Time Complexity Analysis 101 and Data Filtering

COMP110 - Lecture 9 - Spring 2018

1. PollEv: True/False - Given the same inputs, these two functions return the same results.

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

2. PollEv: When each function is called as follows, how many times does each function recursively call itself before it completes?

`max1(cons(3, null))` vs. **`max2(cons(3, null))`**

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

3. PollEv: When each function is called as follows, how many times does each function recursively call itself before it completes?

`max1(cons(2, cons(3, null)))` vs.
`max2(cons(2, cons(3, null)))`

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

Algorithm Analysis 101

- "How many steps does an algorithm take to complete..." is an important question in computer science.
  - In fact, it's an entire research subfield: Algorithm Analysis

- We will *briefly* introduce this concept using the previous examples of functions *max1* and *max2*.

- *Time complexity* is the term we use to describe how many steps an algorithm takes to complete.

- Two different algorithms which solve the same problem can have ***dramatically*** different *time complexities*.

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

`max1(null)`

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

`max1(null)`

`0`

Recursive calls were made to compute `max1(null)`

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
max1(cons(3, null))
```

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

`max1(cons(3, null))` → `max1(null)`

1

Recursive call was made to compute `max1(cons(3, null))`

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
max1(cons(2, cons(3, null)))
```

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

`max1(cons(2, cons(3, null)))` → `max1(cons(3, null))`

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

`max1(cons(2, cons(3, null)))` → `max1(cons(3, null))` → `max1(null)`

2

Recursive calls were made to compute `max1(cons(2, cons(3, null)))`

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

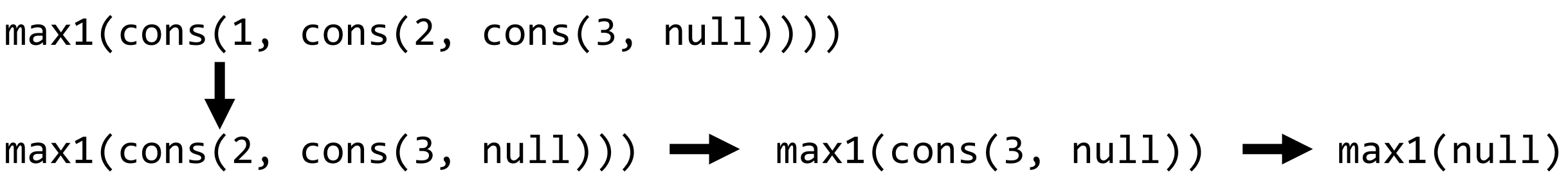
```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```

```
max1(cons(1, cons(2, cons(3, null))))
```

Let's analyze **max1**

Given some input, how many recursive calls does max1 take to complete?

```
let max1 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    let front: number = first(values);
    let maxRest: number = max1(rest(values));
    if (front > maxRest) {
      return front;
    } else {
      return maxRest;
    }
  }
};
```



3

Recursive calls were made to compute `max1(cons(1, cons(2, cons(3, null))))`

Time Complexity Analysis of **max1**

Input	Length of Input (n)	Time Complexity
`null`	0	0
`cons(3, null)`	1	1
`cons(2, cons(3, null))`	2	2
`cons(1, cons(2, cons(3, null)))`	3	3

Notice as the length of the input grows, the time complexity, or number of steps the computer will take, to solve **`max1`** grows ***linearly*** with it.

Generally, **`max1`** takes **n** steps to complete when given a List of length **n**.

IT'S PEANUT BUTTER JELLY TIME!!!

Let's analyze **max2**

Given some input, how many recursive calls does **max2** take to complete?

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

`max2(null)`

`0`

Recursive calls were made to compute `max2(null)`

Let's analyze **max2**

Given some input, how many recursive calls does **max2** take to complete?

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

`max2(cons(3, null))` → `max2(null)`

1

Recursive call was made to compute `max2(cons(3, null))`

Let's analyze **max2**

Given some input, how many recursive calls does **max2** take to complete?

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

1

`max2(cons(2, cons(3, null)))`

`max2(cons(3, null))`  `max2(null)`

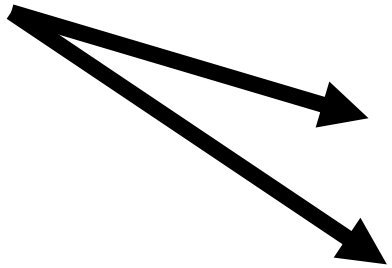
Let's analyze **max2**

Given some input, how many recursive calls does **max2** take to complete?

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

2

```
max2(cons(2, cons(3, null)))
```



```
max2(cons(3, null))  ⟶  max2(null)
max2(cons(3, null))  ⟶  max2(null)
```

4

Recursive calls were made to compute `max2(cons(2, cons(3, null)))`

Let's analyze **max2**

Given some input, how many recursive calls does **max2** take to complete?

```
let max2 = (values: List<number>): number => {
  if (values === null) {
    return Number.MIN_SAFE_INTEGER;
  } else {
    if (first(values) > max2(rest(values))) {
      return first(values);
    } else {
      return max2(rest(values));
    }
  }
};
```

1

2

```
max2(cons(2, cons(3, null)))
```

```
max2(cons(3, null))  ⟶  max2(null)
max2(cons(3, null))  ⟶  max2(null)
```

4

Recursive calls were made to compute `max2(cons(2, cons(3, null)))`

Why?

Notice that when the max value of the rest of the list is greater than or equal to the first value, we are calling the max2 function *again* in the return statement.

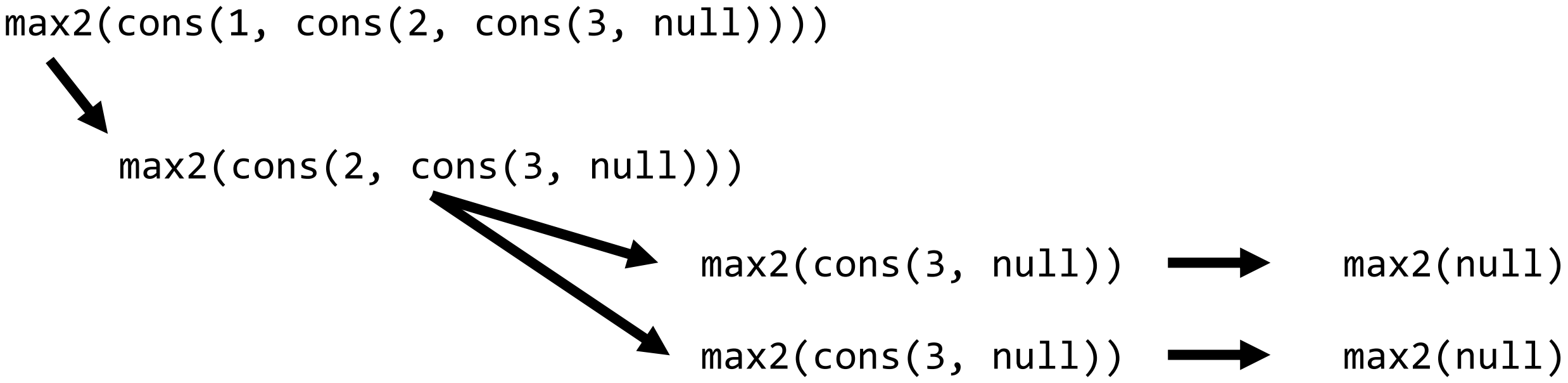
Let's analyze **max2**

To save space, the code for max2 is not shown.
It is the same as previous slides.

```
max2(cons(1, cons(2, cons(3, null))))
```

Let's analyze **max2**

To save space, the code for max2 is not shown.
It is the same as previous slides.

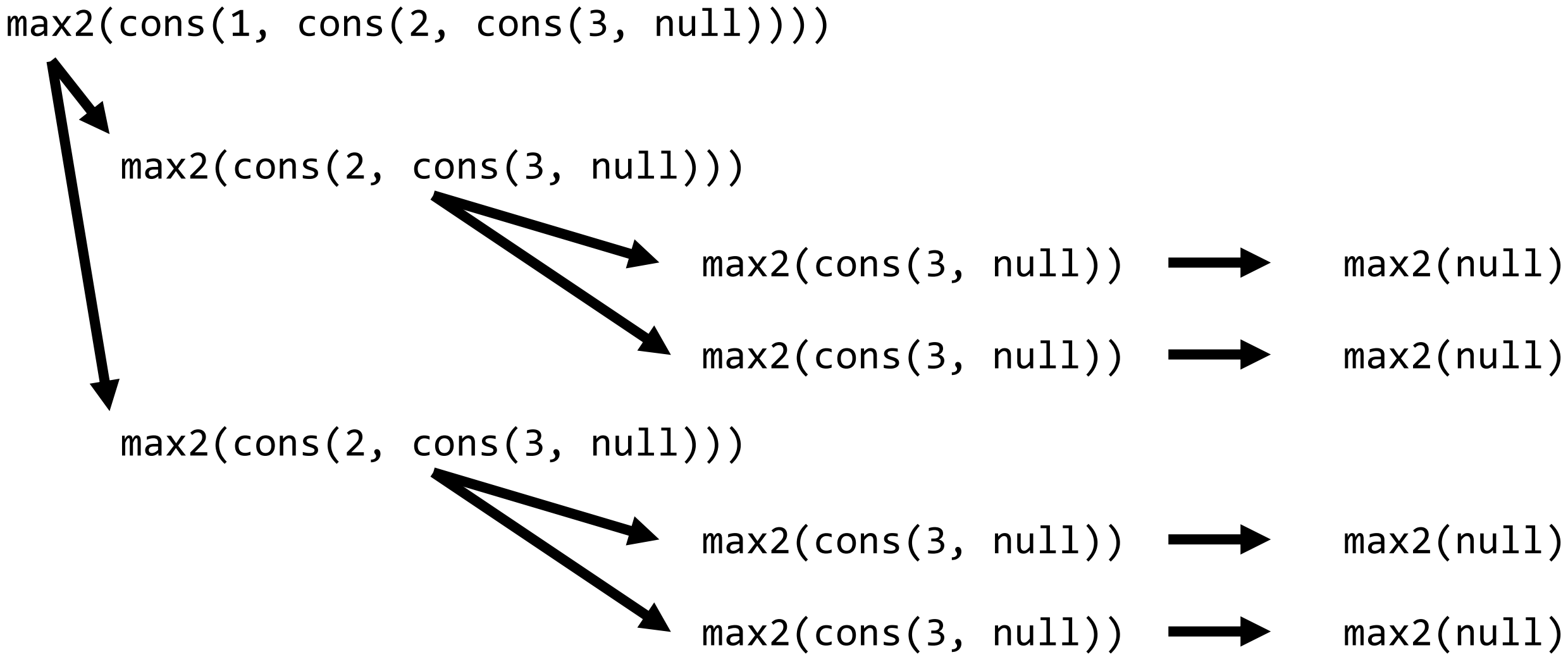


Let's analyze **max2**

To save space, the code for max2 is not shown.
It is the same as previous slides.

10

Recursive calls were made to compute
`max2(cons(1, cons(2, cons(3, null))))`



Time Complexity Analysis of **max2**

Input	Length of Input (n)	Time Complexity
`null`	0	0
`cons(3, null)`	1	1
`cons(2, cons(3, null))`	2	4
`cons(1, cons(2, cons(3, null)))`	3	10
`listify(1, 2, 3, 4)`	4	

Notice as the length of the input grows, the number of recursive steps the computer will take to solve **`max2`** grows *exponentially* with it.

Generally, **`max2`** takes: **`2 + 2 * max2`**$_{\texttt{steps}}$**`(n - 1)`** `to complete`

Linear vs. Exponential Time Complexity

- What's the big deal?
- Imagine we have a list of everyone's grades in COMP110 and we are trying to find the highest grade. Assume 270 students.
- Using **`max1`**, this will take the computer 270 steps. Using **`max2`**, this will take the computer *more steps than there are atoms in the universe*. (Spoiler: It'll never finish.)

Length of List	max1 steps	max2 steps
0	0	0
1	1	1
2	2	4
3	3	10
…	…	…
30	30	1,610,612,734
31	31	3,221,225,470
32	32	6,442,450,942
…	…	…
270	270	1.78×10^{80}

Follow-along: Tinkering with **`max1`** and **`max2`**

- Let's add function calls for max1 and max2 beneath each TODO:

```
print(max1(input));
```

```
print(max2(input));
```

- Now let's try changing the number of values in the input List.

```
listify(1, 2, 3, 4, 5);
```

Rules of Recursive Functions

1. Test for a base case
   - List Parameter: Is the list empty?
   - Number Parameter: Is the number equal to 0?

2. Always change at least one argument when recurring
   - List Parameter: Use the rest of the List when recurring.
   - Number Parameter: Subtract 1 from the number when recurring.

3. To build a list, process the first value, and then cons it onto the result of repeating the same process recursively on the rest of the list.

4. **Never make a recursive function call with the same arguments more than once per invocation of a function. If you need to use recursive result multiple times, store the result in a variable first.**

Back to Data Processing

- Last class: Of **all games** this season, how many **points** has Joel Berry II scored **in total**?

- Today: we'll get more practice working with a list of data objects

Hands-on #1 – Pair Up and Comment

- Open Lec08 / 01-filtering-app.ts
- Pair up with your neighbor and play rock paper scissors.
- The winner should read through and describe what the foo function is doing. Add this description in the TODO COMMENT section above the foo function.
- The not-quite-winner should read through and describe what the bar function is doing. Add this description in the TODO COMMENT section above the bar function.
- Check-in on PollEv.com/compunc when you have added descriptions for each.

Hands-on #2

- Still in 01-filtering-app.ts
- In the main function there are two TODOs.
- Your goal is to replace each of the assigned 0's with function calls which will correctly compute each variable using the **`games`** List as an argument.
  - TODO #2 will need only one function call.
  - TODO #3 will need two function calls (one nested as an argument to other)
  - Hint: refer to last class' last slide and read your comments on foo/bar functions
- Check-in on PollEv.com/compunc when you have 26 games and 455 points printing.

```
// Hands-on #2 TODO: Replace 0 with a correct function call.
let totalGames: number = foo(games);
print("JBII has played in " + totalGames + " games this szn.");

// Hands-on #3 TODO: Replace 0 with correct, nested function
// calls to gamesToPoints and a function that adds up a list
// of numbers.
let totalPoints: number = bar(gamesToPoints(games));
print("JBII has scored " + totalPoints + " points this szn.");
```

Hands-on #3

- How could we find Joel's points in only the games **where** UNC won?

- Find the `filterByWins` function at TODO #4. Here we need to **build a list recursively.** How?
  1. If the **games** list is empty, then return an empty list.
  2. Else
     1. Setup a variable to hold the first game of the games List:
        `let game: Game = first(games);`
     2. If the first game of the games list's **uncPoints** property is greater than its **opponentPoints** property, then return the first game cons'ed onto the natural recursion.
        `return cons(game, filterByWins(rest(games)));`
     3. Else
        - Return the natural recursion: `filterByWins(rest(games))`

- At TODO #5 – Assign to the games variable the result of calling `filterByWins` using the original games list as its argument. Check-in when you have the correct results printing ().

```
let filterByWins = (games: List<Game>): List<Game> => {
    // TODO #4
    if (games === null) {
        return null;
    } else {
        let game: Game = first(games);
        if (game.uncPoints > game.opponentPoints) {
            return cons(game, filterByWins(rest(games)));
        } else {
            return filterByWins(rest(games));
        }
    }
};
```

```
// TODO #5 - assign to games the result of calling filterByWins
games = filterByWins(games);
```

Doesn't this break our recursion rule #4?

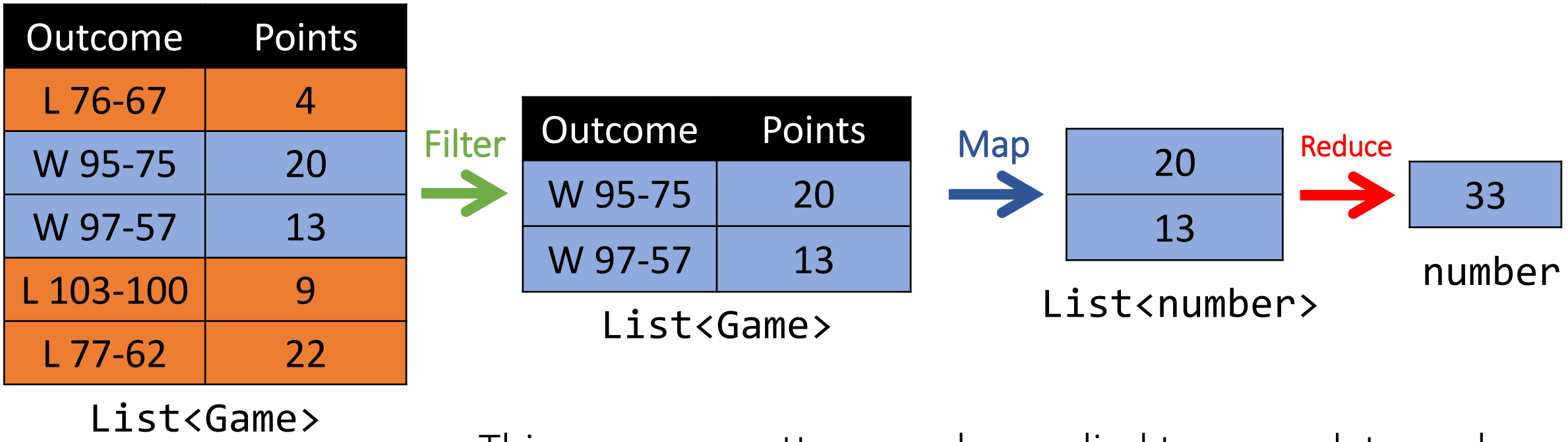
4) Never make a recursive function call with the same arguments more than once per invocation of a function. If you need to use recursive result multiple times, store the result in a variable first.

```
let game: Game = fir (games);
if (game.uncPoints > ame.opponentPoints) {
   return cons(game, filterByWins(rest(games)));
} else {
   return filterByWins(rest(games));
}
```

No! Notice the recursive calls are each in either the then- or else-blocks. Per invocation of the `filterByWins` function, the recursive call will only happen *once* because *either* UNC won or we lost.

Filter-Map-Reduce Pipeline Introduction

Of games that UNC won, how many points did the player score in total?



This common pattern can be applied to many data analyses. We have written, and will continue to write, functions that can be classified as filtering, mapping, or reducing functions.

Filtering Functions

Of games that UNC won, how many points did the player score in total?

Outcome	Points
L 76-67	4
W 95-75	20
W 97-57	13
L 103-100	9
L 77-62	22

`List<Game>`

Filter

Outcome	Points
W 95-75	20
W 97-57	13

`List<Game>`

- A **filter** function sifts out data objects we want to ignore
  - e.g. games that UNC lost
  - Think: the club bouncer algo!
- Conversely, the result of a filtering function consists of only the data we do care about
  - e.g. only the games UNC won
- Notice that the input type and output types of a filter function are the same.
  - This is useful because *many* filters can be applied one after another.
  - e.g. games where UNC won AND where JBII scored 10+ points

Mapping Functions

Of games that UNC won, **how many points** did the player score in total?

Outcome	Points
W 95-75	20
W 97-57	13

`List<Game>`

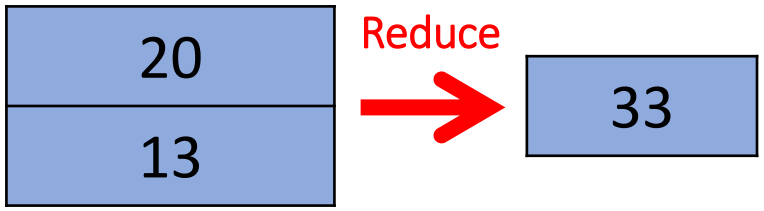
20
13

`List<number>`

- **Mapping** functions transform data from one form to another.
- Often, we have an analysis we want to run (e.g., sum a list of numbers) but our data is in another format (a list of Game objects).
- So we write a function that **maps** from a List of Games to a List of numbers.
  - Notice that the filtering process *does* change the data's type.
  - Notice there's a 1:1 correspondence with input data and output data.

Reducing Functions

Of games that UNC won, how many points did the player score in **total**?



- **Reducing** functions take a collection of input data and *reduce* or *simplify* it in some way.
- A classic example is **`sum`**. A **`sum`** function reduces a list of numbers to a single number.
  - Other usual suspects: count, min, max, average, stddev
- Foreshadow: Reducers *can also* result in a collection of values.
  - Pedantically: a *filtering* function can be thought of as a specific kind of reducing function.