# Problem Solving for Computer Science

IS51021C

Goldsmiths Computing

March 1, 2021

Wk 7

# Today

*Analysing Algorithms*

1. Review of RAM model
2. Growth of functions and Big O notation
3. Worst-case analysis

# Today

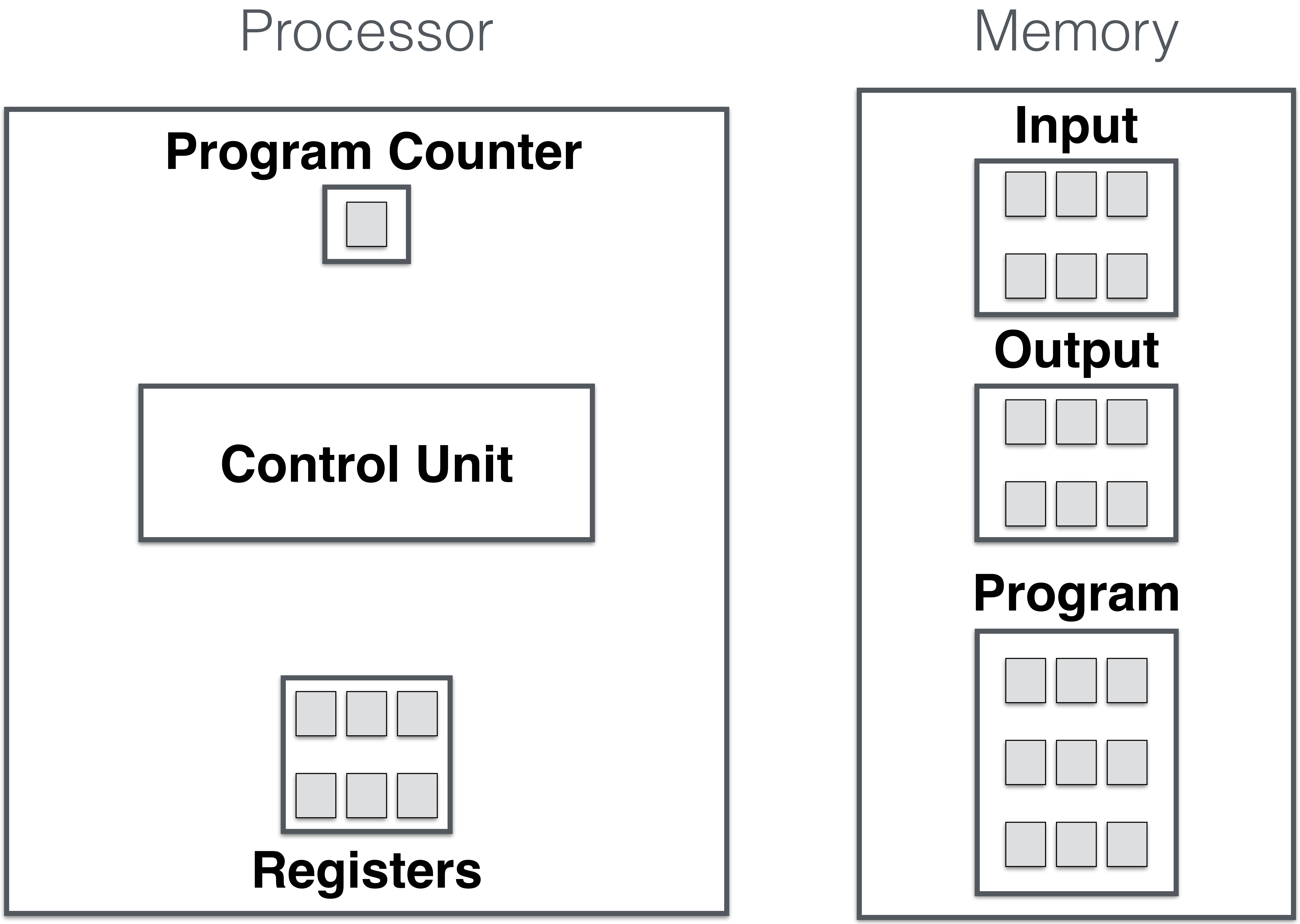
*Analysing Algorithms*

1. **Review of RAM model**

2. Growth of functions and Big O notation

3. Worst-case analysis

Random Access Machine

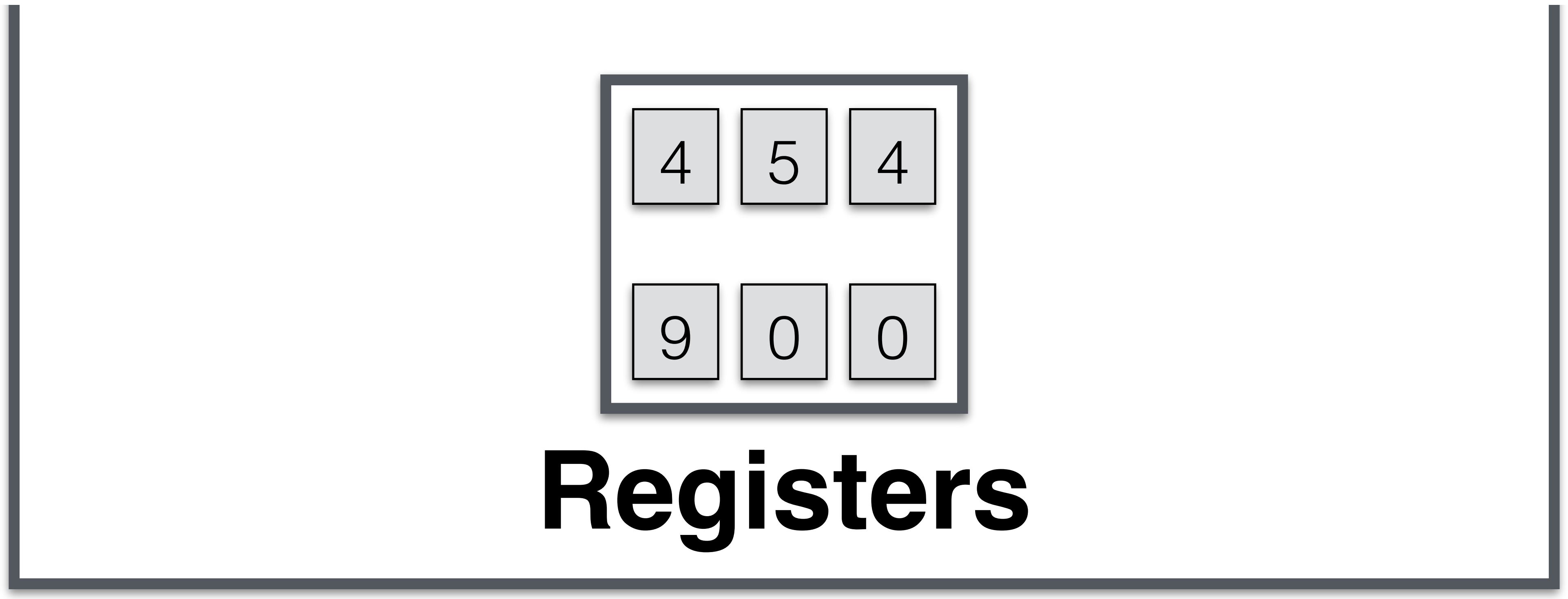


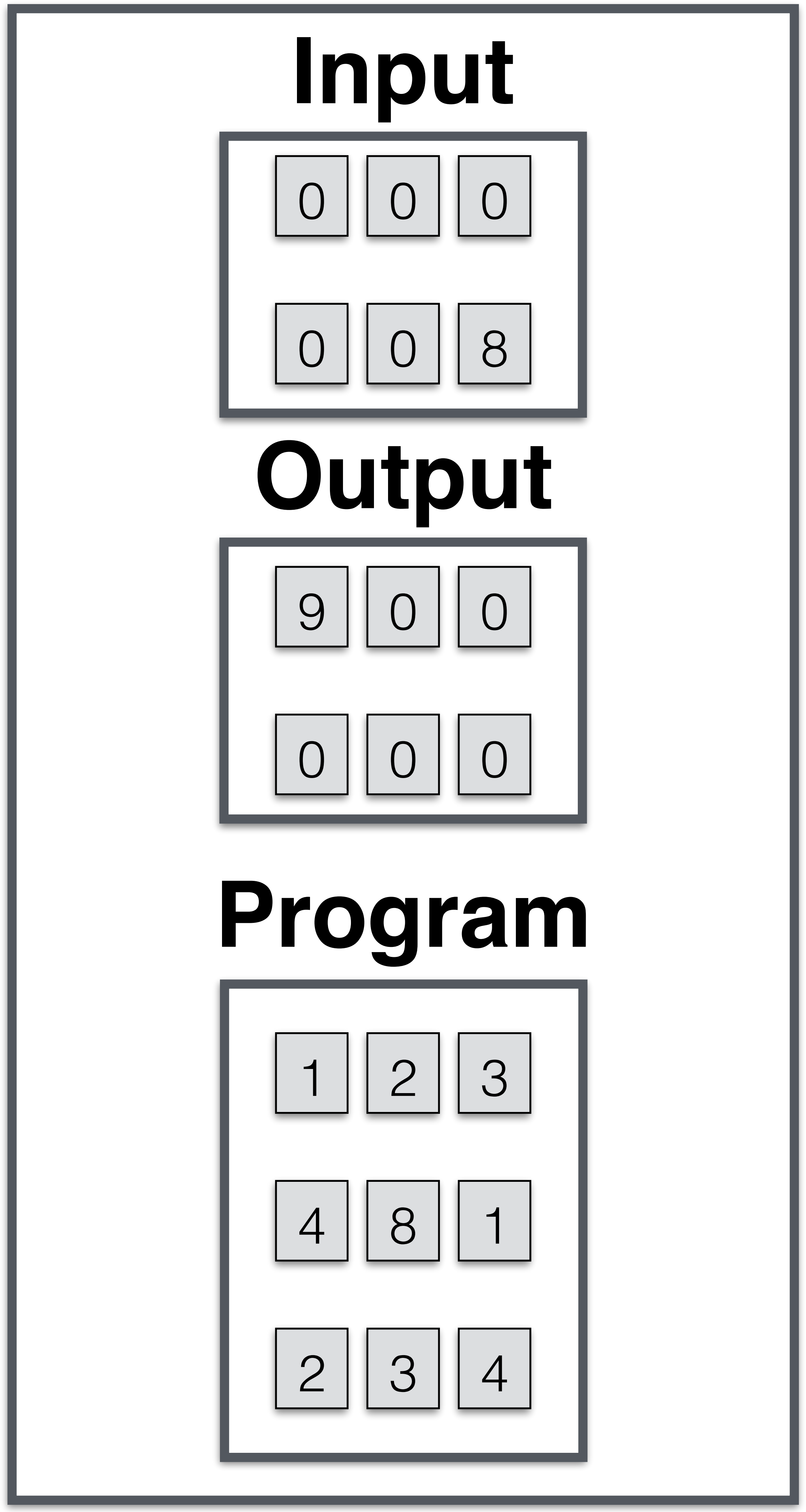
# Random Access Machine

Each memory unit can store an arbitrary integer

Must be non-negative for Program Counter

Depending on values, Control Unit does an operation



Control Unit can:

Read and write values in single memory units

Do simple arithmetic (add, subtract, multiply, divide)

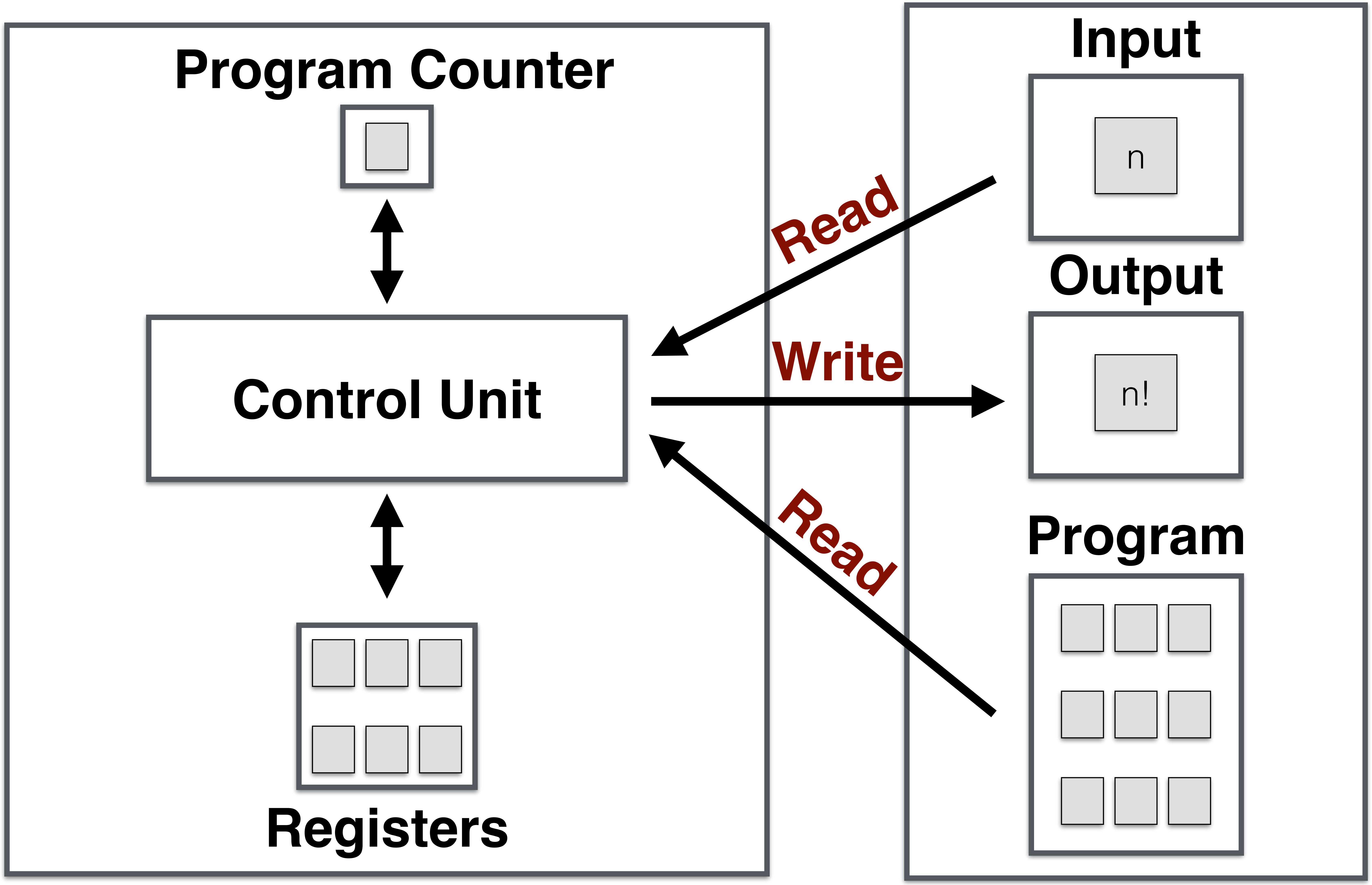
**Above operations done in one time-step**

Perform conditional operations: if then conditionals

**One time-step for comparison in if statement**

# Case study: Calculating the factorial of a number

$$n! = n * (n-1) * (n-2) * \ldots * 1$$



$n! = n * (n-1) * (n-2) * \ldots * 1$

Useful for calculating permutations of objects

```
function factorial(n) {
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    return a;
}
```

$n! = n * (n-1) * (n-2) * \dots * 1$

```
function factorial(n) {
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    return a;
}
```

Imagine we call *factorial(n)* for some value of *n*

How many operations are required to implement in RAM model?

$N_{op}$ = number of operations

$n! = n * (n-1) * (n-2) * \ldots * 1$

```
function factorial(n) {
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    return a;
}
```

Writes value to output in memory and stops

Imagine we call *factorial(n)* for some value of *n*

How many operations are required to implement in RAM model?

$N_{op}$ = number of operations

```javascript
function factorialCount(n) {
    var count = 0;
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    //return a;
    return count;
}
```

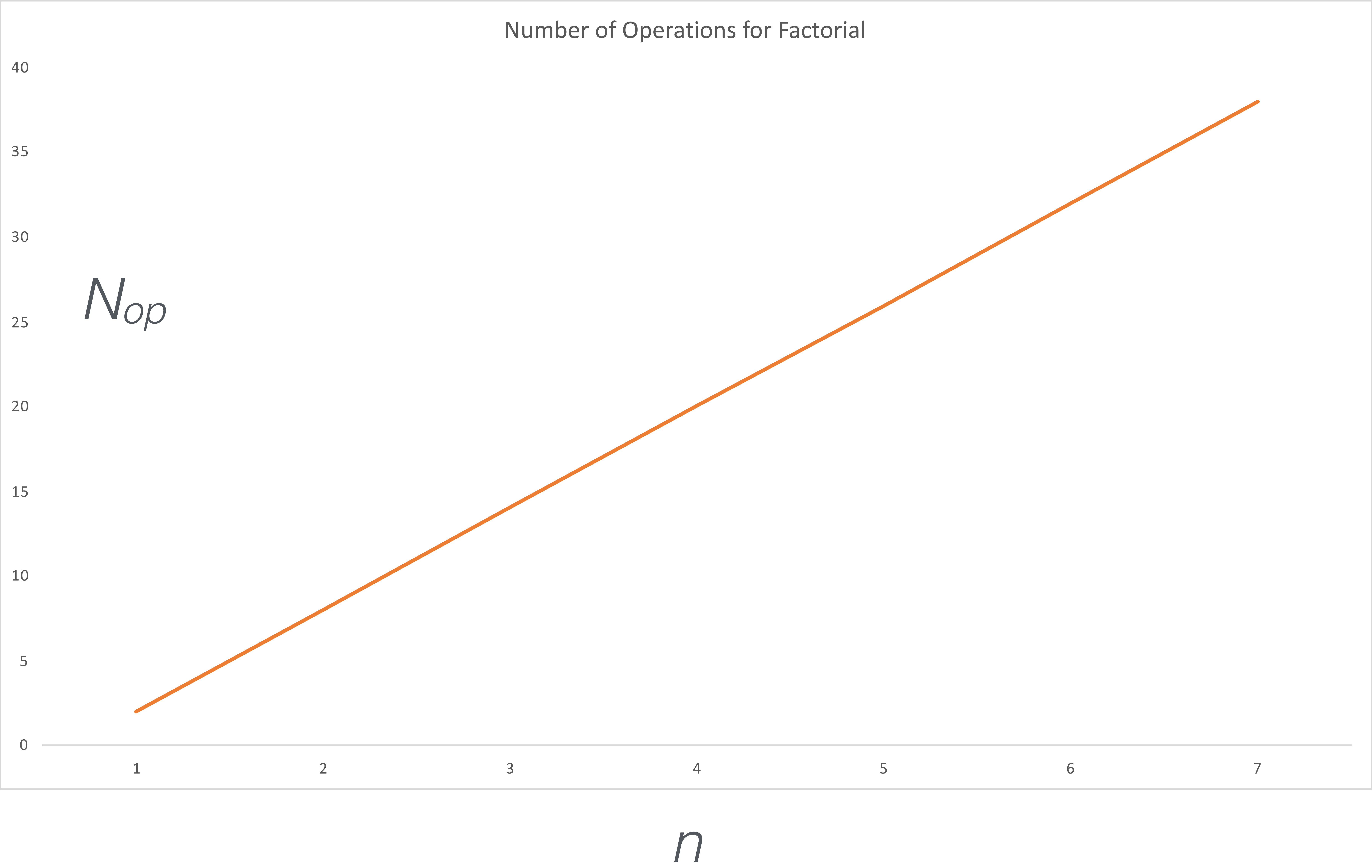
In factorialCount,
let's put in:

*count++, or*
*count+=2, or*
*count+=3 …*

for every RAM
operation in original
factorial function

$N_{op}$ will be final value of count

# A possible count of operations



$$n! = n * (n-1) * (n-2) * \ldots * 1$$

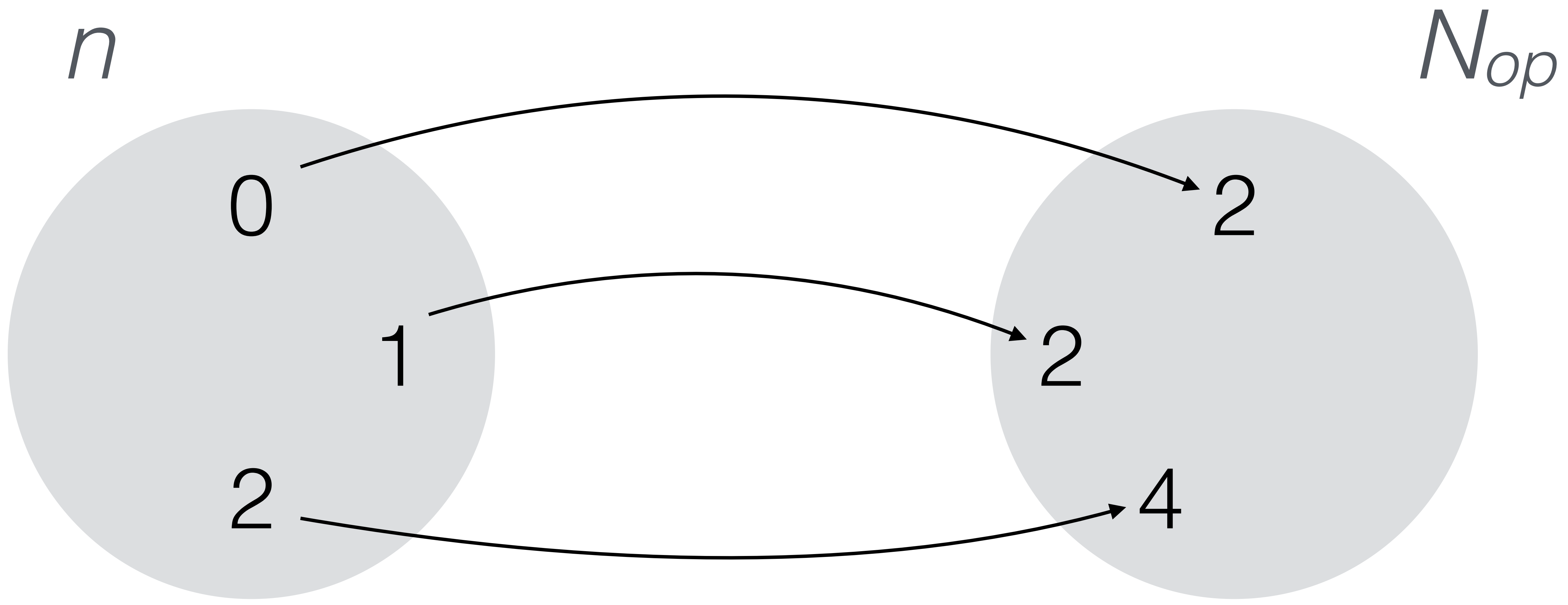
Number of operations depends on n

$$N_{op} = f(n)$$

n! = n * (n-1) * (n-2) * … * 1

Number of operations depends on n

$N_{op} = f(n)$

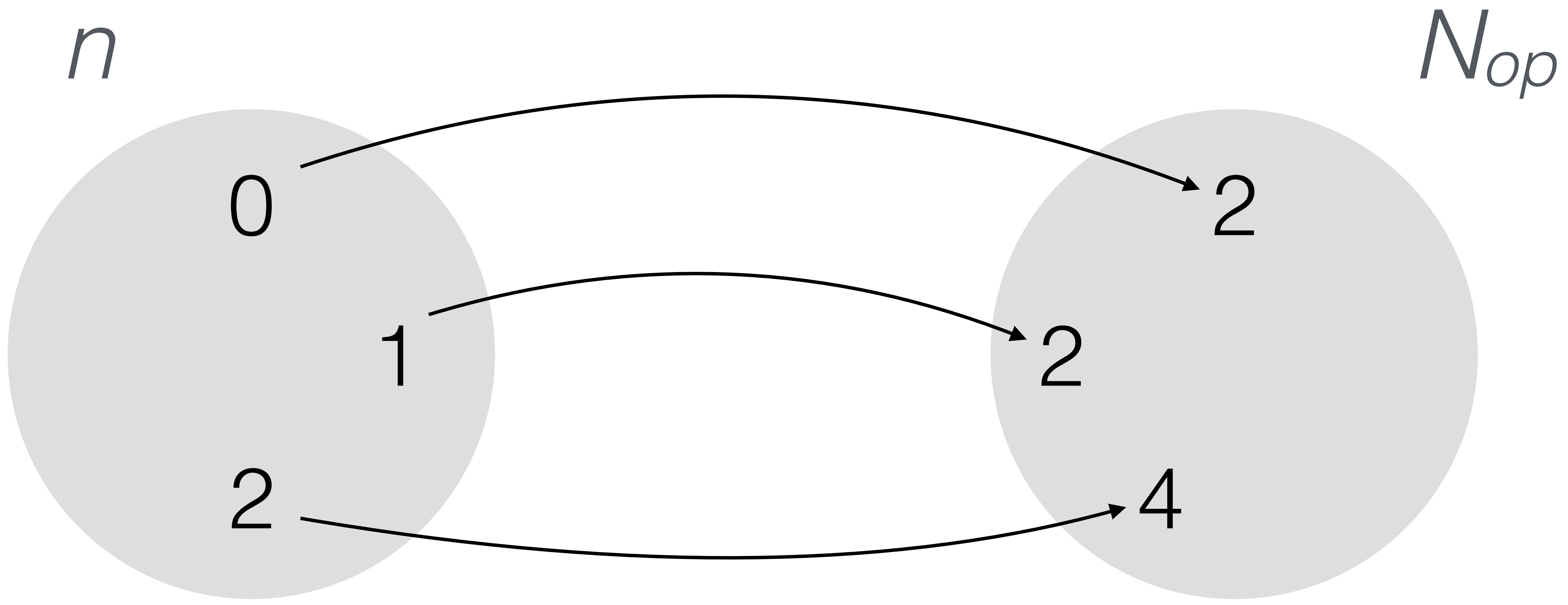



Mathematical function from integers to non-negative integers

A formula that tells us how much an implementation “costs” n! = n * (n-1) * (n-2) * … * 1

Number of operations depends on n

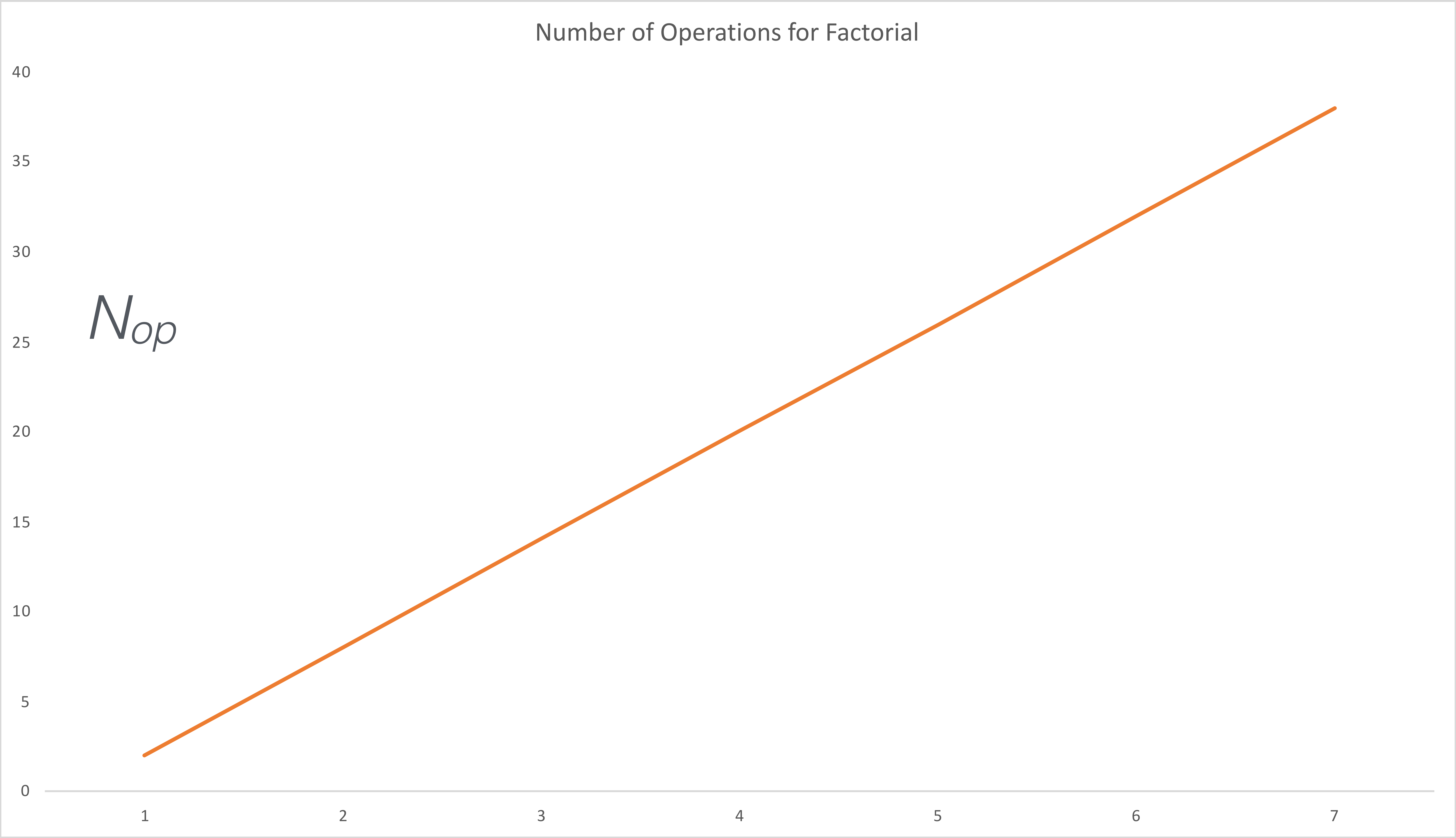
$N_{op} = f(n)$




Mathematical function from integers to non-negative integers

What is this function?

# A possible count of operations




$n$

$$\text{e.g. } N_{op} = f(n) = 2 + 6(n-1)$$

$$N_{op} = f(n) = 2 + 6(n-1)$$

$$N_{op} = f(n) = 2 + 6(n-1)$$

$$N_{op} = f(n) = 3 + 5(n-1)$$

$$N_{op} = f(n) = 2 + 7(n-1)$$

…

There are different possibilities depending on how we count operations, e.g. if a variable assignment is just one operation or more

These are technicalities that introduce constants

The universal thing: **proportionality to *n***

As *n* increases so does the number of operations

$$N_{op} = f(n) = 2 + 6(n-1)$$

$$N_{op} = f(n) = 3 + 5(n-1)$$

$$N_{op} = f(n) = 2 + 7(n-1)$$

…

There are different possibilities depending on how we count operations, e.g. if a variable assignment is just one operation or more

These are technicalities that introduce constants

The universal thing: **proportionality to *n***

*Why* is it proportional to *n*?

$$N_{op} = f(n) = 2 + 6(n\text{-}1)$$

$$N_{op} = f(n) = 3 + 5(n\text{-}1)$$

$$N_{op} = f(n) = 2 + 7(n\text{-}1)$$

$$\ldots$$

There are different possibilities depending on how we count operations, e.g. if a variable assignment is just one operation or more

These are technicalities that introduce constants

The universal thing: **proportionality to *n***

**In analysing algorithms, we want general statements**

Recap:

- Number $n$ as input
- To compute the factorial, we presented a program
- In the RAM implementation of this program, we counted the number of operations $N_{op}$ for each $n$
- $N_{op}$ is defined by a function dependent on $n$

As n increases time taken to compute factorial grows **proportionally**: we need to wait longer and longer for the solution as n gets larger and larger

There were constants hovering around making things messy: ignore them as they don't add to our understanding…

# Today

*Analysing Algorithms*

1. Review of RAM model
2. **Growth of functions and Big O notation**
3. Worst-case analysis

To avoid discussing specific constants, we use:

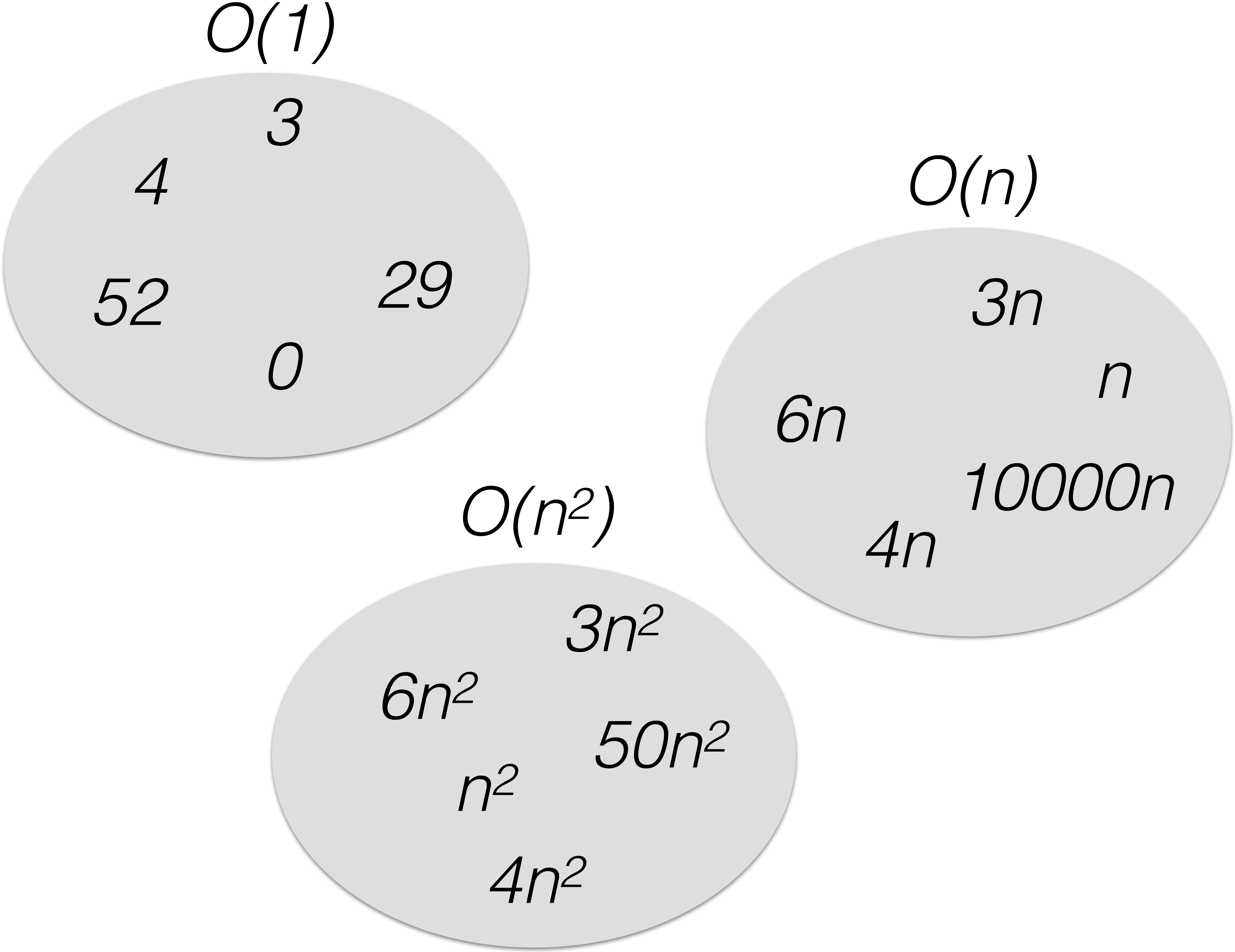
**"Big O" notation**

Denotes the *most useful* information about a function of one variable

**"Treats all multiplying constants as if equal to 1"**

Functions belong to a "Big O" class

e.g. $f(n) = 6n$ belongs to $O(n)$
$f(n) = 5$ belongs to $O(1)$


$O(1)$
3
4
52
29
0
$O(n)$
$3n$
$n$
$6n$
$10000n$
$4n$
$O(n^2)$
$3n^2$
$6n^2$
$50n^2$
$n^2$
$4n^2$

# "Big O" notation goes further

More complicated functions

e.g. $f(n) = n^3 + 26n^2 + 34n + 2$

1) Treat all non-zero constants as 1

$\longrightarrow$ $n^3 + n^2 + n + 1$

2) Consider **fastest growing part as *n* increases**

$\longrightarrow$ $O(n^3)$

# "Big O" notation goes further

Consider **fastest growing part as n increases**

Fastest growing: how the gap between $f(n)$ and $f(n+1)$ changes with bigger $n$

Does it get larger? Does it stay the same? Does it shrink?

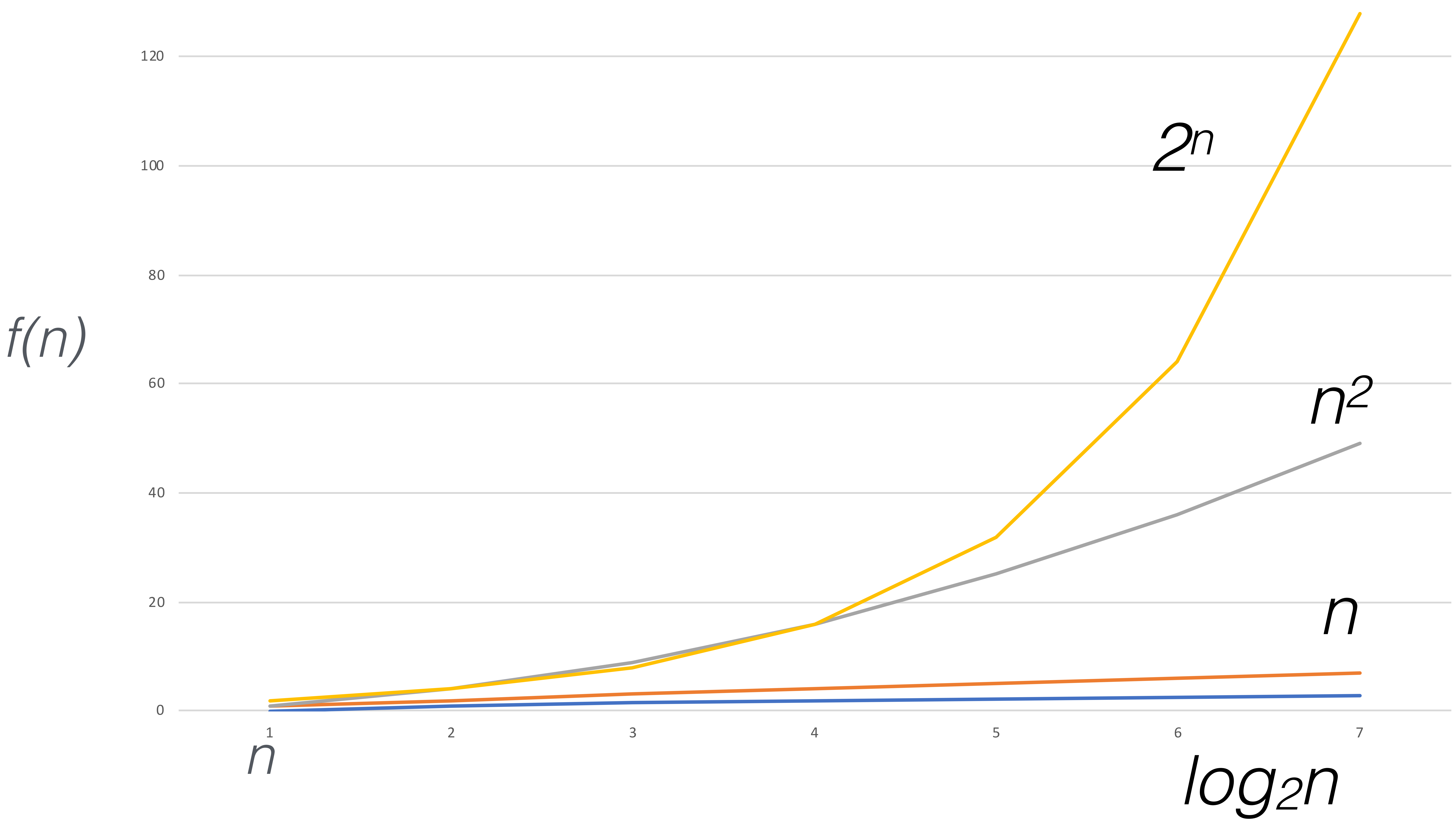
$f(n+1) - f(n)$

| | |
|---|---|
| Exponential | |
| $O(2^n)$ | |
| Polynomial | |
| $O(n^2)$ | |
| $O(n)$ | |
| Logarithmic | |
| $O(\log_2 n)$ | |

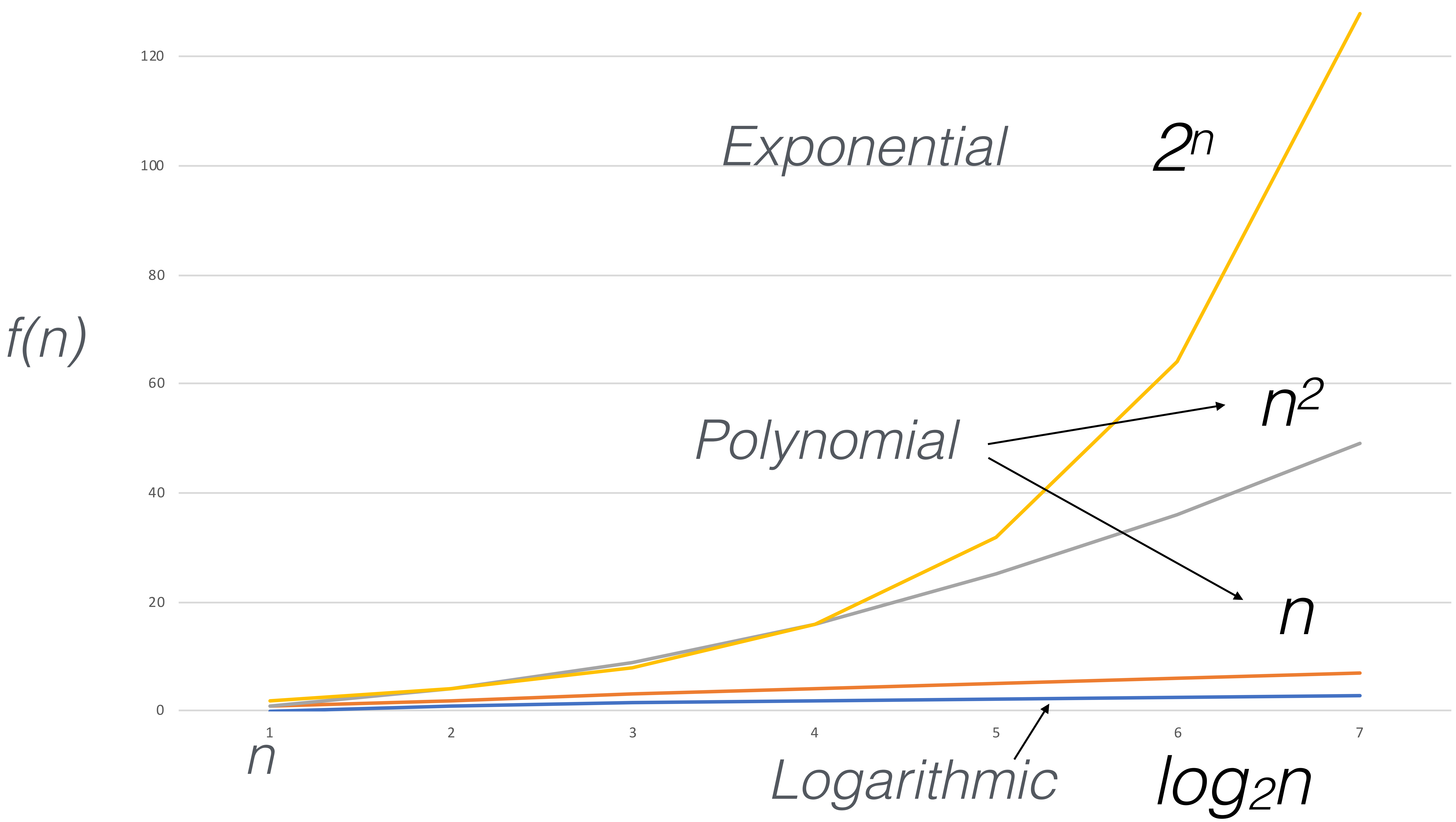
$$f(n+1) - f(n)$$

| | |
|---|---|
| Exponential $O(2^n)$ | $2^{n+1} - 2^n = 2^n(2-1)=2^n$ doubling of the difference |
| Polynomial $O(n^2)$ | $(n+1)^2 - n^2 = n + 1$ linear |
| $O(n)$ | $(n+1) - n = 1$ constant |
| Logarithmic $O(\log_2 n)$ | $\log_2(n+1) - \log_2 n = \log_2((n+1)/n) < 1.5/n$ inverse linear |

# Fastest growing functions



# Fastest growing functions



Every polynomial O($n^k$) for $k>0$ **grows faster** than O($n^c$) for all $c < k$

Every polynomial O($n^k$) for $k>0$ **grows faster** than logarithmic class $O(\log_2 n)$

Every exponential O($k^n$) for $k>1$ **grows faster** than every polynomial class O($n^k$) for $k>0$

Every polynomial O($n^k$) for $k>0$ **grows faster** than O($n^c$) for all $c < k$

*e.g. O($n^3$) grows faster than O($n^2$) and O($n$)*

Every polynomial O($n^k$) for $k>0$ **grows faster** than logarithmic class O($\log_2 n$)

*e.g. O($\sqrt{n}$) grows faster than O($\log_2 n$)*

Every exponential O($k^n$) for $k>1$ **grows faster** than every polynomial class O($n^k$) for $k>0$

*e.g. O($1.01^n$) grows faster than O($n^{100}$)*

# “Big O” notation recipe

1) Treat all non-zero constants as 1
2) Include in brackets only the **fastest growing part as *n* increases**

$O(1)$

$3+2$

$4$

$52$

$29+1$

$0$

$O(n)$

$3n+4$

$6n$

$10000n$

$4n+10000$

$O(n^2)$

$3n^2+1$

$n^2$

$50n^2+n$

$4n^2+100$

$O(2^n)$

$2^n$

$2^n+n$

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

$k > 2$

$$O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^k) < O(2^n) < O(2^{2n})$$

Functions in “smaller class” do not grow faster than functions in “bigger class”

Functions in “smaller class” do grow **at most as fast** as functions in “bigger class”

e.g. functions in $O(n)$ definitely **do not grow faster** than functions $O(2^n)$

… there are functions in $O(n)$ that **are not** in $O(1)$

$k>2$

$$O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^k) < O(2^n) < O(2^{2n})$$

"Big O" really says: function will grow **at most as fast** as the thing in the brackets

e.g. $f(n) = 3n + 2$ will be in $O(n)$, **AND** also in $O(2^n)$
**BUT** not in $O(\log_2 n)$

Whatever function you have will belong in a class and then many more

$$k>2$$

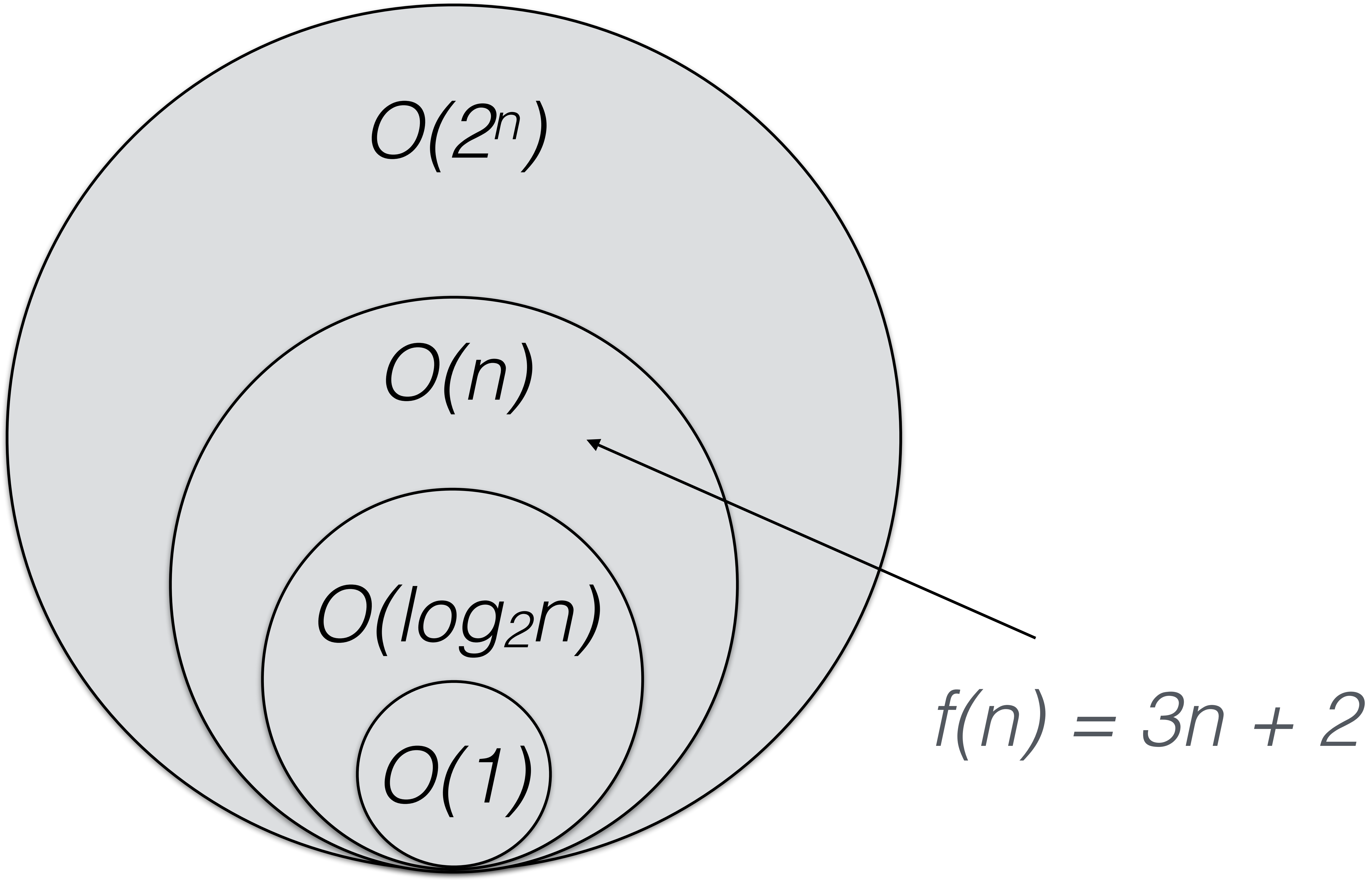
$$O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^k) < O(2^n) < O(2^{2n})$$

**Every "Big O" class is a set of functions**

The faster the function in the brackets grows, the bigger the set

Each smaller set is contained in the next larger set

Set inclusions



# Bases

What about $O(\log_3 n)$?

**It doesn’t matter which base you choose as long as it is larger than 1**

e.g. $O(\log_2 n) = O(\log_3 n)$

Why?

What about $O(10^n)$ instead of $O(2^n)$?

Discuss this during the **Review Seminar**

# Admin

- Fifth quiz available today from 4pm
  - Deadline for fifth quiz: **15th March 4pm**
  - Sixth quiz available next Monday

- Sudoku assignment
  - Deadline **Today 1st March 4pm**
  - Cut-off date is **15th March 4pm**

- Worksheet 5 (not assessed) available today from 11am
  - Virtual Contact Hours will involve meeting discussing Worksheet 5
  - Ask for help with Worksheet 5 in Classmates
  - Can ask for help **this week** with Sudoku assignment, but make it clear

- Primes assignment
  - Worksheet made available next week at 4pm
  - Only involves programming tasks and submission of single js file
  - Deadline **15th March 4pm**
  - Cut-off date **29th March 4pm**

# This was pretty mathematical

1) What made the most sense to you
2) What made the least sense
3) When do constants matter?
4) Can you think of a “Big O” class not mentioned yet?

Let’s return to where we started now we have these mathematical tools

```
function factorial(n) {
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    return a;
}
```

How many operations are required to implement in RAM model?

$N_{op}$ = number of operations

We can now say $N_{op}$ is in the Big O class $O(n)$ for input $n$

```
function factorial(n) {
    var a = 1;
    while (n > 1) {
        a = a * n;
        n--;
    }
    return a;
}
```

How many operations are required to implement in RAM model?

$N_{op}$ = number of operations

We can now say $N_{op}$ is in the Big O class $O(n)$ for input $n$

This is the **Time Complexity** of the algorithm being implemented
- The *smallest* Big O class in which $N_{op}$ lives

```
function sum(n) {
    if (n===0) {
        return 0;
    }

    var a = 0;

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

How many operations (in “Big O” notation) in *n* are required in a RAM implementation?

```
function sumOfFactorials(n) {
    if (n===0) {
        return 1;
    }
    var a = 1;
    for (var i = 1; i <= n; i++) {
        var b = 1;
        for (var j = 1; j <= i; j++) {
            b = b * j;
        }
        a = a + b;
    }
    return a;
}
```

How many operations (in “Big O” notation) in *n* are required in a RAM implementation?

```javascript
function sumOfTwos(n) {
    if (n===0) {
        return 1;
    }
    var a = 0;
    for (var i = 1; i <= n; i++) {
        var b = 0;
        for (var j = i; j <= i + 1; j++) {
            b = b + j;
        }
        a = a + b;
    }
    return a;
}
```

How many operations (in “Big O” notation) in *n* are required in a RAM implementation?

Identifying the number of iterations for an arbitrary input will help us calculate the Time Complexity

# Today

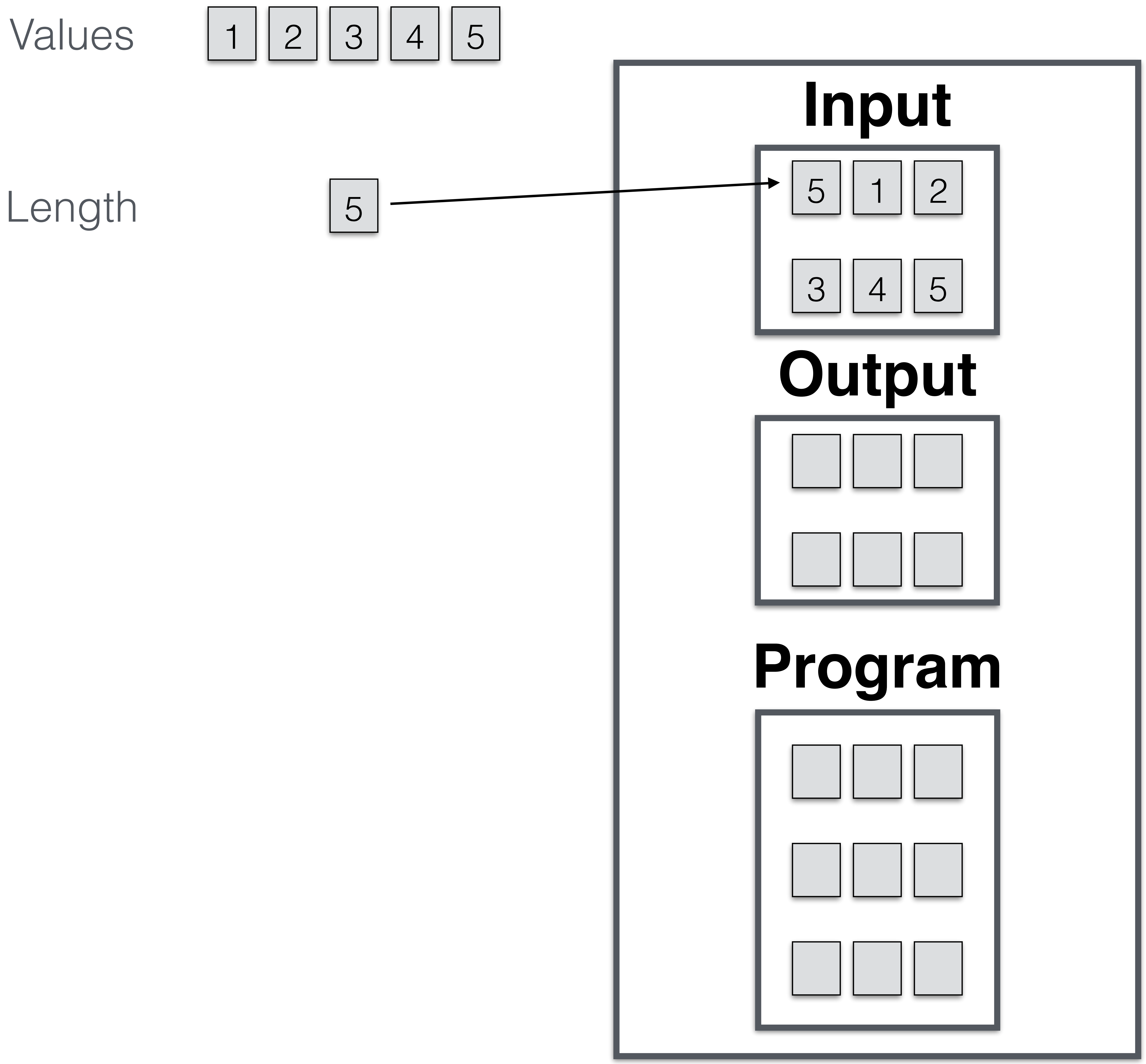
*Analysing Algorithms*

1. Review of RAM model

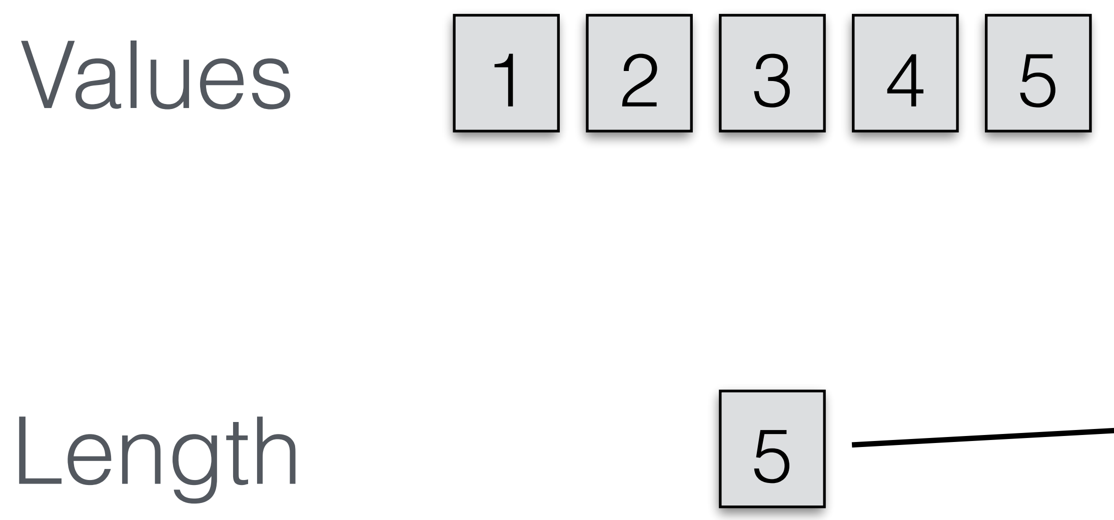
2. Growth of functions and Big O notation

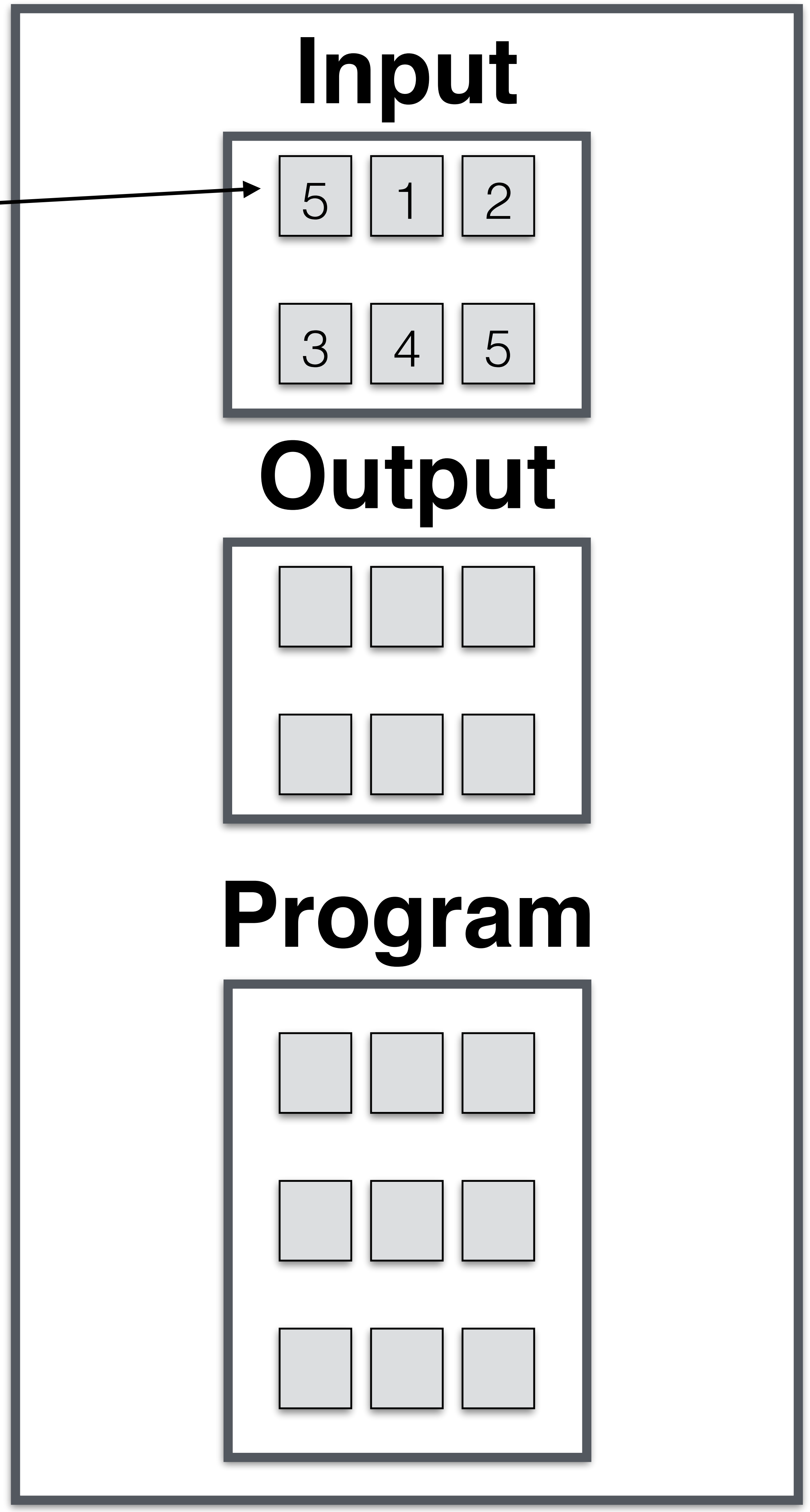
**3. Worst-case analysis**

# Store and use vector in the RAM model



# Store and use vector in the RAM model



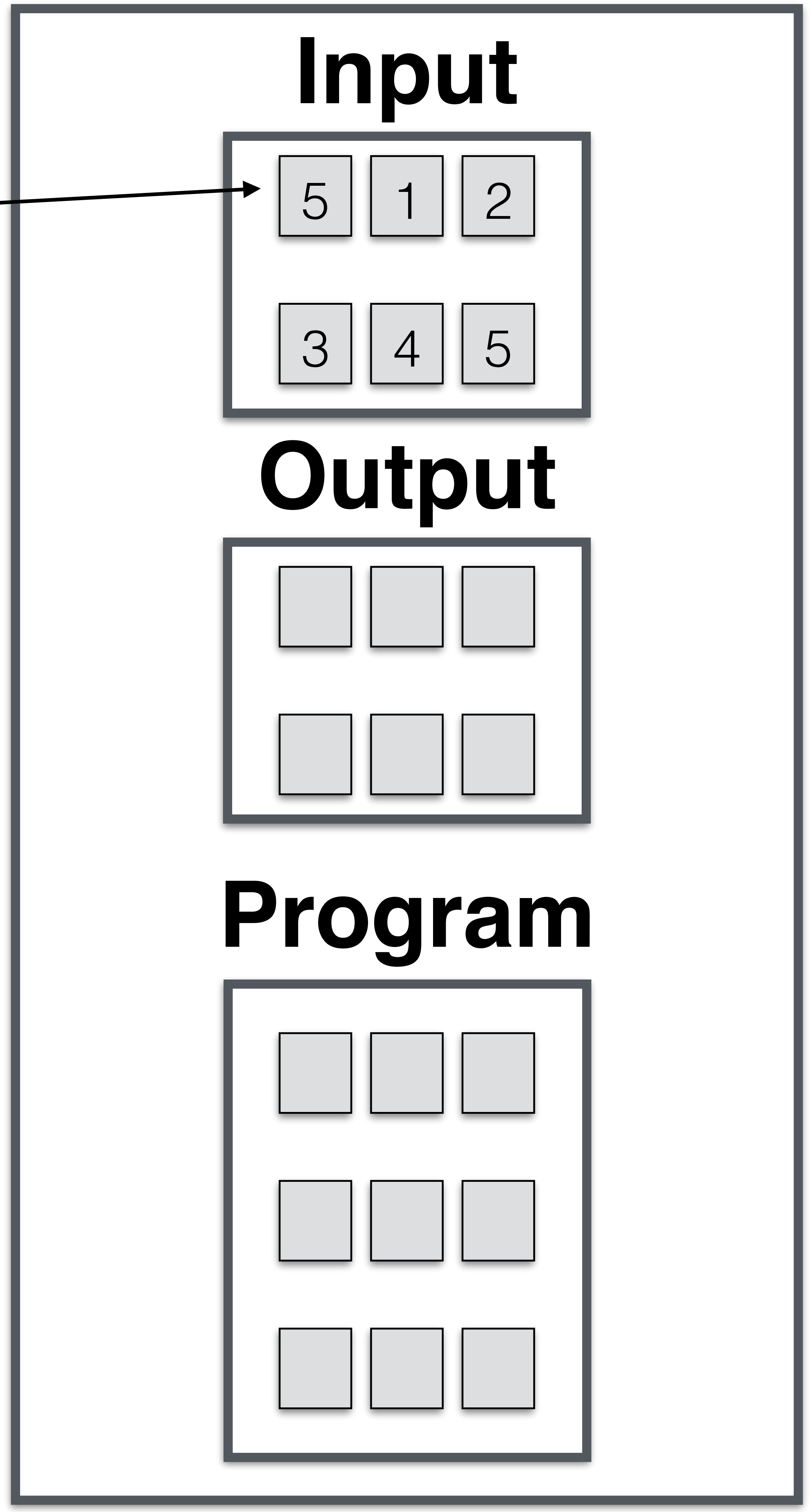

## length

Store the length in the first element of input
Takes one time-step to "look up" the length

# Store and use vector in the RAM model

Values 1 2 3 4 5

Length 5




## length

Store the length in the first element of input
Takes one time-step to “look up” the length

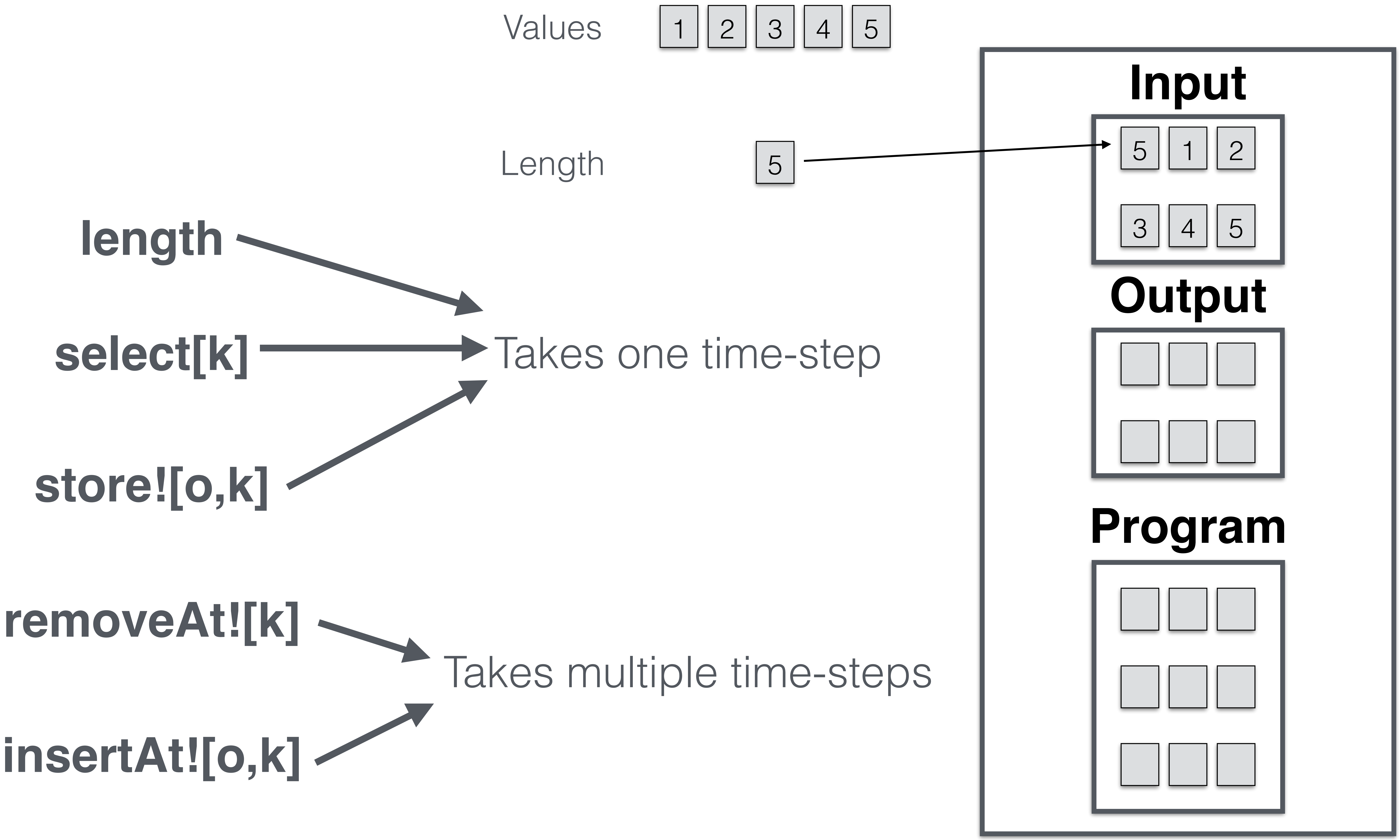
## select[k]

Takes one time-step to read a value in the vector

## store![o,k]

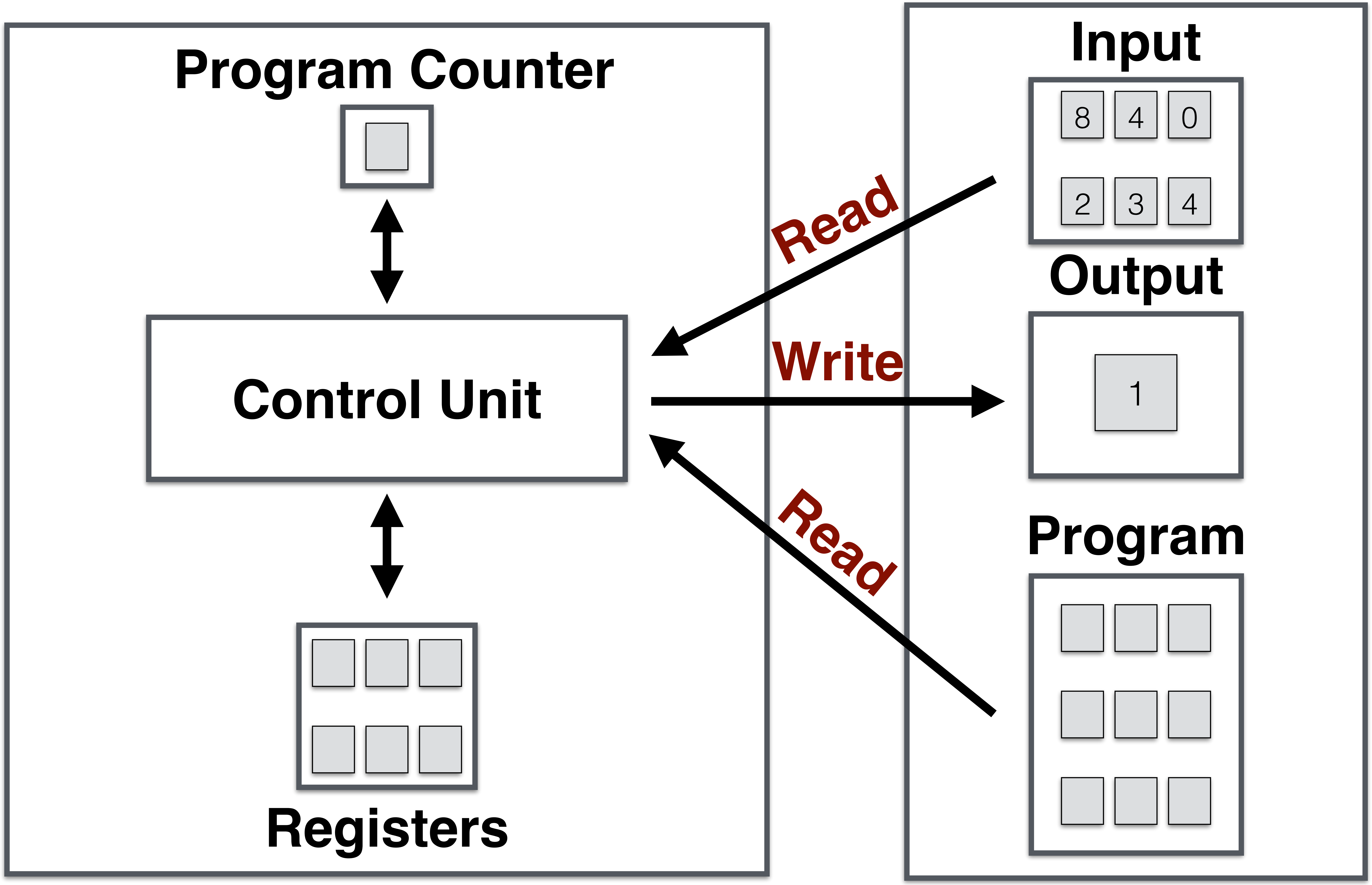
Takes one time-step to store a new value in the vector

# Store and use dynamic array in the RAM model



# Case study: Searching a vector or dynamic array

Is 8 in the following vector? 0 2 3 4



Remember that JavaScript arrays implement dynamic arrays

```
function linearSearch(array,x){

    var n = array.length;

    for (var i = 0; i < n; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;

}
```

Can read the length in one time-step

How many operations (in “Big O” notation) are required in a RAM implementation?

```
function linearSearch(array,x){

    var n = array.length;

    for (var i = 0; i < n; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;

}
```

How many operations (in "Big O" notation) are required in a RAM implementation?

*Depends on the inputs*

We need to consider the **worst-case input** of a fixed size

*The input that will require the most time-steps in a RAM implementation of all inputs of that size*

```
function linearSearch(array,x){

    var n = array.length;

    for (var i = 0; i < n; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;

}
```

Input where *x* is not in the array

*Requires n iterations for length n*
*O(n) time-steps at most*

The variable of interest is the **number of elements** of the array

*Not the numbers **in** the elements per se*

## Worst-Case Time Complexity

The maximum number of operations, or time-steps in “Big O” notation in variable $n$

$n$ could be: number or length of array

## Worst-Case Time Complexity

The maximum number of operations, or time-steps in “Big O” notation in variable $n$

$n$ could be: number or length of array

Reminder: It is the **smallest** Big O class in which the maximum number of operations lives

**Worst-Case Time Complexity**

$O(n)$ for $n$ elements

Sorting algorithms

Important variable: length of vector/dynamic array

# Bubble Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function bubbleSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++){
        var count = 0;
        for (var j = 0; j < n-1; j ++) {
            if (array[j+1] < array[j]) {
                count++;
                swap(array,j,j+1);
            }
        }
        console.log(array);

        if (count == 0) {
            break;
        }
    }

return array;

}
```

# Bubble Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function bubbleSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++){
        var count = 0;
        for (var j = 0; j < n-1; j ++) {
            if (array[j+1] < array[j]) {
                count++;
                swap(array,j,j+1);
            }
        }
        console.log(array);

        if (count == 0) {
            break;
        }
    }

return array;

}
```

How many time-steps in swap?

# Bubble Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function bubbleSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++){
        var count = 0;
        for (var j = 0; j < n-1; j ++) {
            if (array[j+1] < array[j]) {
                count++;
                swap(array,j,j+1);
            }
        }
        console.log(array);

        if (count == 0) {
            break;
        }
    }

return array;

}
```

Best case:

Array already sorted
-only one pass
*O(n)* time-steps

Worst case?

# Bubble Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function bubbleSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++){
        var count = 0;
        for (var j = 0; j < n-1; j ++) {
            if (array[j+1] < array[j]) {
                count++;
                swap(array,j,j+1);
            }
        }
        console.log(array);

        if (count == 0) {
            break;
        }
    }

return array;

}
```

Best case:

Array already sorted - only one pass
*O(n)* time-steps

Worst case:

Array sorted in reverse -
Need (n-1) passes
*O($n^2$)* time-steps

**Worst-Case Time Complexity**

$O(n^2)$ for $n$ elements

# Insertion Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function insertionSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++) {
        var j = i;
        while ((j > 0) && (array[j-1]>array[j])) {
            swap(array,j,j-1);
            j--;
        }
        console.log(array);
    }

    return array;

}
```

# Insertion Sort

```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function insertionSort(array) {

// this should return a sorted array
    var n = array.length;

    for (var i = 1; i < n; i++) {
        var j = i;
        while ((j > 0) && (array[j-1]>array[j])) {
            swap(array,j,j-1);
            j--;
        }
        console.log(array);
    }

    return array;

}
```

1) What is the best-case input array?

2) What is the worst-case input array?

3) What is the worst-case time complexity of Insertion Sort?

# Insertion Sort

**Worst-Case Time Complexity**

$O(n^2)$ for $n$ elements

# Summary

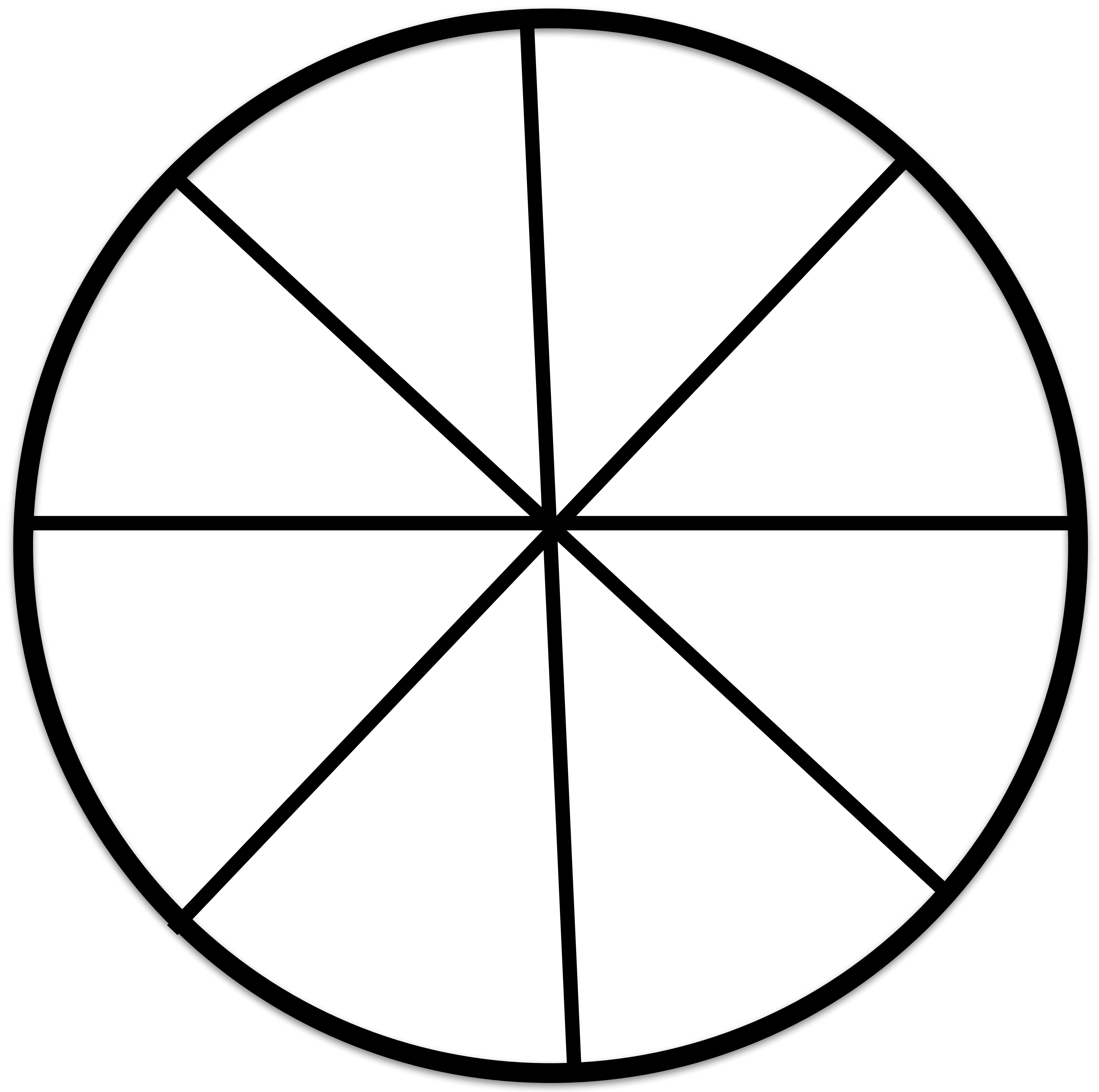
RAM model: abstract model for computers

**Time complexity**: number of operations required as “Big O” class for simple input

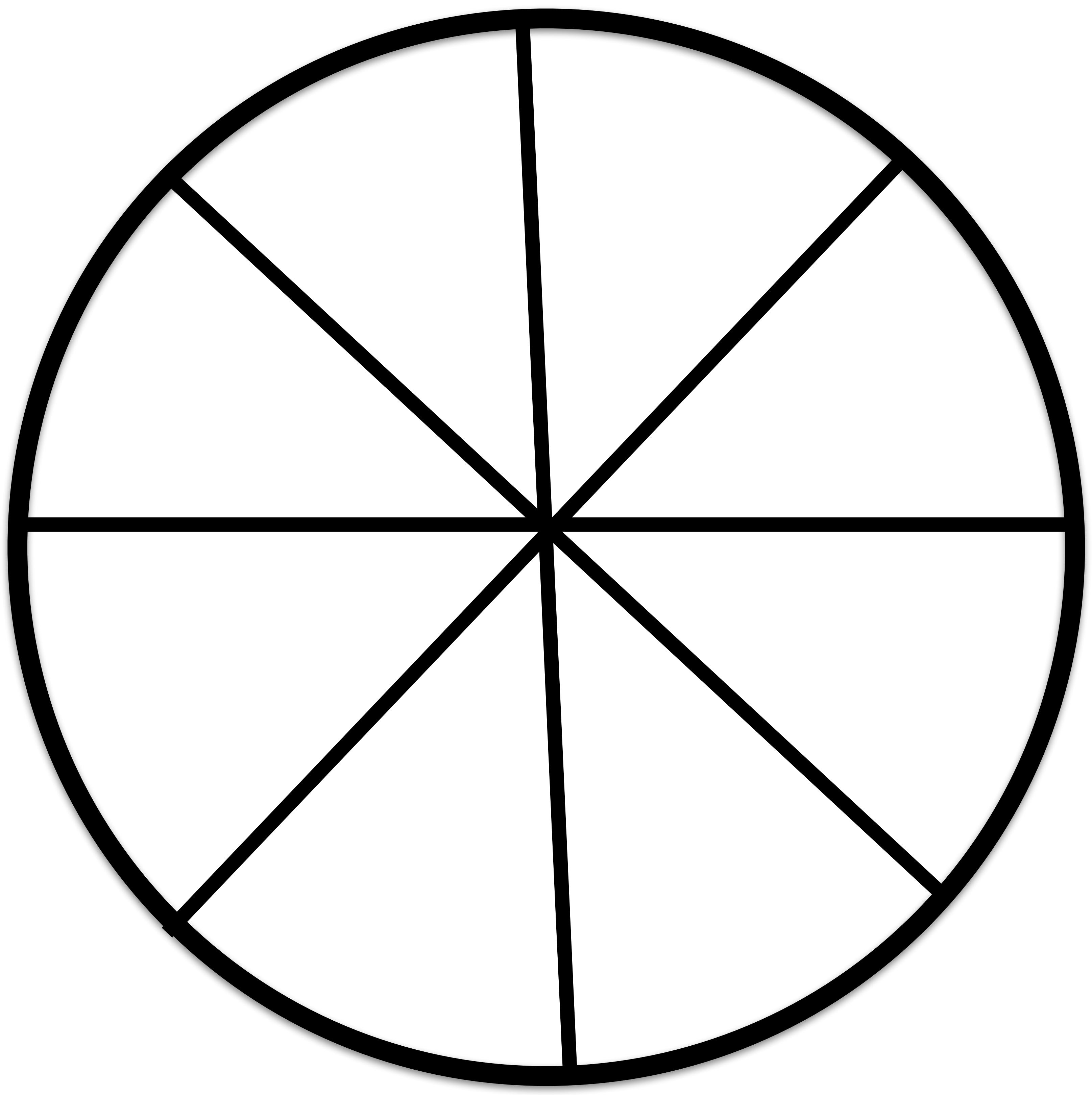
**Worst-case time complexity**: number of operations required as “Big O” class for the worst-case input

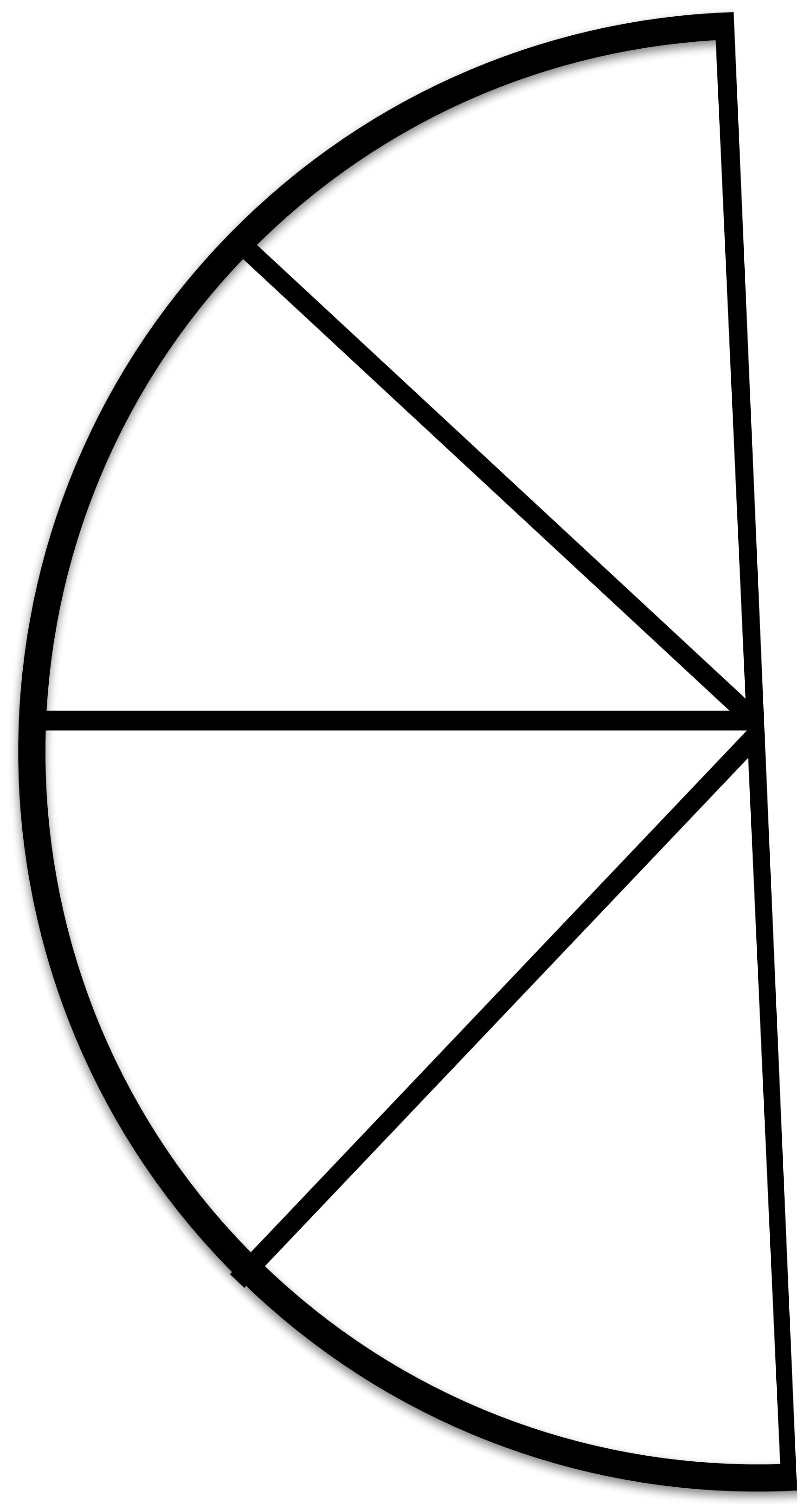
# Problem 5

# Problem 5

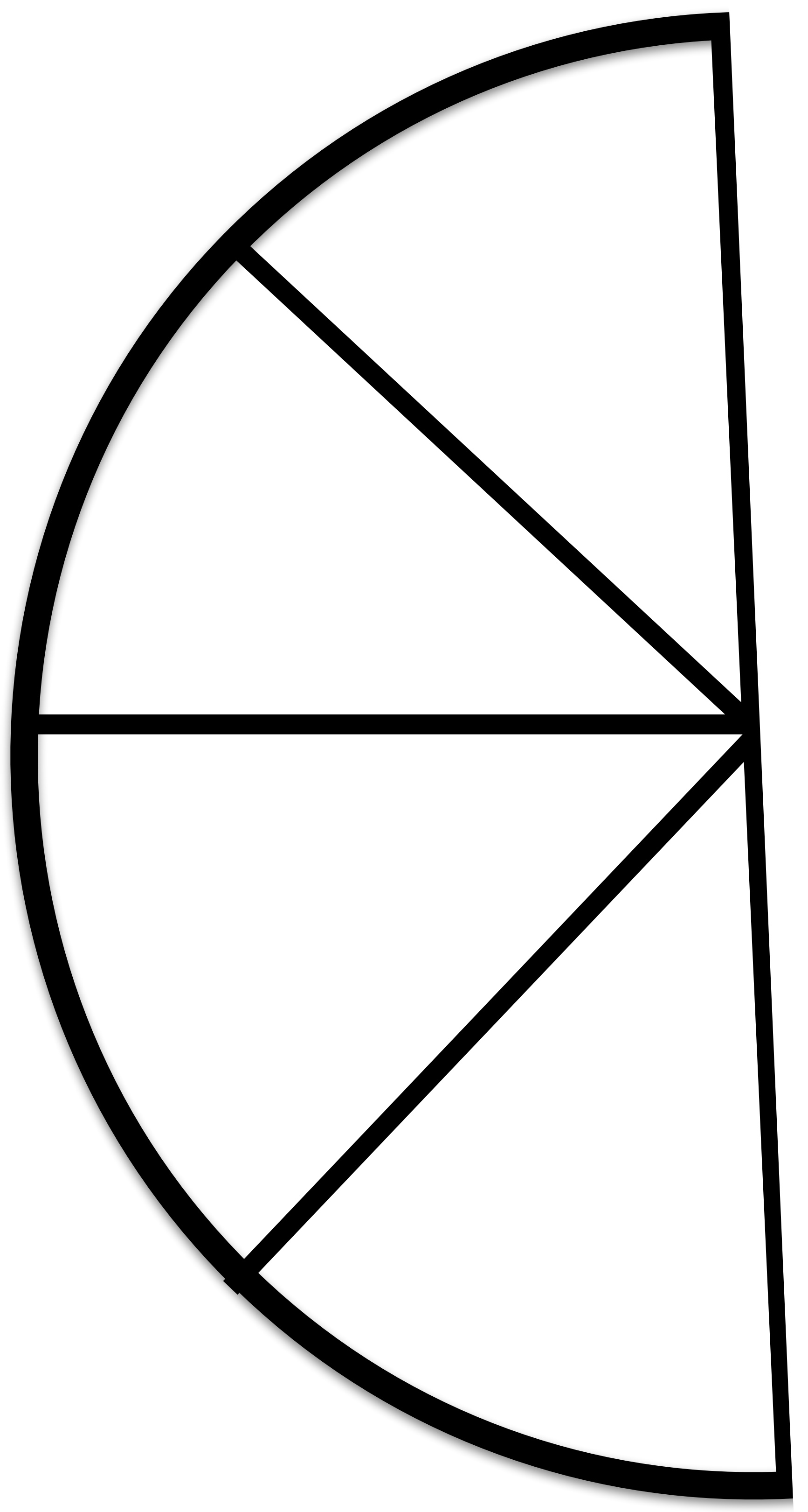


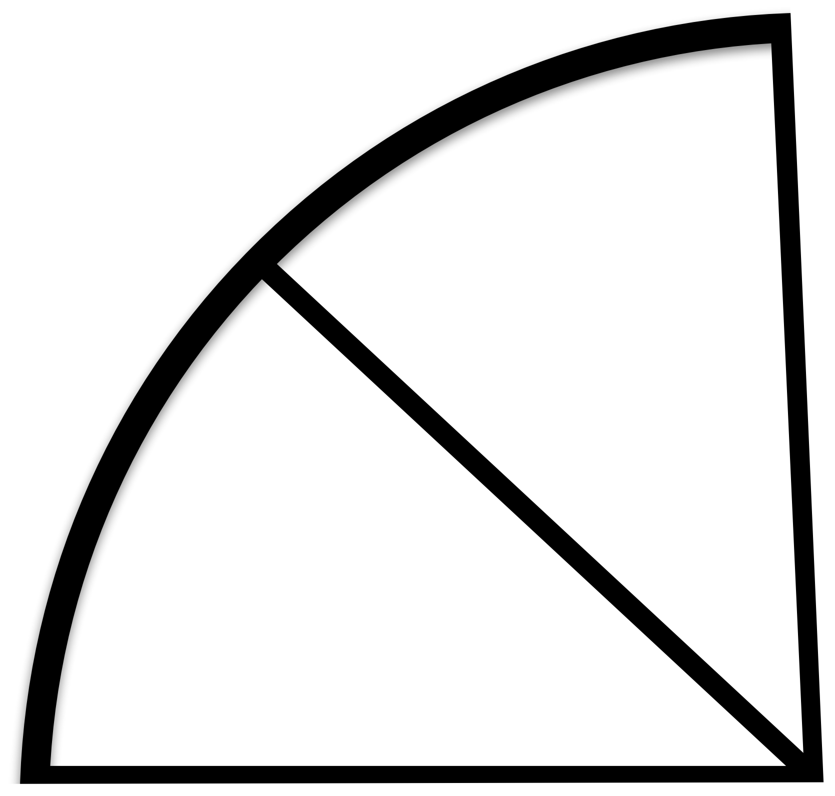
*Group of friends turn up and want half*



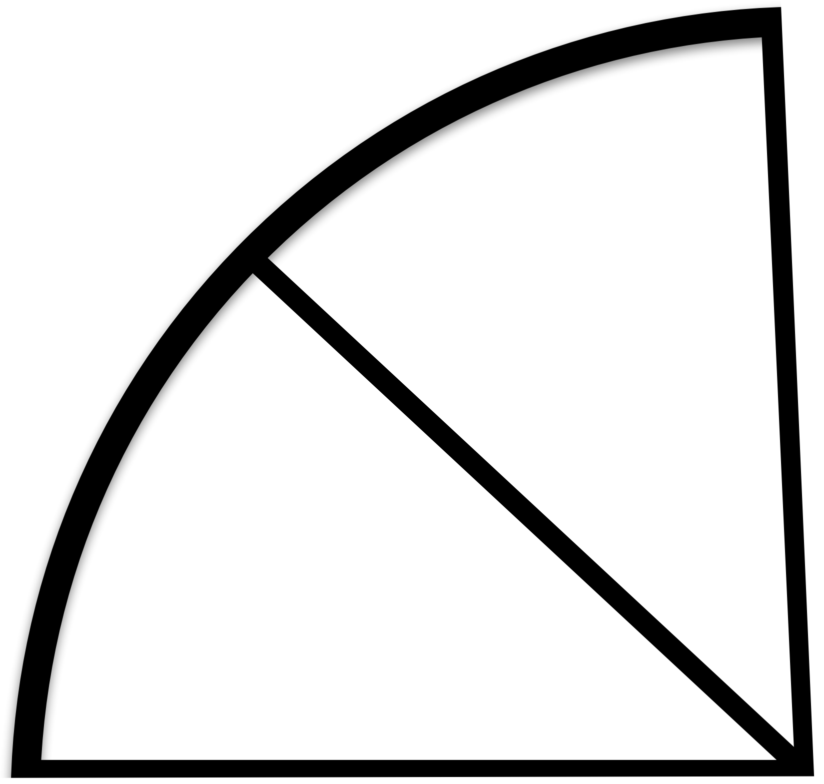


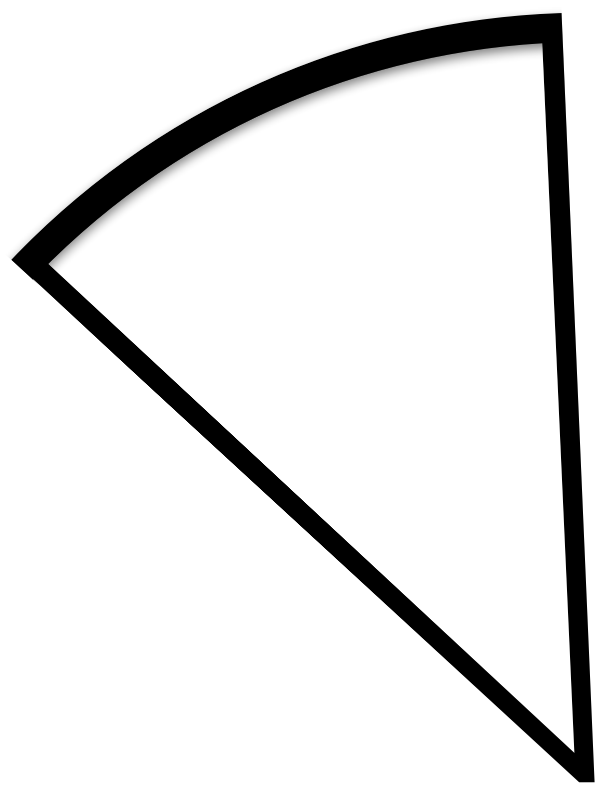
***Another group*** *of friends turn up and want half*



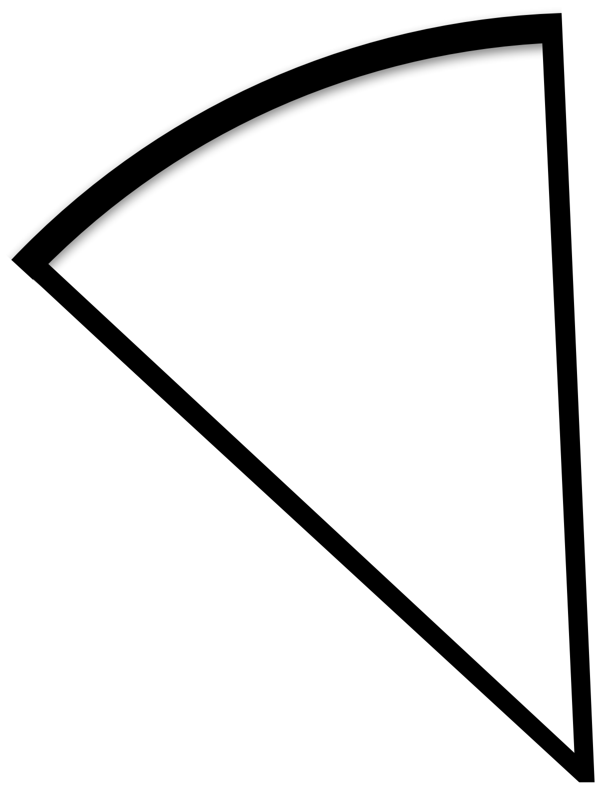


***Another group*** *of friends turn up and want half*





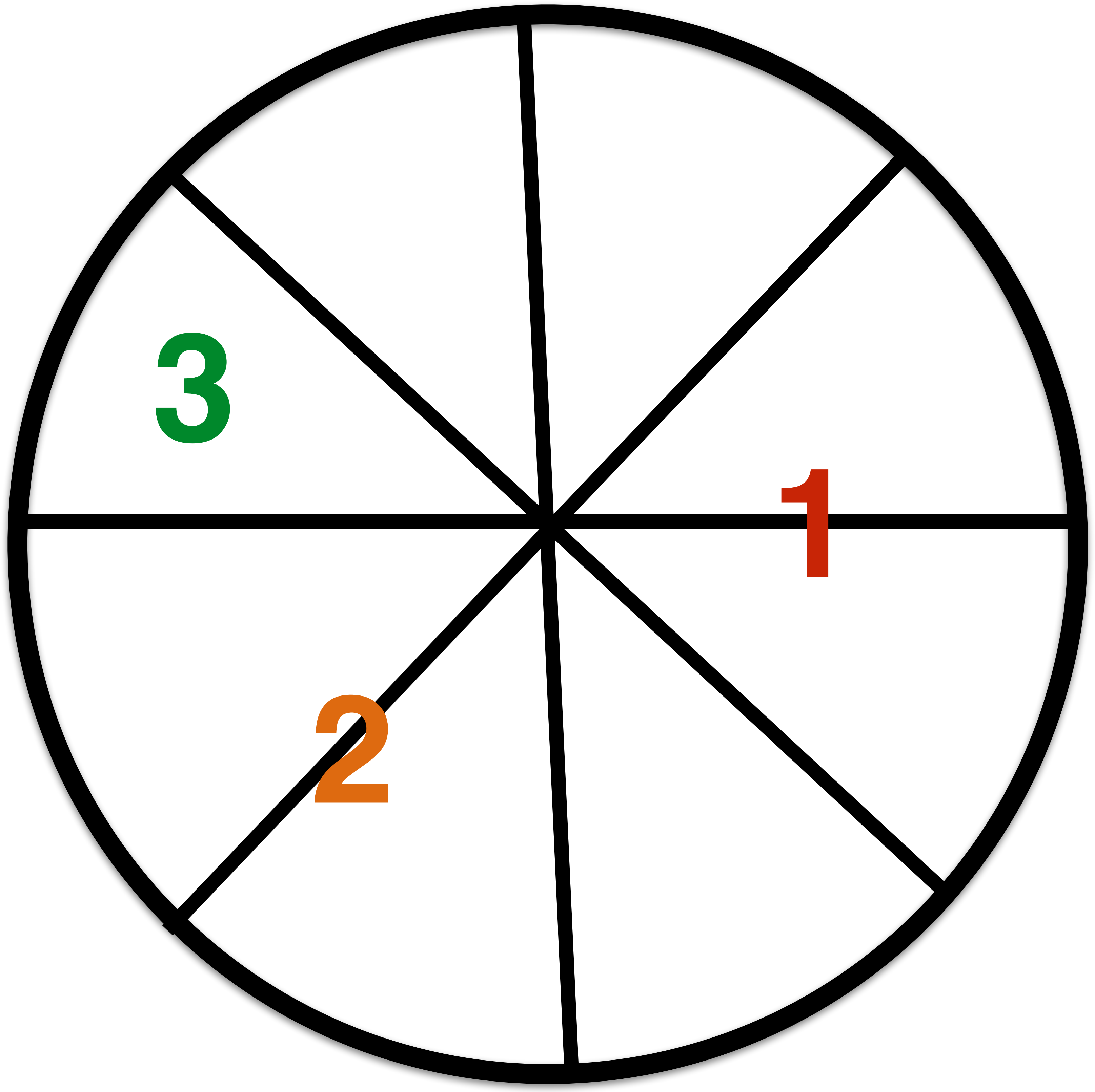
*This last slice is for you*



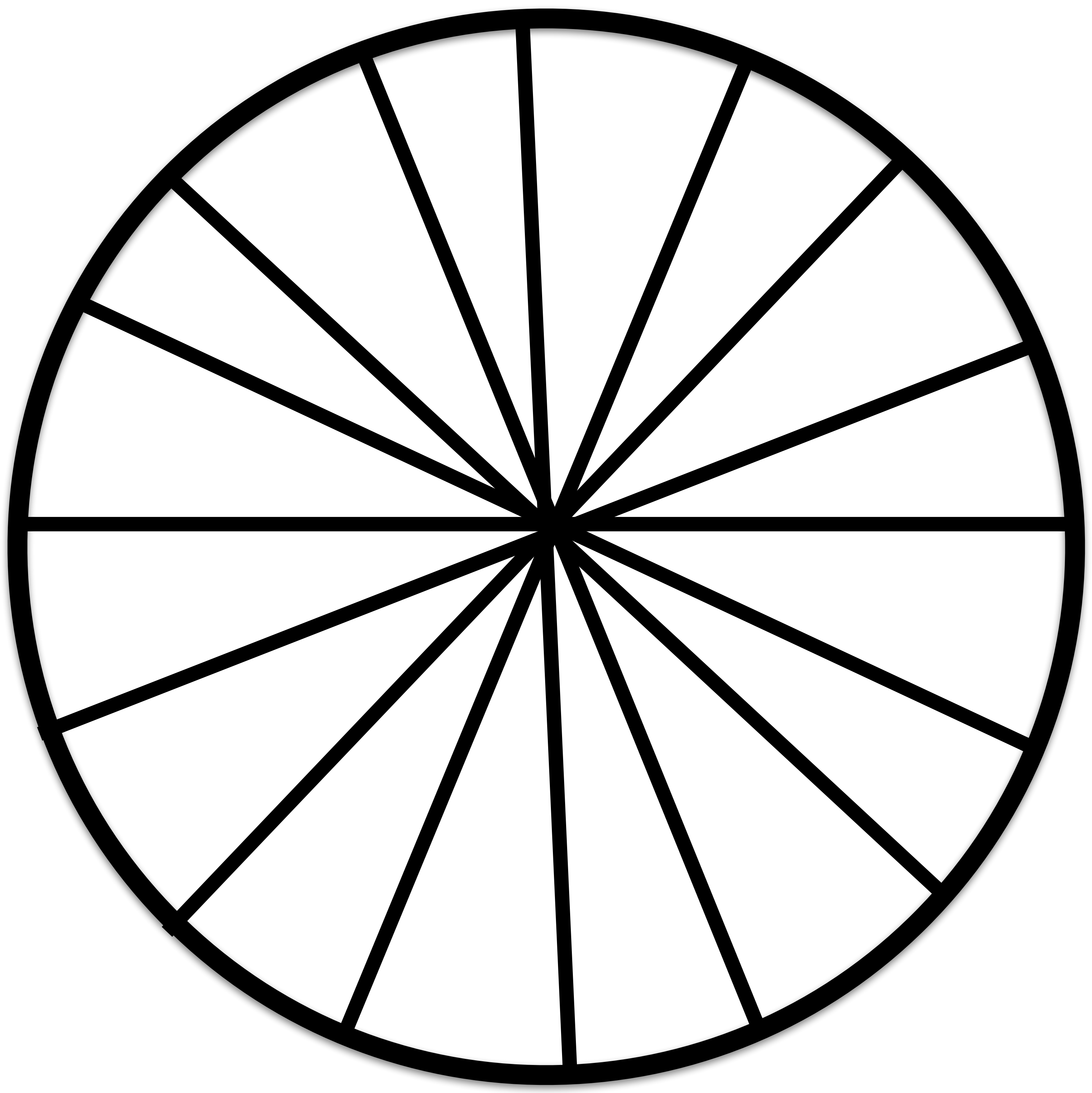
***This last slice is for you***

For **8** slices we could accommodate **3** groups of friends

$$8 * (1/2) * (1/2) * (1/2) = 1$$

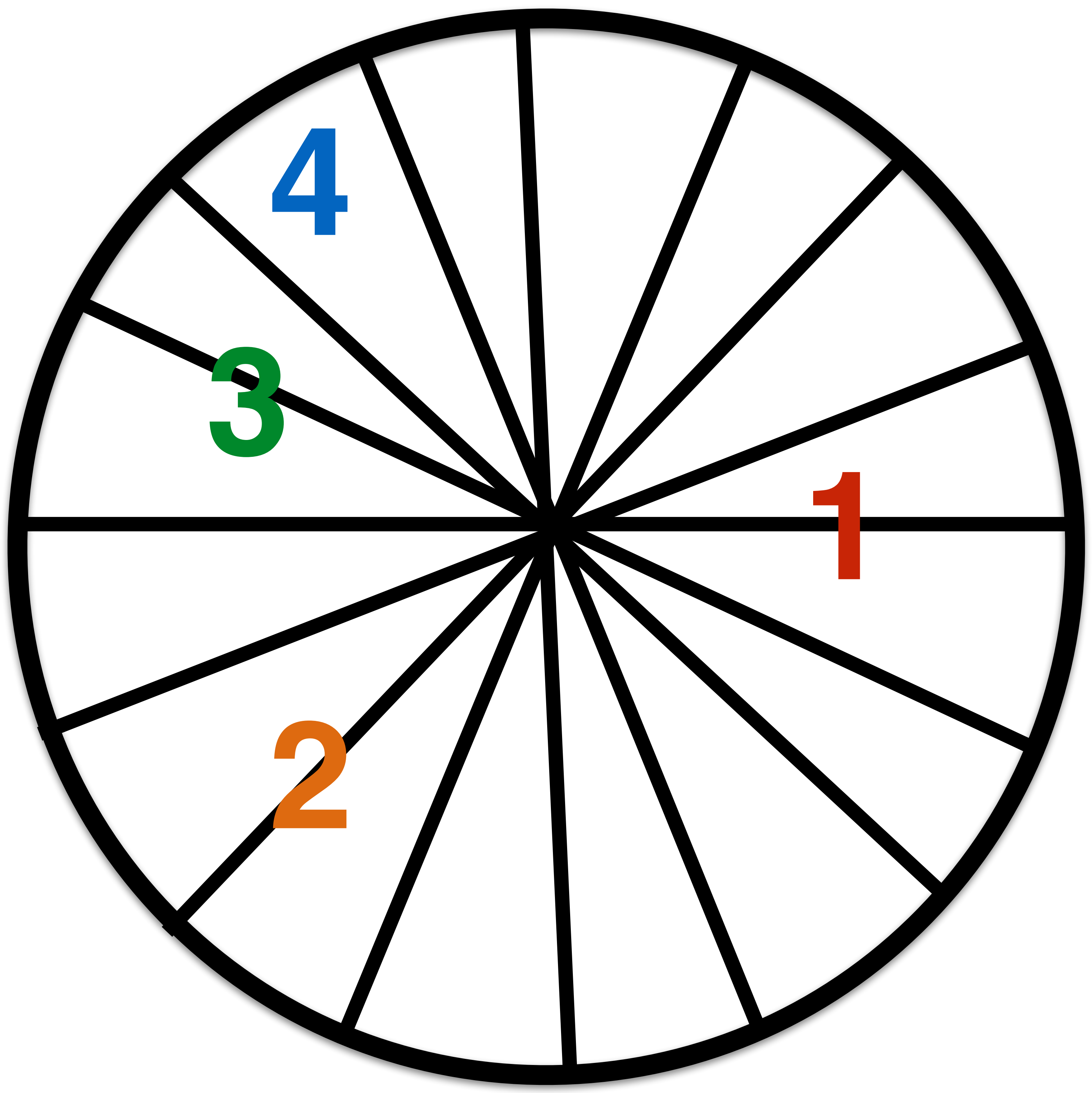



For **8** slices we could accommodate **3** groups of friends



For **16** slices how many groups of friends?

$$8 * (1/2) * (1/2) * (1/2) * (1/2) = 1$$




For **16** slices how many groups of friends?

**4**

For **n** slices how many groups of friends?

*k = number of groups*

For **n** slices how many groups of friends if they ask for two-thirds each time?

*k = number of groups*