Lecture 13: Sorting Part 2

Kaleel Mahmood
Department of Computer Science and Engineering
University of Connecticut

SORTING A LIST: SELECTION SORT

- Goal
  - Sort a list of value (integers) in increasing order
- Idea
  - Find the minimum,
  - Extract it (remove it from the list),
  - Sort the remaining elements,
  - Add the minimum back in front!

Basic Idea Behind Selection Sort

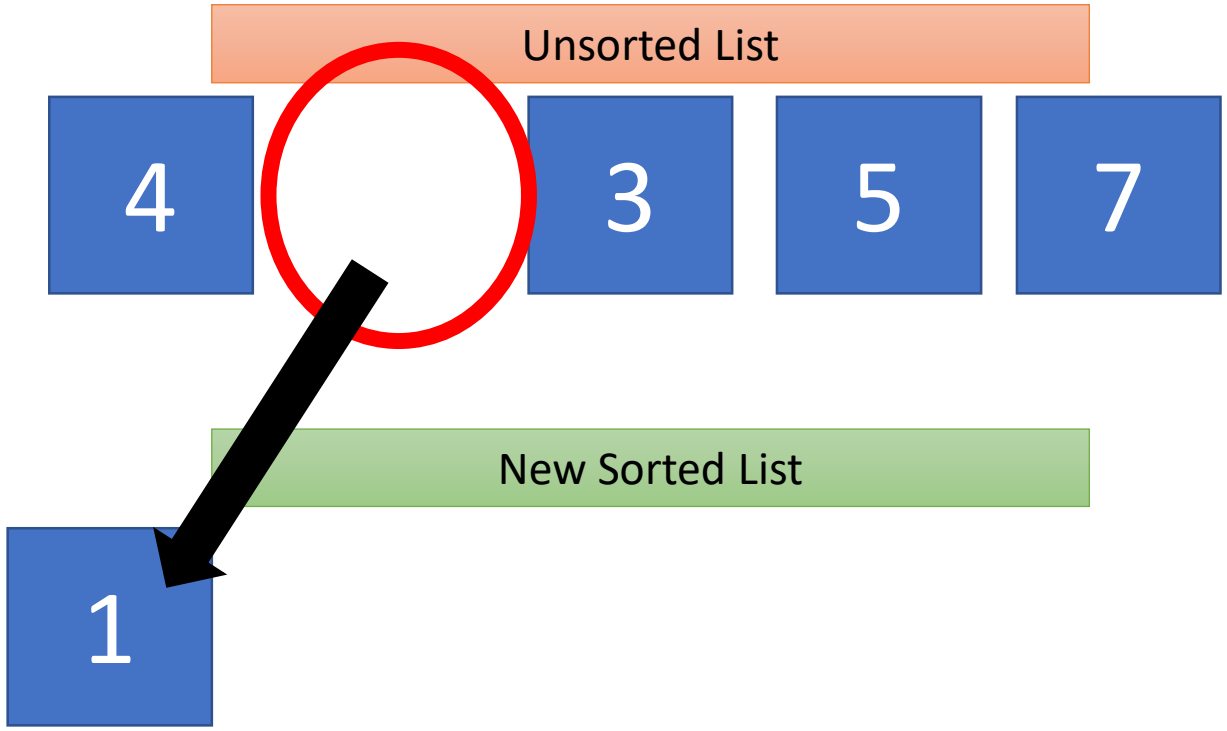
1. Find the minimum element in an unsorted list.
2. Remove that element from the unsorted list.
3. Put that element in a new sorted list.
4. Repeat steps 1-3 on the unsorted list until no elements remain.
5. Return the new sorted list.

Pictorial Idea of Selection Sort

Pictorial Idea of Selection Sort

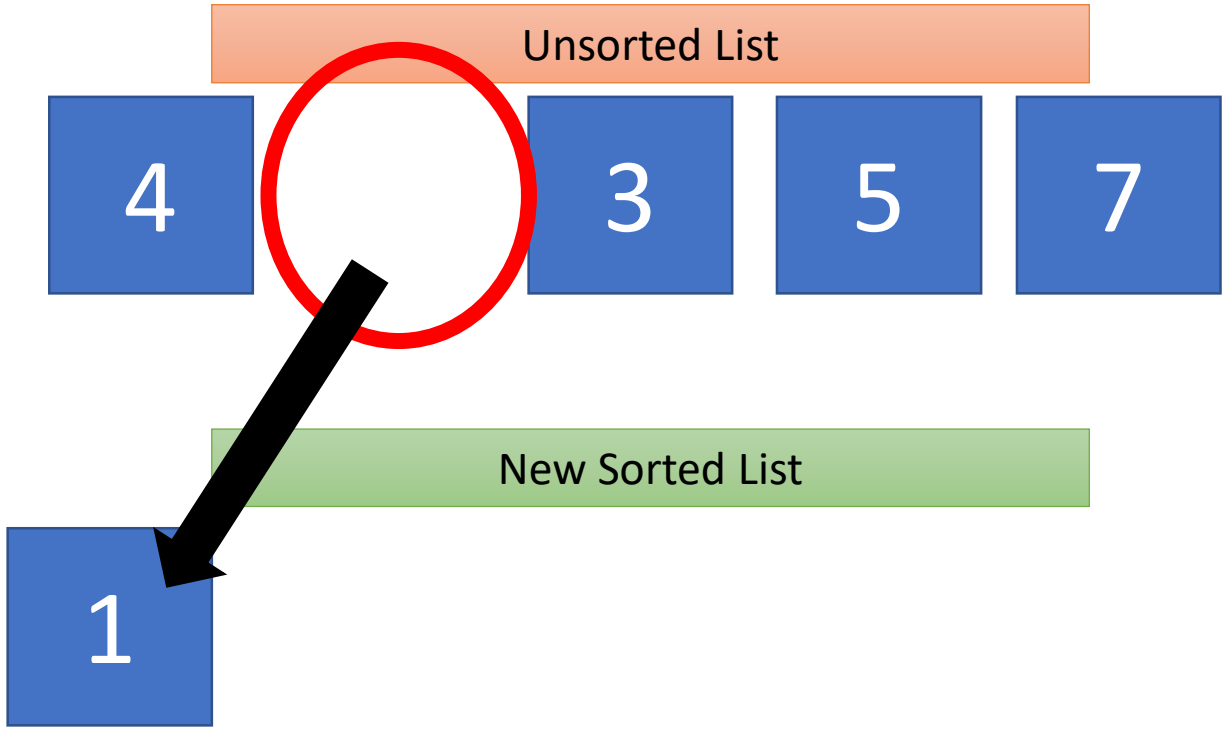
1. Identify Minimum

Pictorial Idea of Selection Sort



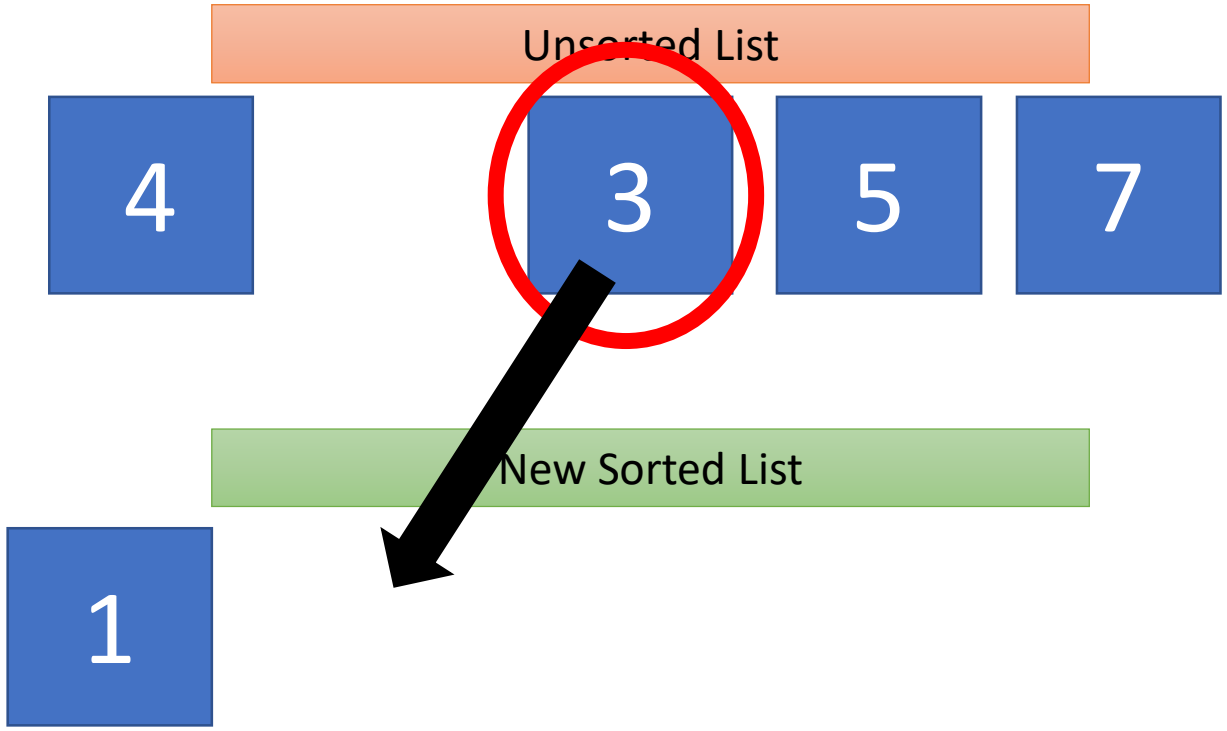
Steps 2 and 3: Remove from unsorted list and place in new sorted list.

Pictorial Idea of Selection Sort



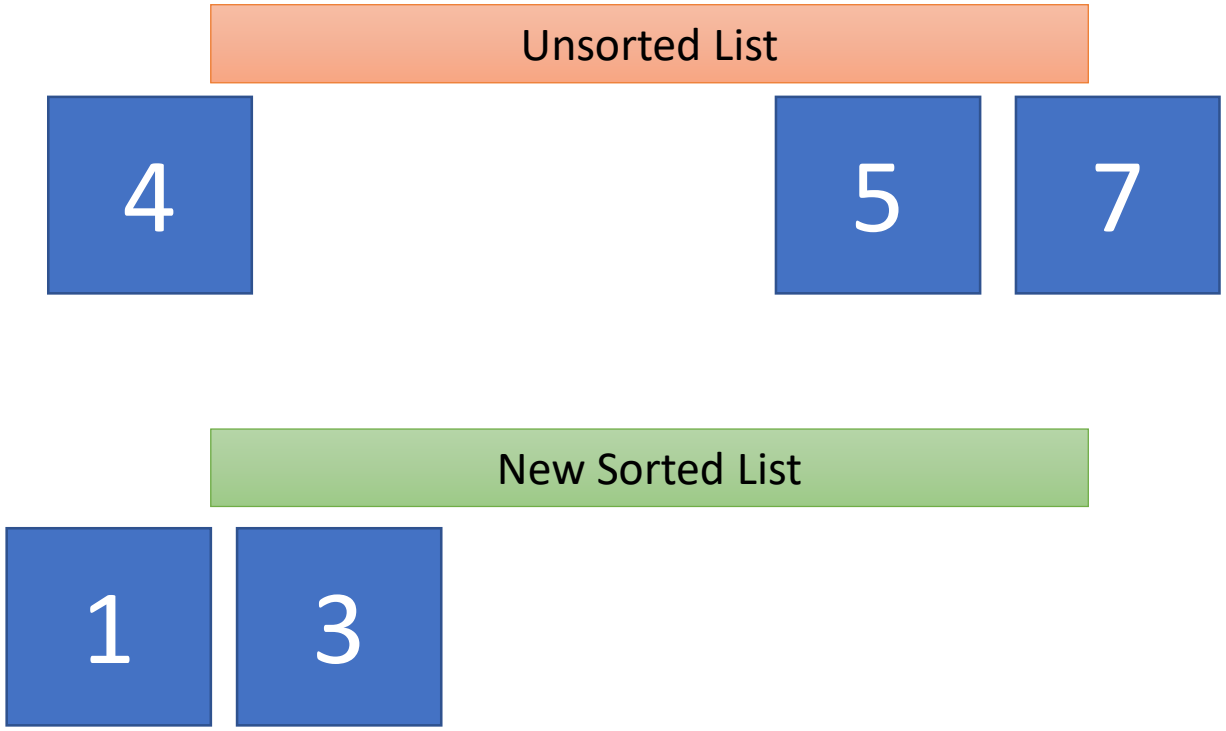
Repeat steps 1-3

Pictorial Idea of Selection Sort



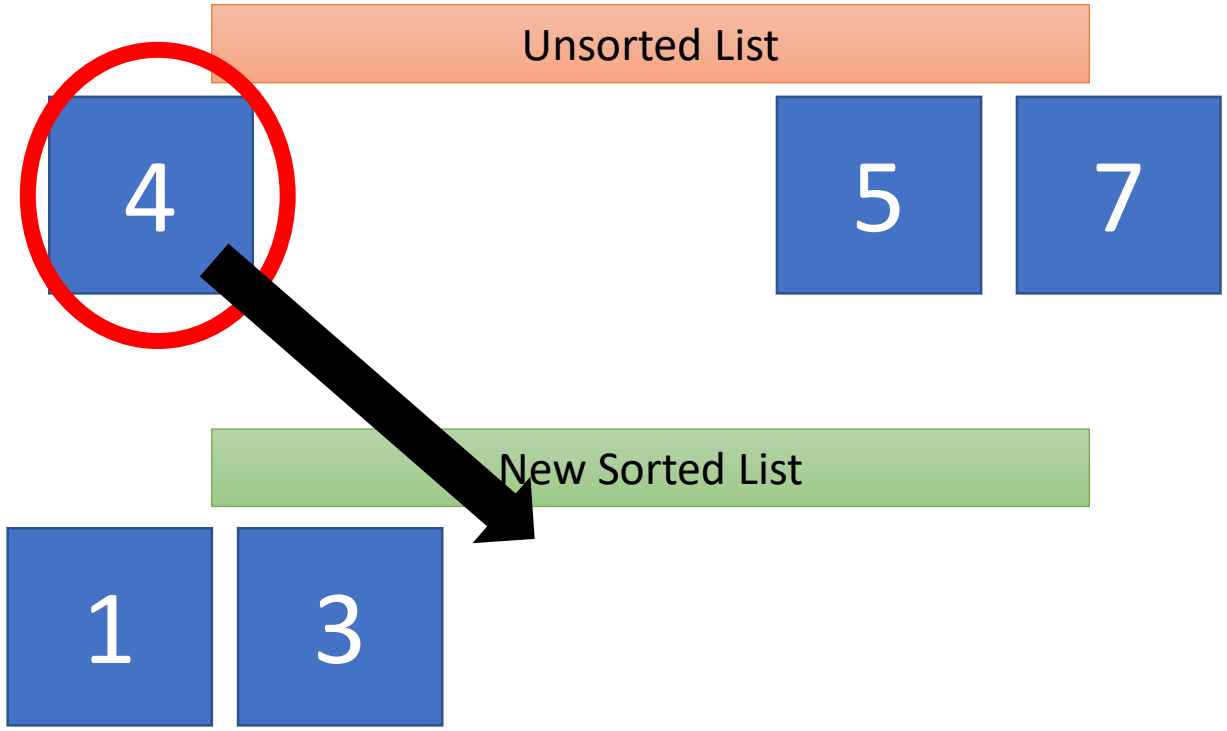
Repeat steps 1-3

Pictorial Idea of Selection Sort



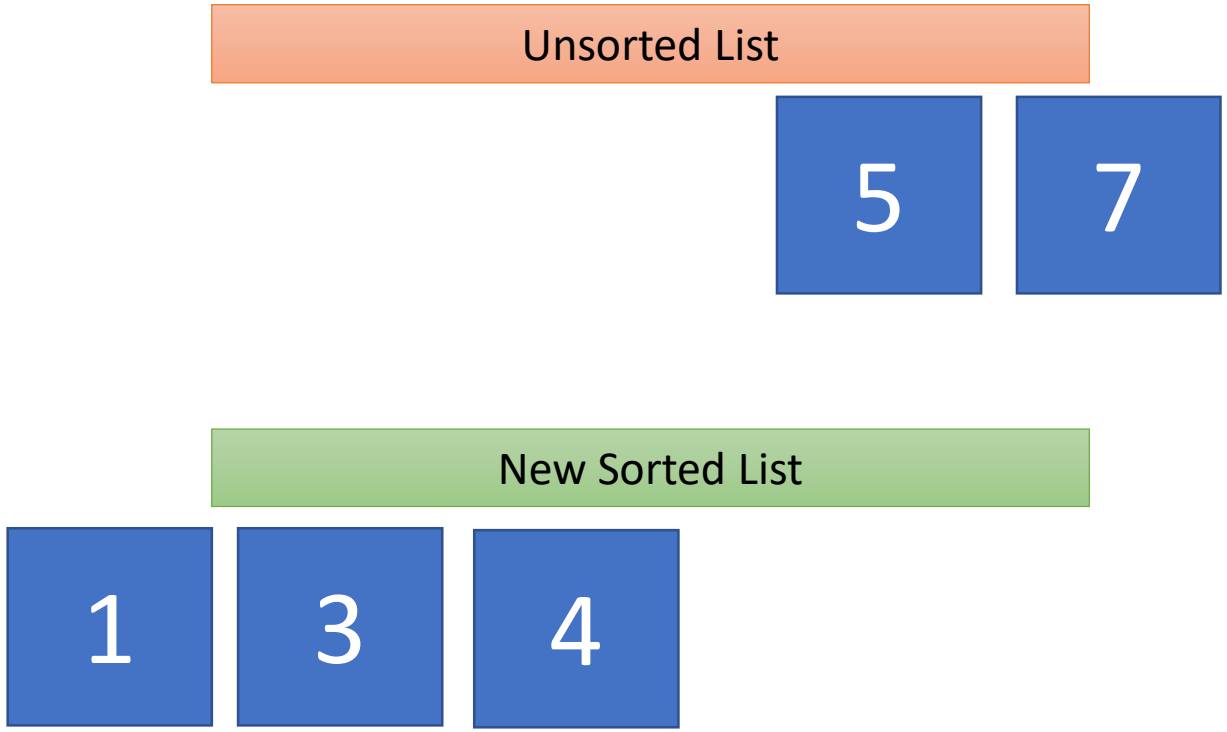
Repeat steps 1-3

Pictorial Idea of Selection Sort



Repeat steps 1-3

Pictorial Idea of Selection Sort



Repeat steps 1-3


A FEW
MOMENTS LATER

Pictorial Idea of Selection Sort

Unsorted List

New Sorted List

All done!

Basic Idea Behind Selection Sort

1. **Find the minimum element in an unsorted list.**
2. Remove that element from the unsorted list.
3. Put that element in a new sorted list.
4. Repeat steps 1-3 on the unsorted list until no elements remain.
5. Return the new sorted list.

For clarity let's first try it in Python

Step 1: Find the minimum:

```
3   #Find the minimum in a list
4   def FindMinimum(inputList):
5       minVal = sys.float_info.max #Set the minimum arbitrarily high
6       for i in range(0, len(inputList)):
7           #check if the current element in the list is smaller
8           if inputList[i] <= minVal:
9               minVal = inputList[i]
10      #Return the minimum value
11      return minVal
```

Basic Idea Behind Selection Sort

1. ~~Find the minimum element in an unsorted list.~~
2. **Remove that element from the unsorted list.**
3. Put that element in a new sorted list.
4. Repeat steps 1-3 on the unsorted list until no elements remain.
5. Return the new sorted list.

Python

Step 2: Remove an element from a list

```
13  #Removes an element with minValue from the list
14  def RemoveElement(inputList, minVal):
15      updatedList = []
16      for i in range(0, len(inputList)):
17          #Check if we found the minimum element
18          if minVal != inputList[i]:
19              updatedList.append(inputList[i])
20      return updatedList
```

Basic Idea Behind Selection Sort

1. ~~Find the minimum element in an unsorted list.~~
2. ~~Remove that element from the unsorted list.~~
3. **Put that element in a new sorted list.**
4. **Repeat steps 1-3 on the unsorted list until no elements remain.**
5. **Return the new sorted list.**

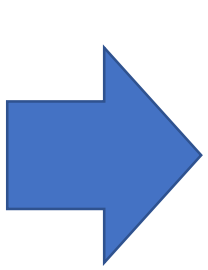
Python

Putting it all together:

```
22  def SelectionSort(inputList):
23      sortedList = []
24      for i in range(0, len(inputList)):
25          #Find the current minimum
26          currentMin = FindMinimum(inputList)
27          #Remove the current minimum and add it to our sorted list
28          inputList = RemoveElement(inputList, currentMin)
29          sortedList.append(currentMin)
30      return sortedList
```

```
22  def SelectionSort(inputList):
23      sortedList = []
24      for i in range(0, len(inputList)):
25          #Find the current minimum
26          currentMin = FindMinimum(inputList)
27          #Remove the current minimum and add it to our sorted list
28          inputList = RemoveElement(inputList, currentMin)
29          sortedList.append(currentMin)
30      return sortedList
```

```
32  sampleList = [4, 1 ,3, 7, 5]
33  #We know solution is [1,3,4,5,7]
34  print(SelectionSort(sampleList))
```

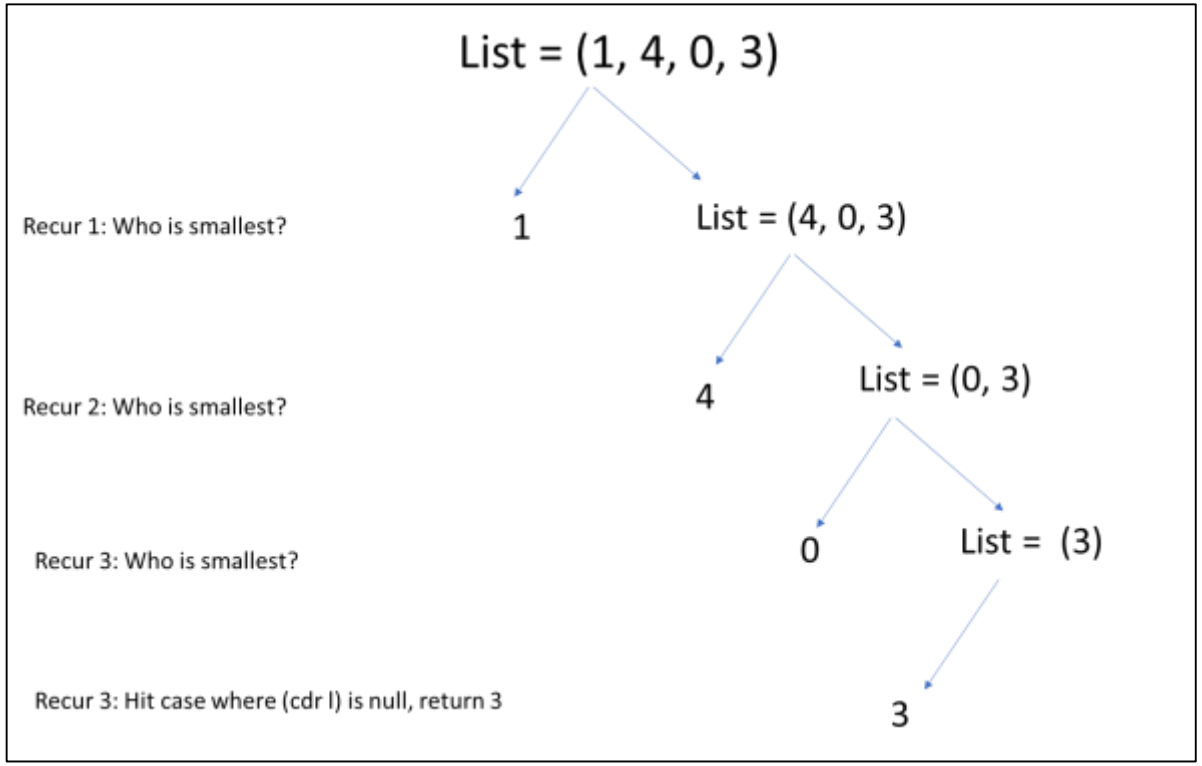


```
C:\WINDOWS\system32\cmd.exe
[1, 3, 4, 5, 7]
Press any key to continue . . .
```

Now let's do the same thing in Scheme

Step 1: Finding the minimum element in Scheme:

```
(define (smallest l)
  (define (smaller a b) (if (< a b) a b))
  (if (null? (cdr l))
      (car l)
      (smaller (car l) (smallest (cdr l)))))
```



Scheme

Step 1: Finding the minimum element in Scheme:

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements)))))))
```

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements))))))
```

v 5

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements)))))))
```

v 5

elements 4 1 3 5 7

Cons:

Remove Call 1:

Car Elements

4

Remove Call 2:

1 3 5 7

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements)))))))
```

v 5

elements 4 1 3 5 7

Remove Call 1:

Cons:

Car Elements: 4

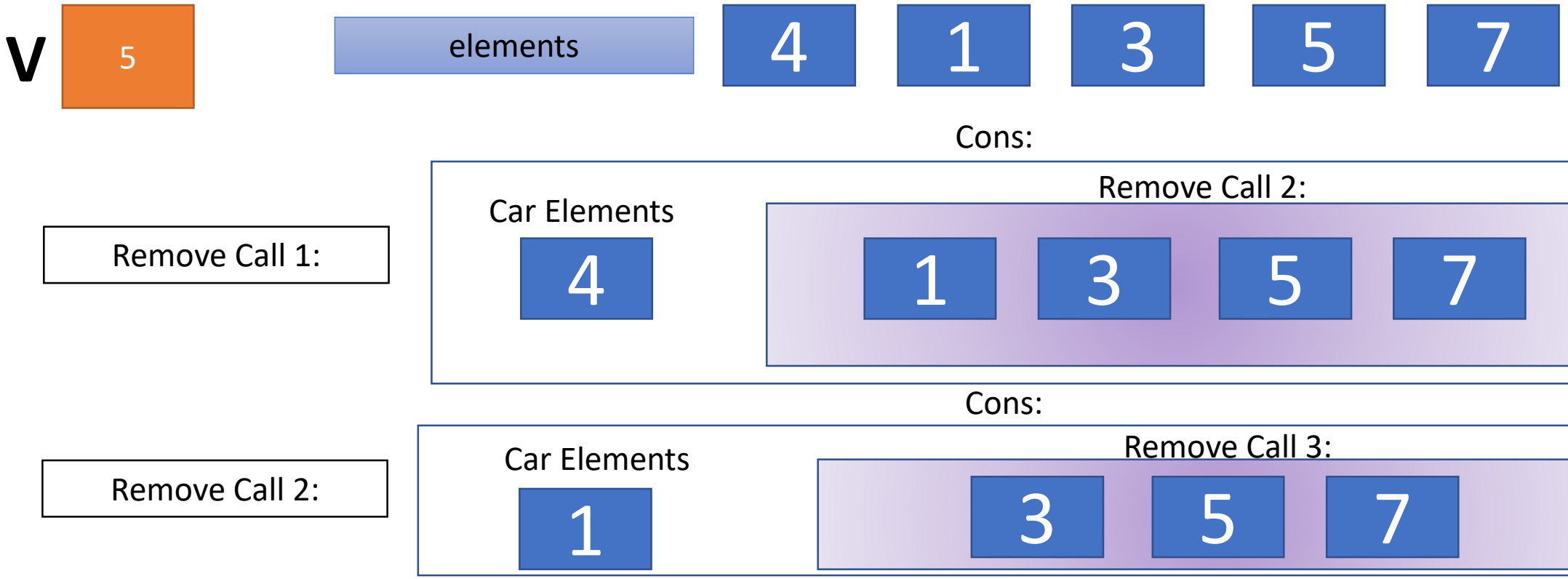
Remove Call 2: 1 3 5 7

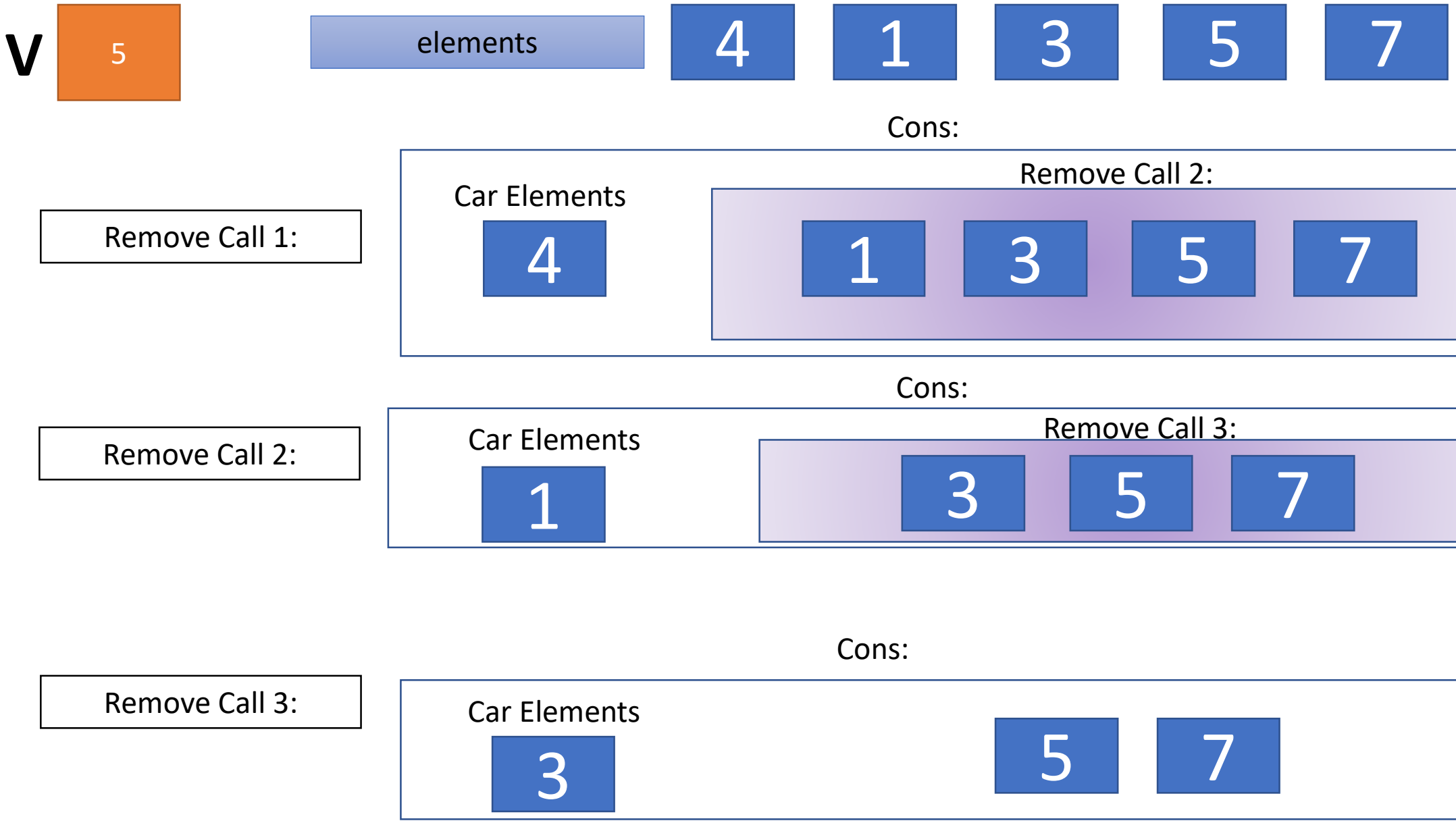
Remove Call 2:

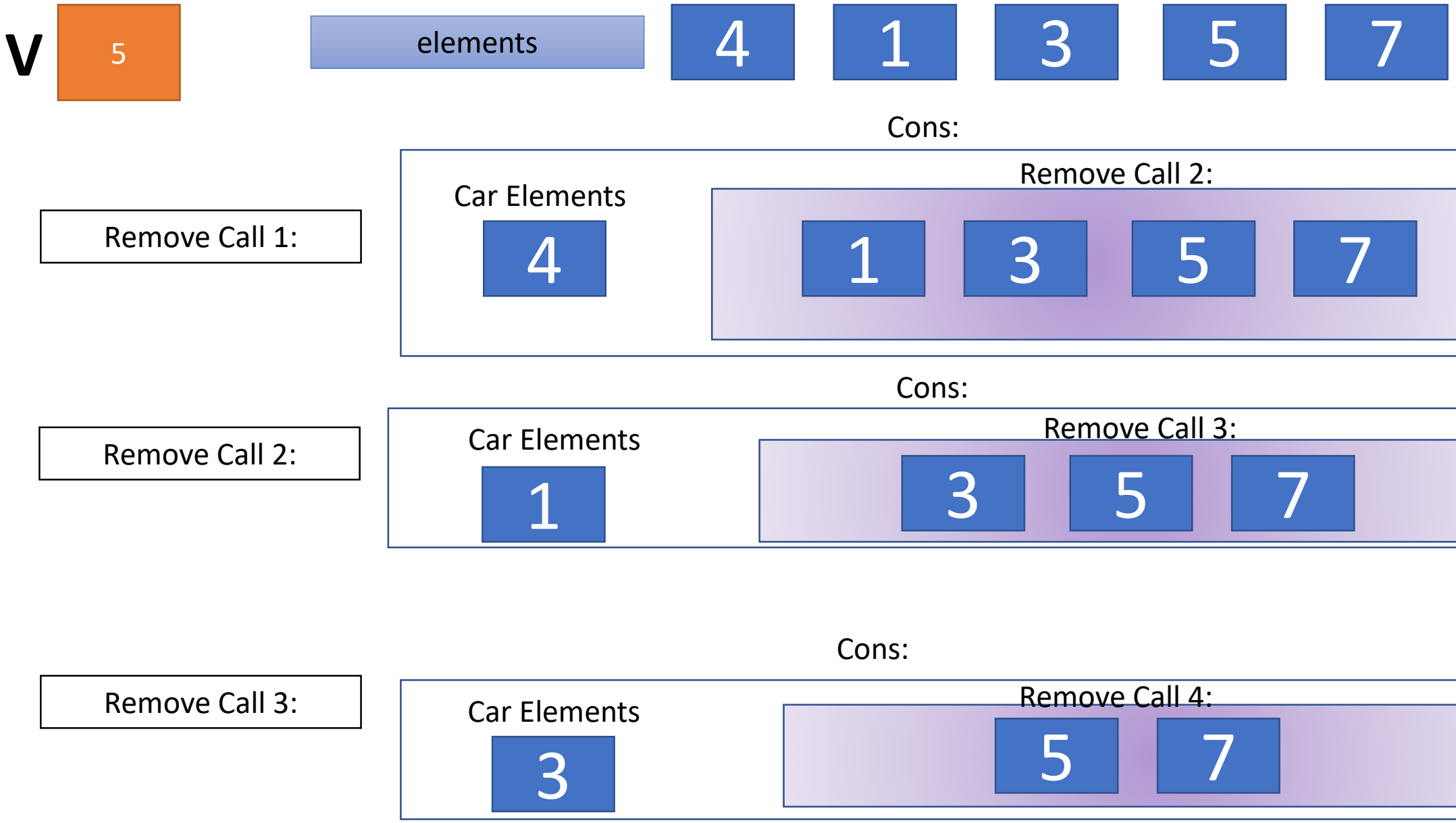
Cons:

Car Elements: 1

3 5 7

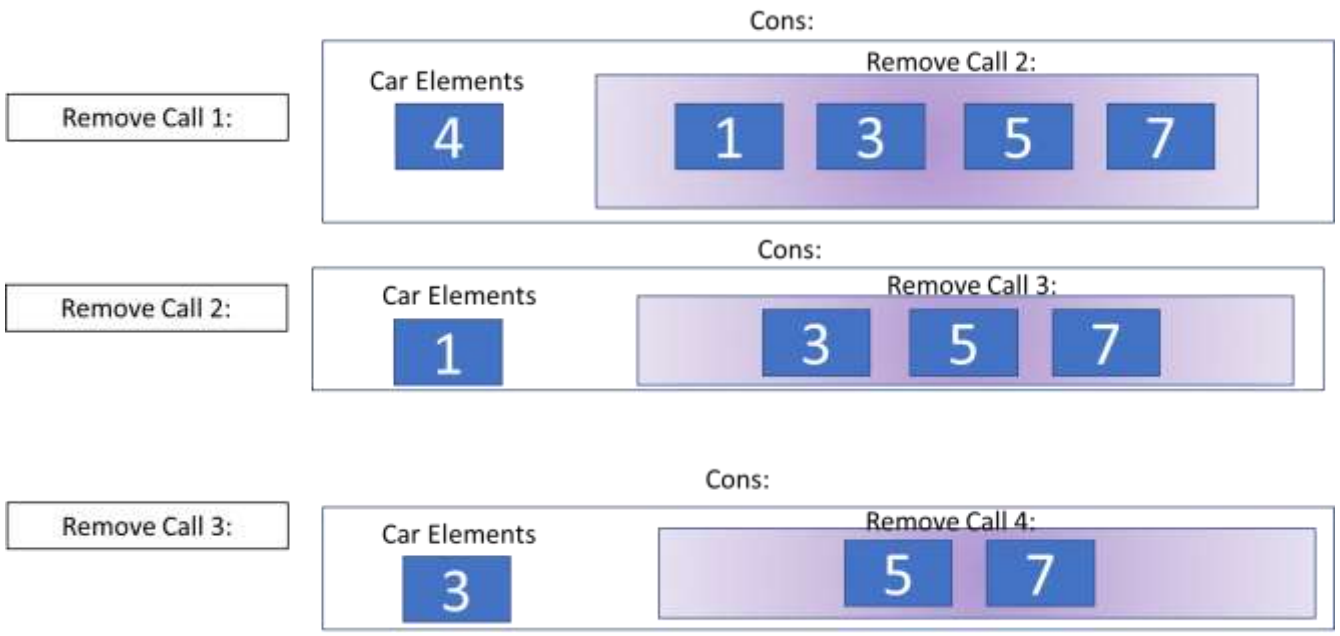

V
5
elements
4
1
3
5
7
Cons:
Remove Call 1:
Car Elements
4
Remove Call 2:
1
3
5
7
Cons:
Remove Call 2:
Car Elements
1
Remove Call 3:
3
5
7


v
5
elements
4
1
3
5
7
Cons:
Remove Call 1:
Car Elements
4
Remove Call 2:
1
3
5
7
Cons:
Remove Call 2:
Car Elements
1
Remove Call 3:
3
5
7
Cons:
Remove Call 3:
Car Elements
3
5
7


V
5
elements
4
1
3
5
7
Cons:
Remove Call 1:
Car Elements
4
Remove Call 2:
1
3
5
7
Cons:
Remove Call 2:
Car Elements
1
Remove Call 3:
3
5
7
Cons:
Remove Call 3:
Car Elements
3
Remove Call 4:
5
7

v 5

elements 4 1 3 5 7

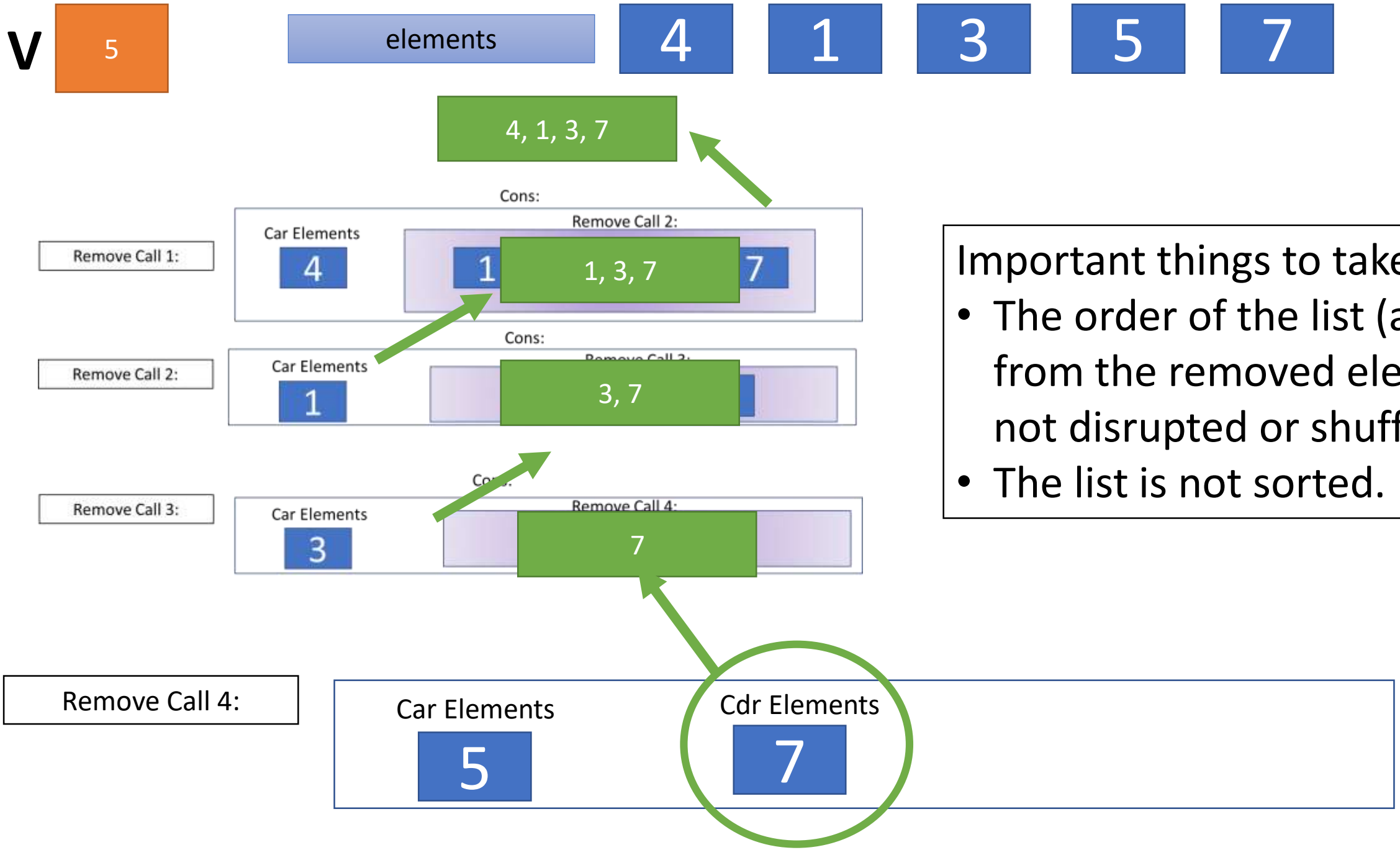



```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements))))))
```

Remove Call 4:

Car Elements: 5

Cdr Elements: 7



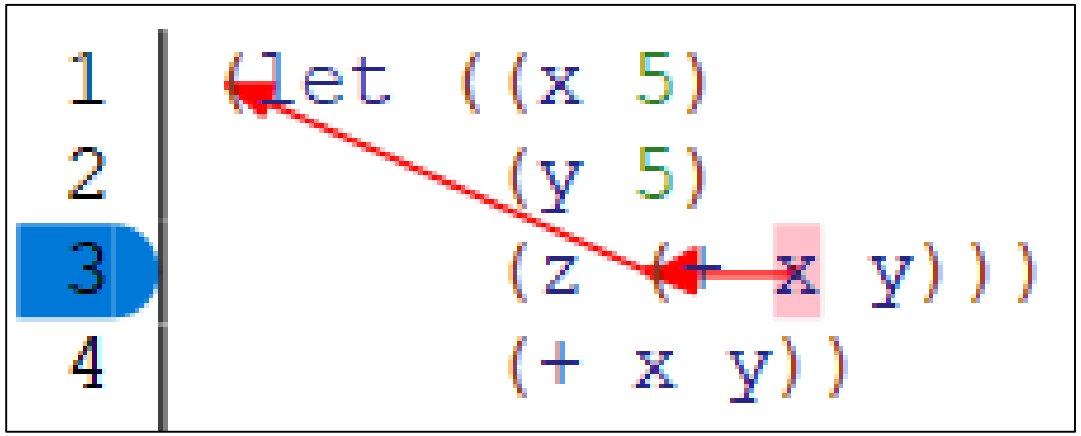
Important things to takeaway:

- The order of the list (aside from the removed element) is not disrupted or shuffled.
- The list is not sorted.

One small thing before sorting: let vs let*

- In Scheme can we write this expression?

```
1 (let ((x 5)
2       (y 5)
3       (z (+ x y)))
4       (+ x y))
```




```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
x: undefined;
 cannot reference an identifier before its definition
>
```

Limitations of let

3.9 Local Binding: `let`, `let*`, `letrec`, ...

```
(let ([id val-expr] ...) body ...+)                    syntax
(let proc-id ([id init-expr] ...) body ...+)
```

The first form evaluates the *val-expr*s left-to-right, creates a new location for each *id*, and places the values into the locations. It then evaluates the *body*s, in which the *id*s are bound. The last *body* expression is in tail position with respect to the `let` form. The *id*s

```
(let* ([id val-expr] ...) body ...+)                   syntax
```

Like `let`, but evaluates the *val-expr*s one by one, creating a location for each *id* as soon as the value is available. The *id*s are bound in the remaining *val-expr*s as well as the

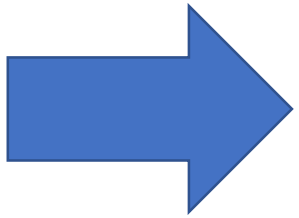
- What does all this fancy lingo mean?
- In simple terms let* allows you to create variables that have dependencies on one another, it's a sequential evaluation.

Source: https://docs.racket-lang.org/reference/let.html

Using let*

- In Scheme can we write this expression?

```
1 (let* ((x 5)
2        (y 5)
3        (z (+ x y)))
4        (+ x y))
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
10
>
```

Now back to sorting…

PUTTING THE PIECES TOGETHER TO SORT

- Use smallest and remove!

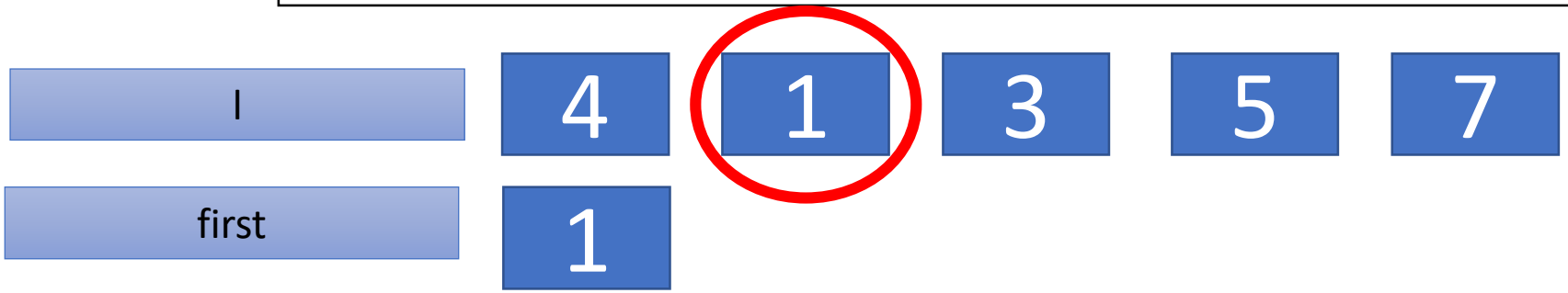
```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort rest)))))
```

- Use a `let*`
  - To first bind `first` to the smallest element of the list;
  - Then *use* `first`'s value to trim the list.

Understanding the code in Scheme

```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
```

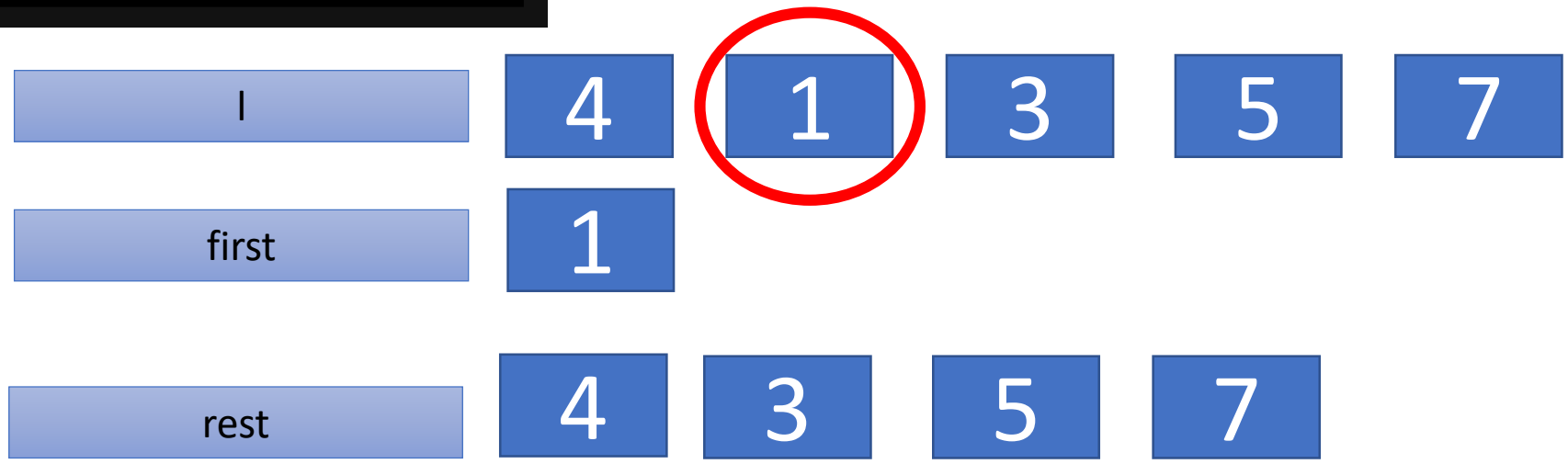
Step 1: Find the smallest element in the current list.



Understanding the code in Scheme

```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
```

Step 2: Remove the smallest element from the list

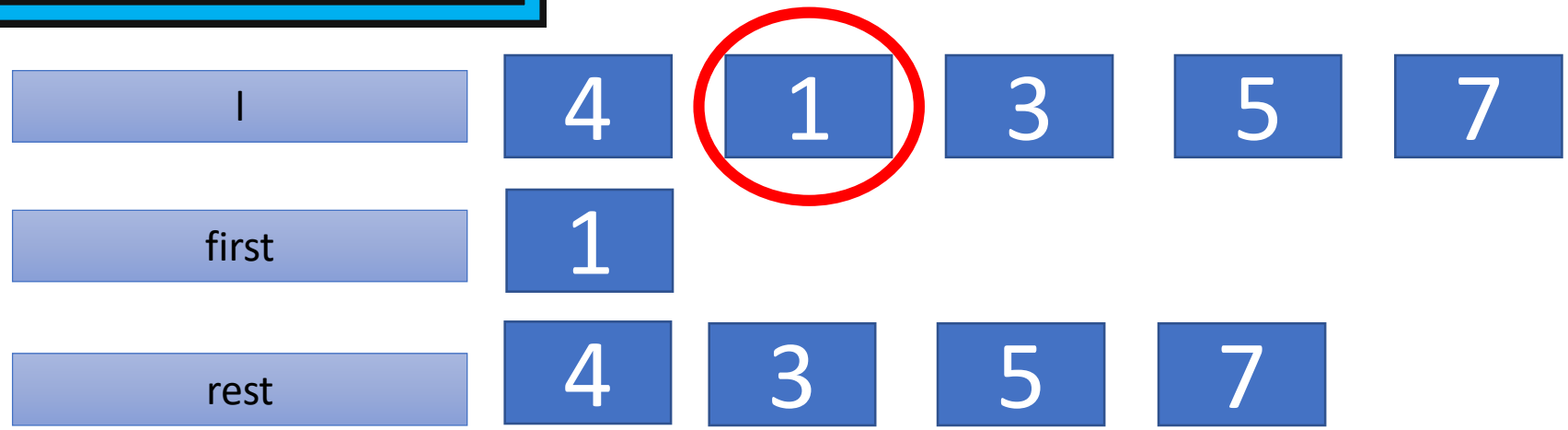


Now what do we need to do next?

Understanding the code in Scheme

```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort rest)))))
```

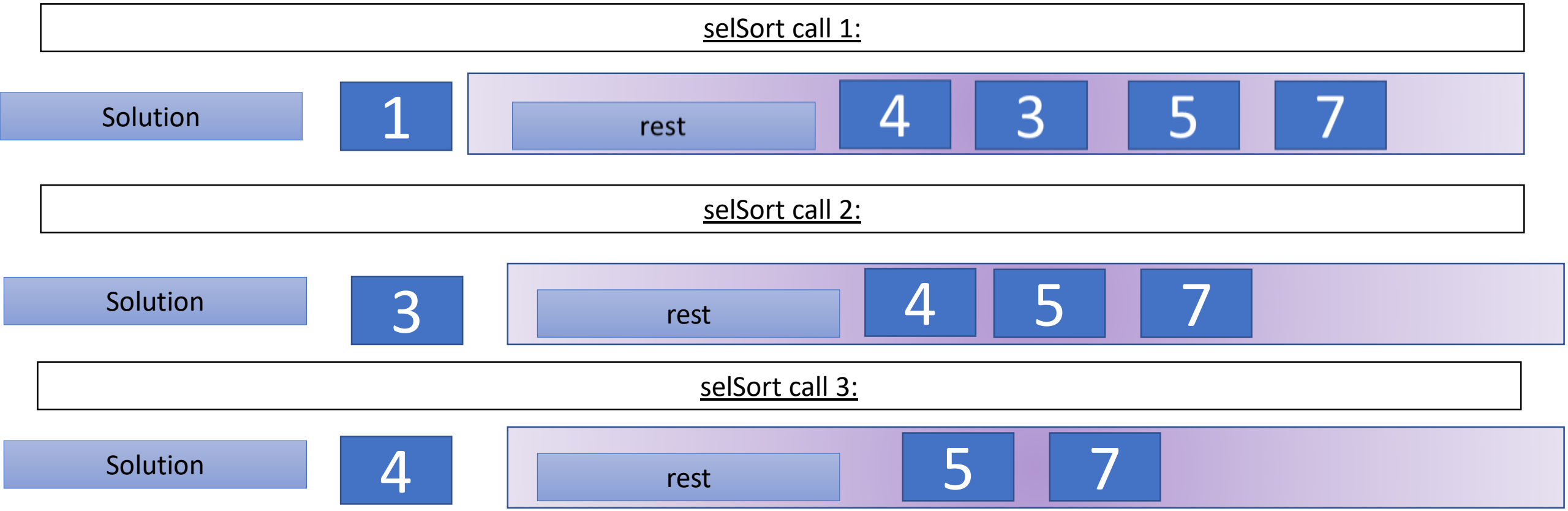
Step 3: Start creating the solution and recur



Call selSort a second time with this input:

What will the code look like then?

- Purple box = need to do another call to selSort with the input being whatever is in the purple box



PUTTING THE PIECES TOGETHER TO SORT

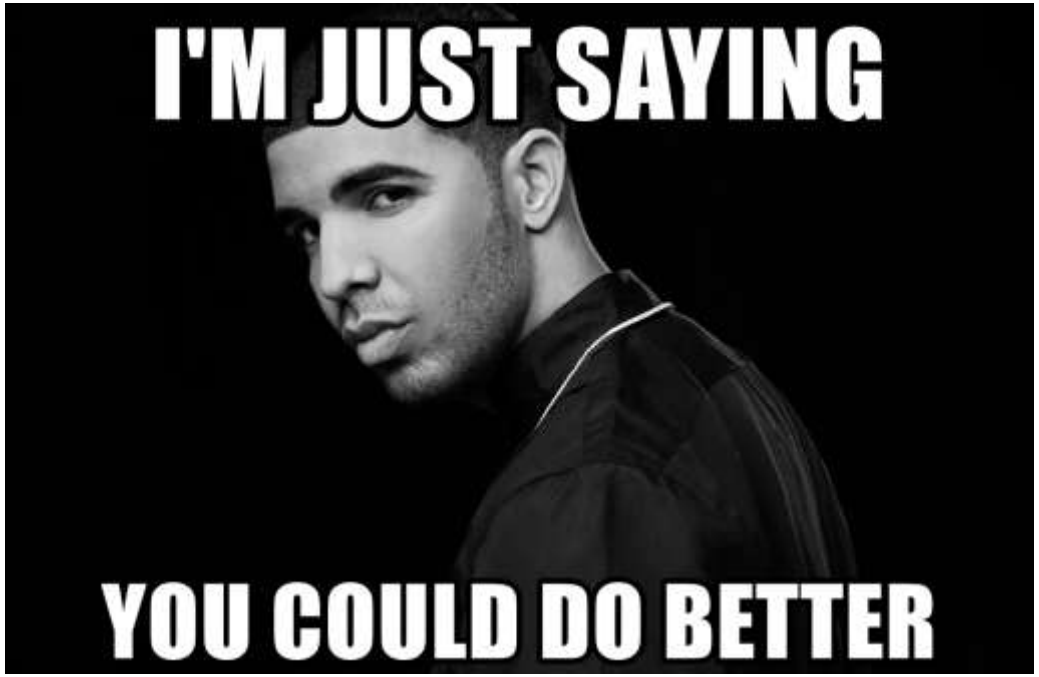
- Use smallest and remove!

```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort rest)))))
```

One question: Could we have used let instead of *let in the above code?

AND...TO MINIMIZE CLUTTER

```
(define (selSort l)
  (define (smallest l)
    (define (smaller a b) (if (< a b) a b))
    (if (null? (cdr l))
        (car l)
        (smaller (car l) (smallest (cdr l)))))
  (define (remove v l)
    (if (null? l)
        l
        (if (equal? v (car l))
            (cdr l)
            (cons (car l) (remove v (cdr l))))))
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort r)))))
```



Singer Drake famously said the above quote. But why?

He was referencing the fact that in the original Scheme implementation of selection sort we traverse the list more times than necessary.

NO NEED TO TRAVERSE THE LIST TWICE;

ONE PASS EXTRACTION & MINIMIZATION

- Goal
  - Find and Extract the smallest element from a list (in one pass!).
- Idea
  - Return two things (a pair!)
    - The extracted element
  - The list without the extracted element.

MINIMIZATION AND EXTRACTION IN ONE SWEEP

- To improve readability, we introduce *convenience* functions to make & consult pairs.
- Reserve `cons`/`car`/`cdr` for list operations

```
(define (make-pair a b) (cons a b))
(define (first p) (car p))
(define (second p) (cdr p))
```

Assume ℓ has at least one element

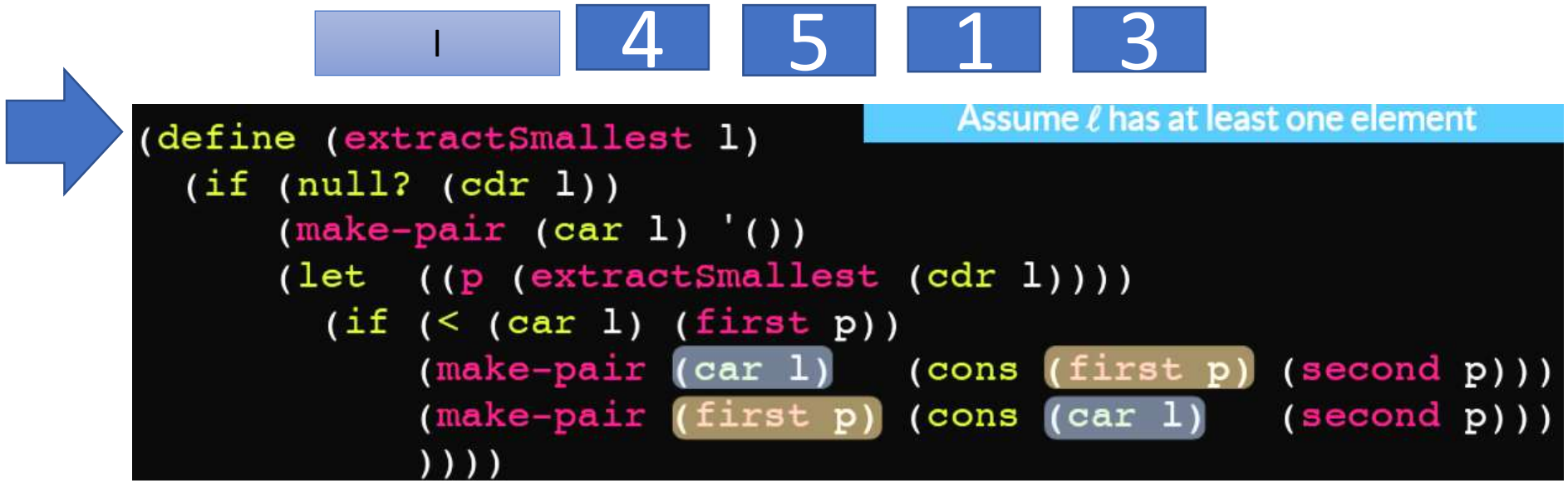
```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

What is going on in this code?

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

- Basic idea: Go through the list and recur until you reach a list with only one element. Then recur back up and keep putting the smallest element at the start of the list.

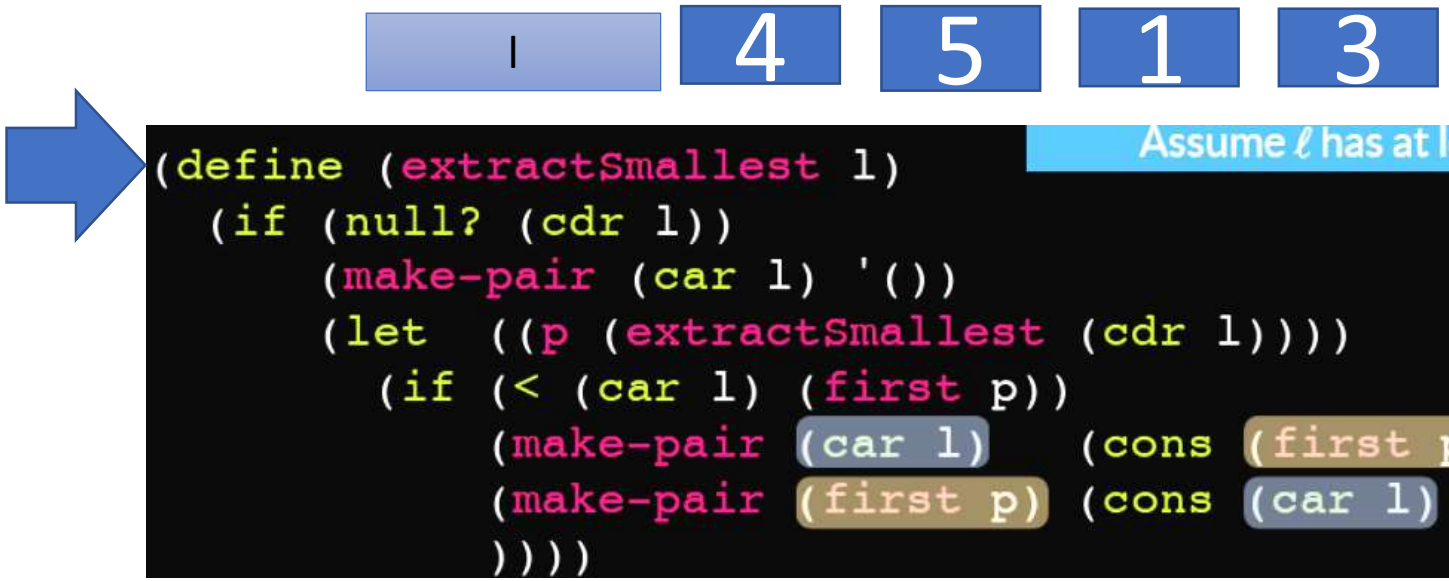


ES Call #1: 4 5 1 3


```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element

ES Call #1:



ES Call #1:

ES Call #2:


```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element

ES Call #1:

ES Call #2:

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1:

ES Call #2:

ES Call #3:

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1:

ES Call #2:

ES Call #3:

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1: 

ES Call #2: 

ES Call #3: 

ES Call #4: 

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1: 

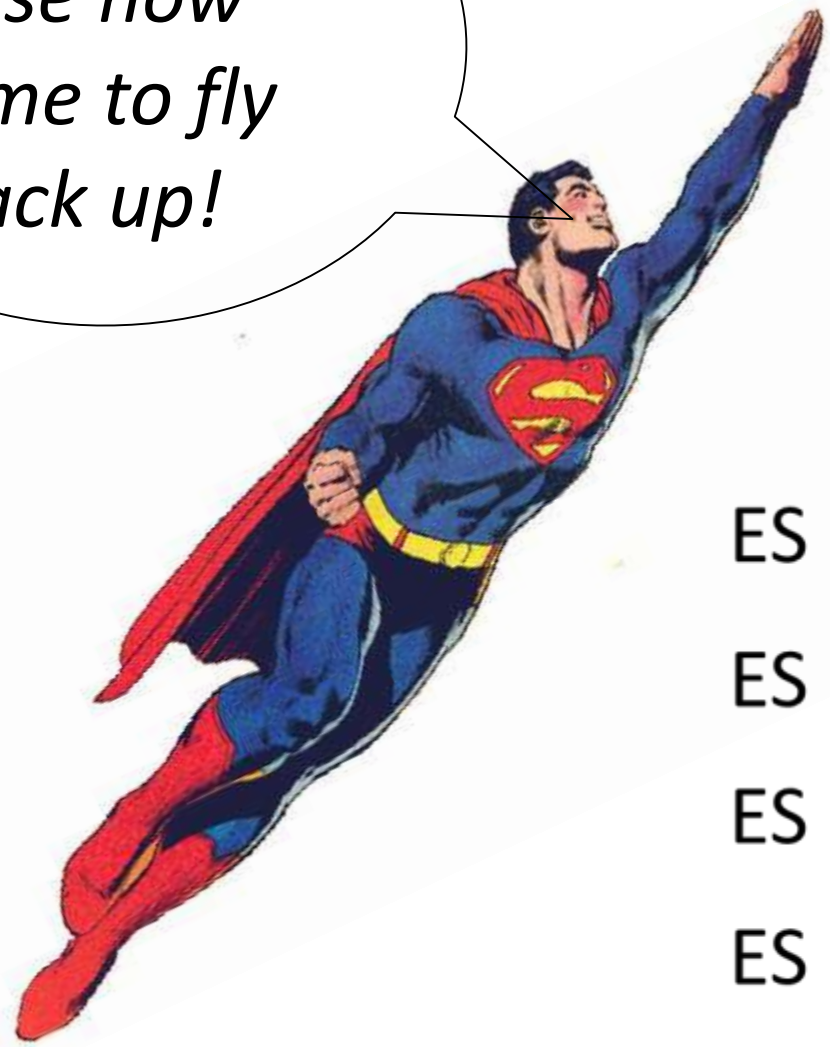
ES Call #2: 

ES Call #3: 

ES Call #4: 

What happens next?

Hit the base case now time to fly back up!



Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1: | 4 | 5 | 1 | 3 |

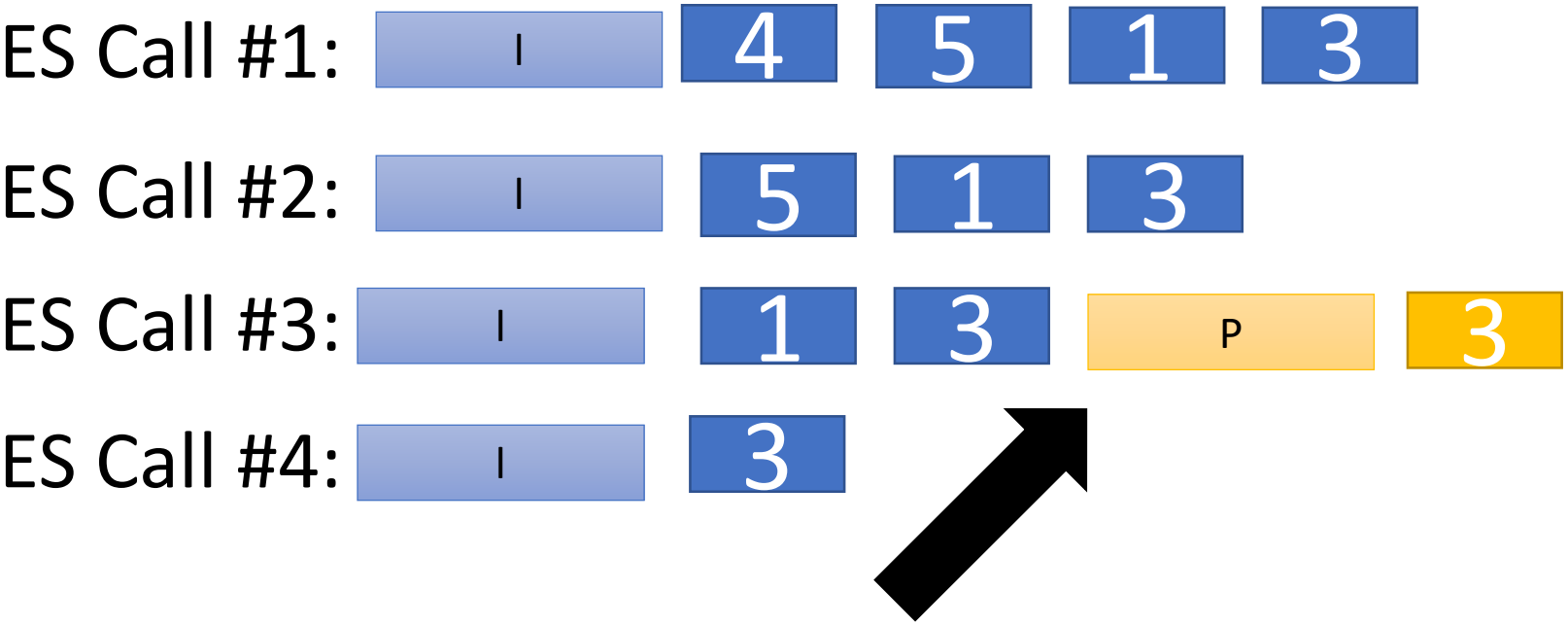
ES Call #2: | 5 | 1 | 3 |

ES Call #3: | 1 | 3 |

ES Call #4: | 3 |

Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```



Assume ℓ has at least one element

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #1: l 4 5 1 3

ES Call #2: l 5 1 3

ES Call #3: l 1 3 P 3

ES Call #4: l 3

- Now we are constructing a new list and checking who is smaller. The start of P or the start of l.
- Whoever is smallest we put at the head of the new list and return

```
(define (extractSmallest l)                          Assume ℓ has at least one element
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

ES Call #4: l | 1 | 3 | P | 3

Car l = 1 < First P = 3

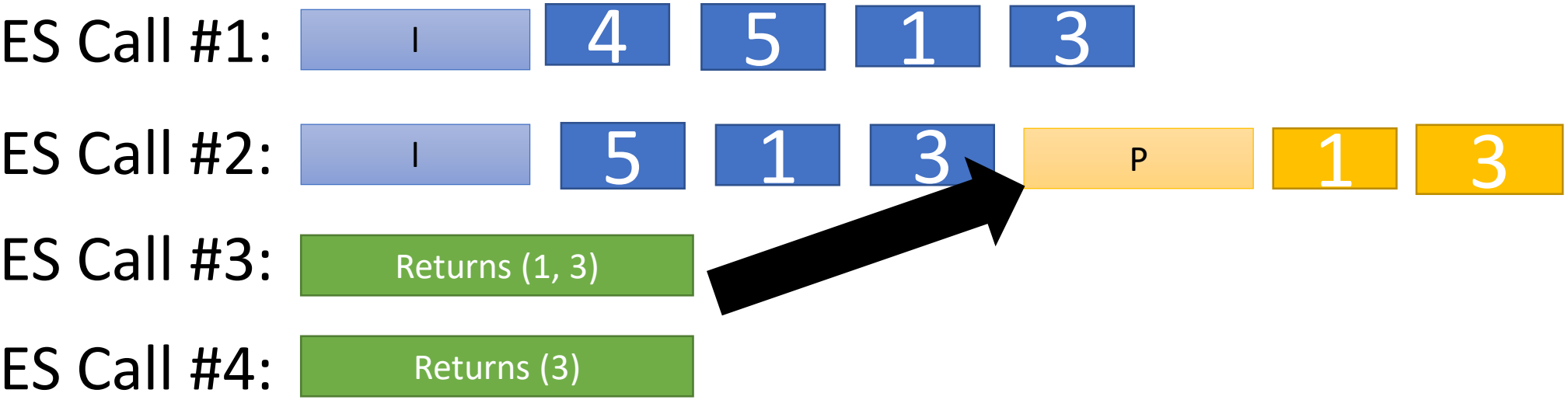
So that means we build the return list like this:

```
(make-pair (car l)   (cons (first p) (second p)))
```

Return: 1 | 3 | Empty List


```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element



l | 4 | 5 | 1 | 3

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element

ES Call #2: l | 5 | 1 | 3 | P | 1 | 3

Car l = 5 < First P = 1

```
(make-pair (first p) (cons (car l)   (second p)))
```

1 | 5 | 3


```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element

ES Call #1:

ES Call #2: Returns (1, 5, 3)

ES Call #3: Returns (1, 3)

ES Call #4: Returns (3)

l	4	5	1	3

```
(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)   (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)   (second p)))
            ))))
```

Assume ℓ has at least one element

ES Call #1:

l	4	5	1	3

P	1	5	3

Car l = 4 < First P = 1

```
(make-pair (first p) (cons (car l)   (second p)))
```

1	4	5	3

Summary Of What We Just Did:

START: 4 5 1 3

END: 1 4 5 3

- We showed that we can improve the sorting algorithm by finding the smallest AND removing the smallest with only one list traversal (instead of two).
- We traced an example of this pictorial in Scheme.

Figure Sources

- https://i.ytimg.com/vi/-2Z0Y3Kk8nU/maxresdefault.jpg
- https://docs.racket-lang.org/reference/let.html
- https://memegenerator.net/img/instances/33234550.jpg
- https://wompampsupport.azureedge.net/fetchimage?siteId=7575&v=2&jpgQuality=100&width=700&url=https%3A%2F%2Fi.kym-cdn.com%2Fentries%2Ficons%2Fmobile%2F000%2F018%2F489%2Fnick-young-confused-face-300x256-nqlyaa.jpg
- https://qph.fs.quoracdn.net/main-qimg-c4be6b43fcd817d94a78cde18074f7b0-pjlq