), we will be focusing on stating running time in terms of **worst-case scenario**

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

To compute the running time of this algorithm, we will go line-by-line and state the running time of each operation (worst-case scenario)

At the end, add everything up to get the total running time

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

N

Worse case scenario, this for loop will run N times (N = size of the array)

```java
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {
		if(array[i] == s) {
			return i;
		}
	}
	return -1;
}
```

N

```
public int find_element_in_array(int[] array, int s) {
	for(int i = 0; i < array.length; i++) {    N
		if(array[i] == s) {    1
			return i;
		}
	}
	return -1;
}
```

This is a primitive operation, so it will always run in constant time

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {      N
        if(array[i] == s) {      1
            return i;
        }
    }
    return -1;
}
```

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {        N
        if(array[i] == s) {        1
            return i;        1
        }
    }
    return -1;
}
```

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {        N
        if(array[i] == s) {        1
            return i;        1
        }
    }
    return -1;        1
}
```

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {      N
        if(array[i] == s) {      1
            return i;      1
        }
    }
    return -1;      1
}
```

This whole block consists of only primitive operation, so we will group everything together and call it one single primitive operation

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {     N
        if(array[i] == s) {
            return i;                            1
        }
    }
    return -1;     1
}
```

Total Running Time =

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {      N
        if(array[i] == s) {
            return i;                              1
        }
    }
    return -1;      1
}
```

Total Running Time = N * 1 + 1

The if statement is inside the for loop, so we must multiply it by N (number of time the for loop runs)

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

N

1

1

Total Running Time = N + 1

```java
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {        N
        if(array[i] == s) {
            return i;                            1
        }
    }
    return -1;                                   1
}
```

Total Running Time = N + 1

O(N + 1)

"Big-Oh"

Big-O = Running Time in terms of worst-case scenario

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {    N
        if(array[i] == s) {
            return i;                            1
        }
    }
    return -1;    1
}
```

Total Running Time = N + 1

O(N + 1)   where N = Size of Array

"Big-Oh"

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant

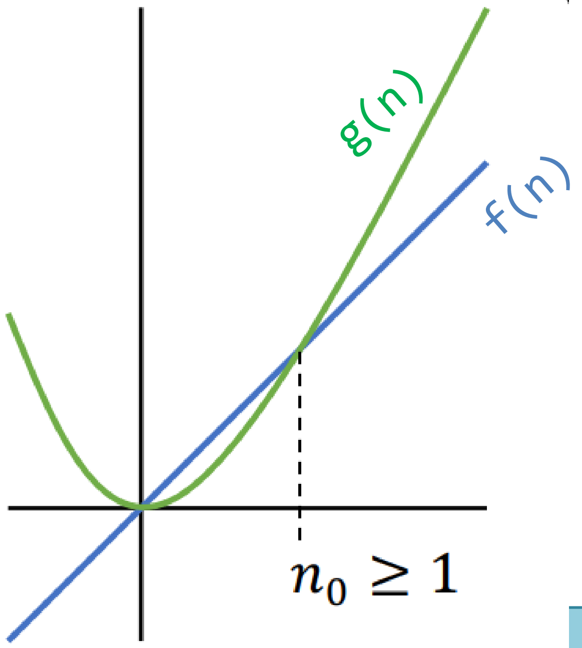
Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant



$$\forall n \geq 1, n^2 \geq n$$
$$\Rightarrow n \in O(n^2)$$

O -notation provides an upper bound on some function $f(n)$

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

In Big-O, we can drop non-dominant factors

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

In Big-O, we can drop non-dominant factors

$$x^2 + x + 10$$

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

$\boxed{\text{In Big-O, we can drop non-dominant factors}}$

$$x^2 + \underbrace{x + 10}$$

When X is really *really big*, these factors don't contribute very much at all

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

$\boxed{\text{In Big-O, we can drop non-dominant factors}}$

$$x^2 + \underbrace{x + 10} \in O(x^2)$$

When X is really *really big*, these factors don't contribute very much at all

x^2 is the dominating factor, so we can drop everything else

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

$\boxed{\text{In Big-O, we can drop non-dominant factors}}$

$$x^2 + \underbrace{x + 10}_{\text{When X is really } \textit{really big}\text{, these factors don't contribute very much at all}} \in O(x^2)$$

When X is really *really big*, these factors don't contribute very much at all

$$\boxed{\cancel{x^2 + x + 10 = O(x^2)}}$$

Quick warning on notation

= ✗ ∈ ✓

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

In Big-O, we can drop non-dominant factors

????

```
public int find_element_in_array(int[] array, int s) {
    for(int i = 0; i < array.length; i++) {
        if(array[i] == s) {
            return i;
        }
    }
    return -1;
}
```

N

1

1

Total Running Time = N + 1

O(N + 1)   where N = Size of Array

O(N)   where N = Size of Array


"do you understand it now?"
ohhhhhhhhh yeah now i get it
no

```java
int[] newArray = new int[myArray.length + 1];

for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Create a new array that is one spot larger

Fill new array with contents of old array

Add new value to array and update reference variable

What is the running time of this algorithm?

```java
int[] newArray = new int[myArray.length + 1];

for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Create a new array that is one spot larger

Fill new array with contents of old array

Add new value to array and update reference variable

What is the running time of this algorithm?

We will find the time complexity for each operation!

```
int[] newArray = new int[myArray.length + 1];
```

???

```
for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time =

```
int[] newArray = new int[myArray.length + 1];
```

O(n)

```
for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time = n

```
int[] newArray = new int[myArray.length + 1];

for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

O(n)

Total Running Time = n

Algorithm Analysis: Adding value to an Array/ArrayList

```
int[] newArray = new int[myArray.length + 1];
```

O(n)

```
for(int i = 0; i < myArray.length; i++) {
      newArray[i] = myArray[i];
}
```

O(n)

```
int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time = n + n

```
int[] newArray = new int[myArray.length + 1];      O(n)

for(int i = 0; i < myArray.length; i++) {      O(n)
      newArray[i] = myArray[i];      O(1)
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time = n + n * 1

```
int[] newArray = new int[myArray.length + 1];        O(n)

for(int i = 0; i < myArray.length; i++) {           O(n)
      newArray[i] = myArray[i];                     O(1)
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

When do we add? When do multiply?

Sequential Operations = Add
Nested Operations (in a loop) = Multiply

Total Running Time = n + n * 1

```
int[] newArray = new int[myArray.length + 1];         O(n)

for(int i = 0; i < myArray.length; i++) {          O(n)
      newArray[i] = myArray[i];          O(1)
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time = n + n * 1

```
int[] newArray = new int[myArray.length + 1];        O(n)

for(int i = 0; i < myArray.length; i++) {            O(n)
      newArray[i] = myArray[i];                      O(1)
}

int new_value = 4;                                   O(1)
newArray[myArray.length] = new_value;
myArray = newArray;
```

Total Running Time = n + n * 1 + 1

```
int[] newArray = new int[myArray.length + 1];      O(n)

for(int i = 0; i < myArray.length; i++) {          O(n)
      newArray[i] = myArray[i];                    O(1)
}

int new_value = 4;                                 O(1)
newArray[myArray.length] = new_value;              O(1)
myArray = newArray;
```

Total Running Time = n + n * 1 + 1 + 1

```java
int[] newArray = new int[myArray.length + 1];      O(n)

for(int i = 0; i < myArray.length; i++) {          O(n)
      newArray[i] = myArray[i];                    O(1)
}

int new_value = 4;                                 O(1)
newArray[myArray.length] = new_value;              O(1)
myArray = newArray;                                O(1)
```

Total Running Time = n + n * 1 + 1 + 1 + 1

```
int[] newArray = new int[myArray.length + 1];      O(n)

for(int i = 0; i < myArray.length; i++) {          O(n)
      newArray[i] = myArray[i];                     O(1)
}

int new_value = 4;                                  O(1)
newArray[myArray.length] = new_value;               O(1)
myArray = newArray;                                 O(1)
```

Total Running Time = n + n * 1 + 1 + 1 + 1

= 2n + 3

```
int[] newArray = new int[myArray.length + 1];        O(n)

for(int i = 0; i < myArray.length; i++) {             O(n)
        newArray[i] = myArray[i];                     O(1)
}

int new_value = 4;                                    O(1)
newArray[myArray.length] = new_value;                 O(1)
myArray = newArray;                                   O(1)
```

Total Running Time = n + n * 1 + 1 + 1 + 1

= 2n + 3

O(2n) where n is the size of the array

Big-O

Notation used to describe the running time of an algorithm in terms of worse case scenario

Traits of Big-O-Notation:

In Big-O, we can drop non-dominant factors

In Big-O, we can drop multiplicative constants

```
int[] newArray = new int[myArray.length + 1];        O(n)

for(int i = 0; i < myArray.length; i++) {            O(n)
      newArray[i] = myArray[i];                       O(1)
}

int new_value = 4;                                   O(1)
newArray[myArray.length] = new_value;                O(1)
myArray = newArray;                                  O(1)
```

When we write algorithms, we should still be *aware of* these coefficients

Total Running Time = n + n * 1 + 1 + 1 + 1

= 2n + 3

~~**O(2n) where n is the size of the array**~~ → **O(n) where n is the size of the array**

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
	if(head == null) {
		head = newNode;
	}
	else {
		Node current = head;
		while(current.getNext() != null) {
			current = current.getNext();
		}
		current.setNext(newNode);
	}
}
```

What is the running time of this algorithm?

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
     if(head == null) {
          head = newNode;
     }
     else {
         Node current = head;
         while(current.getNext() != null) {
             current = current.getNext();
         }
         current.setNext(newNode);
     }
}
```

Total Running Time =

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
	if(head == null) {
		head = newNode;
	}
	else {
		Node current = head;
		while(current.getNext() != null) {
			current = current.getNext();
		}
		current.setNext(newNode);
	}
}
```

O(1)

Total Running Time = 1 +

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
     if(head == null) {
          head = newNode;
     }
     else {
          Node current = head;
          while(current.getNext() != null) {
               current = current.getNext();
          }
          current.setNext(newNode);
     }
}
```

O(1)

O(1)

Total Running Time = 1 + 1

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
      if(head == null) {
            head = newNode;
      }
      else {
          Node current = head;
          while(current.getNext() != null) {
              current = current.getNext();
          }
          current.setNext(newNode);
      }
}
```

O(1)

O(1)

O(n)

Total Running Time = 1 + 1 + n

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
     if(head == null) {                    O(1)
          head = newNode;
     }
     else {
          Node current = head;               O(1)
          while(current.getNext() != null) {   O(n)
               current = current.getNext();   O(1)
          }
          current.setNext(newNode);
     }
}
```

Total Running Time = 1 + 1 + n * 1

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
      if(head == null) {                         O(1)
            head = newNode;
      }
      else {
           Node current = head;                  O(1)
           while(current.getNext() != null) {    O(n)
                current = current.getNext();     O(1)
           }
           current.setNext(newNode);             O(1)
      }
}
```

Total Running Time = 1 + 1 + n * 1 + 1

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
      if(head == null) {
           head = newNode;
      }
      else {
          Node current = head;
          while(current.getNext() != null) {
              current = current.getNext();
          }
          current.setNext(newNode);
      }
}
```

O(1)

O(1)

O(n)

O(1)

O(1)

Total Running Time = 1 + 1 + n * 1 + 1

= n + 3

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
      if(head == null) {
           head = newNode;
      }
      else {
          Node current = head;
          while(current.getNext() != null) {
              current = current.getNext();
          }
          current.setNext(newNode);
      }
}
```

O(1)

O(1)

O(n)

O(1)

O(1)

Total Running Time = 1 + 1 + n * 1 + 1

= n + 3

∊ O(n) where n is ????

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
     if(head == null) {
          head = newNode;
     }
     else {
          Node current = head;
          while(current.getNext() != null) {
               current = current.getNext();
          }
          current.setNext(newNode);
     }
}
```

O(1)

O(1)

O(n)

O(1)

O(1)

Total Running Time = 1 + 1 + n * 1 + 1

= n + 3

∊ **O(n) where n is the number of nodes in the LL**

Algorithm Analysis: Adding Node to end of Singly Linked List (no `tail` pointer)

```
public void addToBack(Node newNode) {
      if(head == null) {
           head = newNode;
      }
      else {
          Node current = head;
          while(current.getNext() != null) {
              current = current.getNext();
          }
          current.setNext(newNode);
      }
}
```

O(1)

O(1)

O(n)

O(1)

O(1)

Total Running Time = 1 + 1 + n * 1 + 1
= n + 3

$\in$ **O(n) where n is the number of nodes in the LL**

"Worst case scenario, we have to go through all the nodes in the LL to add something at the end"

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

What is the running time of this algorithm?

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

O(1)

Total Running Time =

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

O(1)

O(1)

Total Running Time =

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

O(1)

O(1)

O(1)

Total Running Time = 1 + 1 + 1

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

O(1)

O(1)

O(1)

"The number of operations required for this algorithm is the same no matter the input"

Total Running Time = 1 + 1 + 1

$\in$ **O(1)**

Algorithm Analysis: Adding Node to end of Singly Linked List (`tail` pointer)

```
public void addToBack(Node newNode) {

    if(head == null && tail == null) {
        head = newNode;
        tail = newNode;
    }
    else {
        tail.setNext(newNode);
        tail = newNode;
    }

}
```

O(1)

O(1)

O(1)

"The number of operations required for this algorithm is the same no matter the input"

Total Running Time = 1 + 1 + 1

$\in$ **O(1)**

3,000,000 Nodes = 3 operations, 10 Nodes = 3 operations

Algorithm Analysis: Printing out funky number triangle

`print_number_triangle(4);`

```
1
22
333
4444
```

`print_number_triangle(7);`

```
1
22
333
4444
55555
666666
7777777
```

`print_number_triangle(9);`

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
      for(int i = 1; i < n + 1; i++) {
           for(int j = 0; j < i; j++) {
               System.out.print(i);
           }
           System.out.println();
      }
}
```

print_number_triangle(4); *print_number_triangle*(7); *print_number_triangle*(9);

```
1
22
333
4444
```

```
1
22
333
4444
55555
666666
7777777
```

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
      for(int i = 1; i < n + 1; i++) {
           for(int j = 0; j < i; j++) {
               System.out.print(i);
           }
           System.out.println();
      }
}
```

O(n)

Total Running Time =

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
      for(int i = 1; i < n + 1; i++) {
            for(int j = 0; j < i; j++) {
                System.out.print(i);
            }
            System.out.println();
      }
}
```

O(n)

O(n)

Total Running Time =

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
      for(int i = 1; i < n + 1; i++) {       O(n)
           for(int j = 0; j < i; j++) {      O(n)
               System.out.print(i);          O(1)
           }
           System.out.println();
      }
}
```

Total Running Time =

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
	for(int i = 1; i < n + 1; i++) {      O(n)
		for(int j = 0; j < i; j++) {      O(n)
			System.out.print(i);          O(1)
		}
		System.out.println();             O(1)
	}
}
```

Total Running Time =

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
	for(int i = 1; i < n + 1; i++) {      O(n)
		for(int j = 0; j < i; j++) {      O(n)
			System.out.print(i);          O(1)
		}
		System.out.println();             O(1)
	}
}
```

Total Running Time = N * ((N * 1) * 1)

Algorithm Analysis: Printing out funky number triangle

```
public static void print_number_triangle(int n) {
	for(int i = 1; i < n + 1; i++) {
		for(int j = 0; j < i; j++) {
			System.out.print(i);
		}
		System.out.println();
	}
}
```

O(n)

O(n)

O(1)

O(1)

Total Running Time = N * ((N * 1) * 1)

= N^2

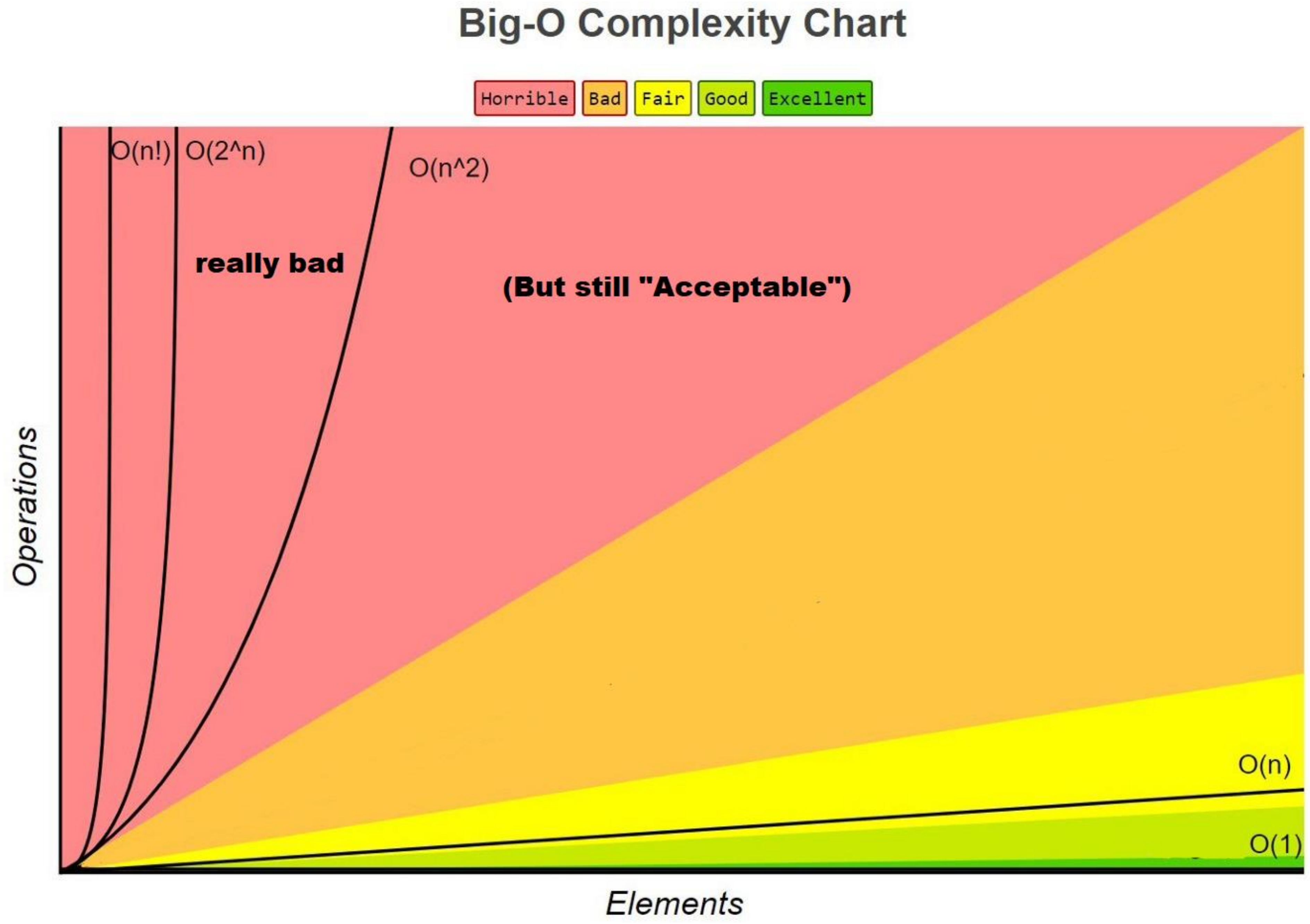
Algorithm Analysis: Printing out funky number triangle

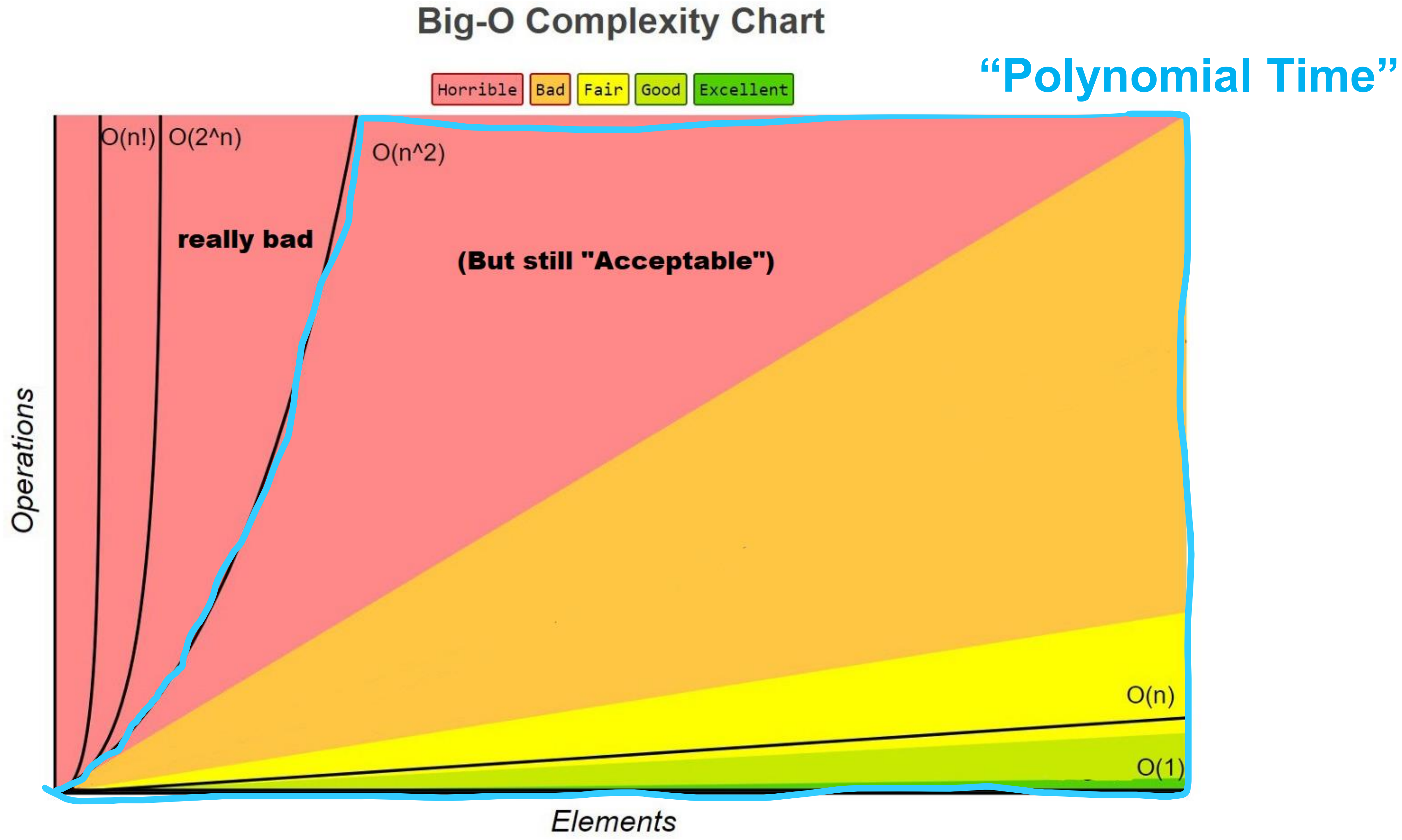
```
public static void print_number_triangle(int n) {
	for(int i = 1; i < n + 1; i++) {        O(n)
		for(int j = 0; j < i; j++) {    O(n)
			System.out.print(i);    O(1)
		}
		System.out.println();           O(1)
	}
}
```

Total Running Time = N * ((N * 1) * 1)

$$= N^2$$

$$\in O(n^2)$$


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
really bad
(But still "Acceptable")
Operations
O(n)
O(1)
Elements


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
"Polynomial Time"
O(n!)
O(2^n)
O(n^2)
really bad
(But still "Acceptable")
Operations
O(n)
O(1)
Elements