PG4200: Algorithms And Data Structures

Lesson 04: Recursion and Test Driven Development (TDD)

Bogdan Marculescu

Recursion

Wikipedia Definition

Recursion *in computer science is a method of solving a problem where the solution depends on solutions to* ***smaller instances*** *of the same problem (as opposed to iteration)… Most computer programming languages support recursion by allowing a function to* ***call itself*** *from within its own code.*

General Definition

Recursion: see definition of **Recursion**

recursion

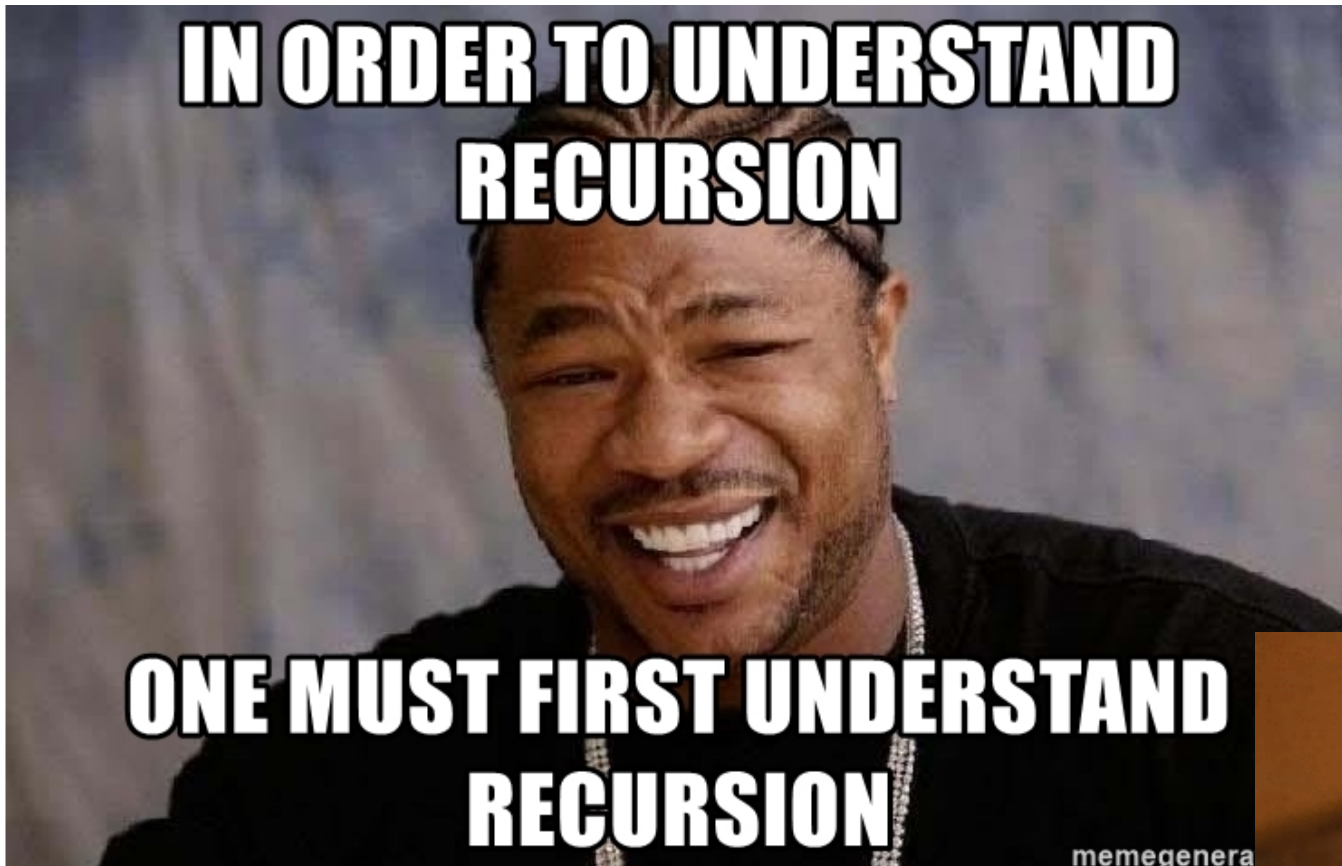
All Images Videos News Books More Tools

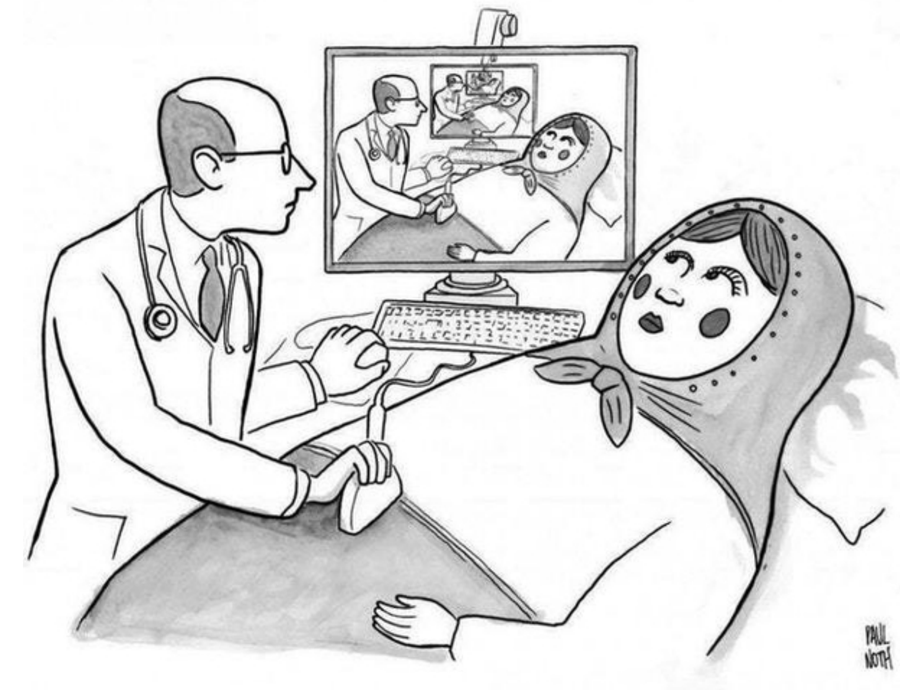
About 666,000,000 results (0.51 seconds)

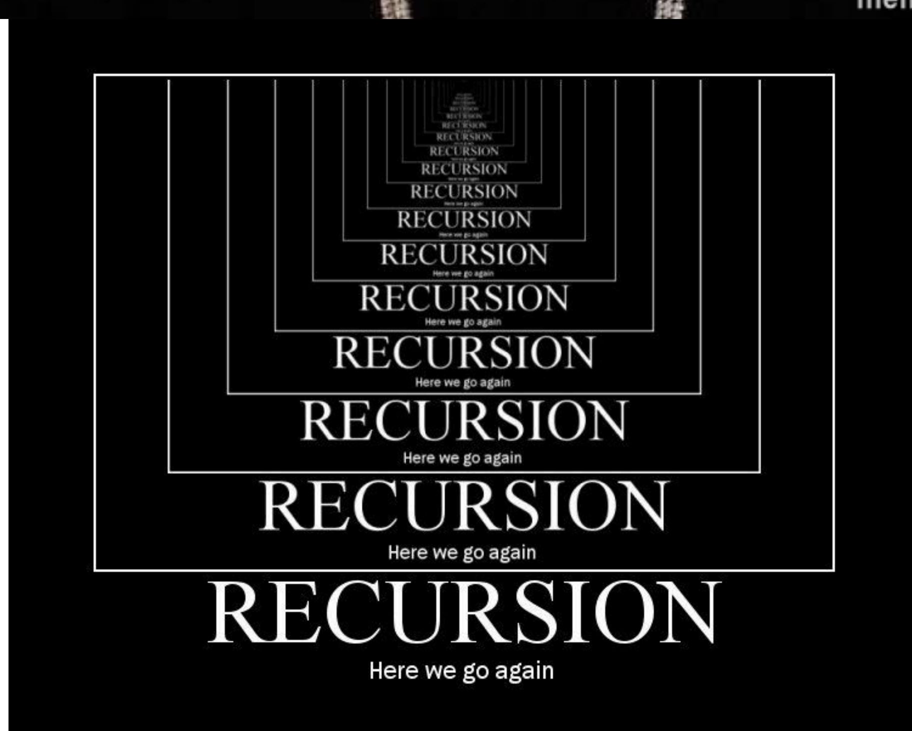
Did you mean: ***recursion***

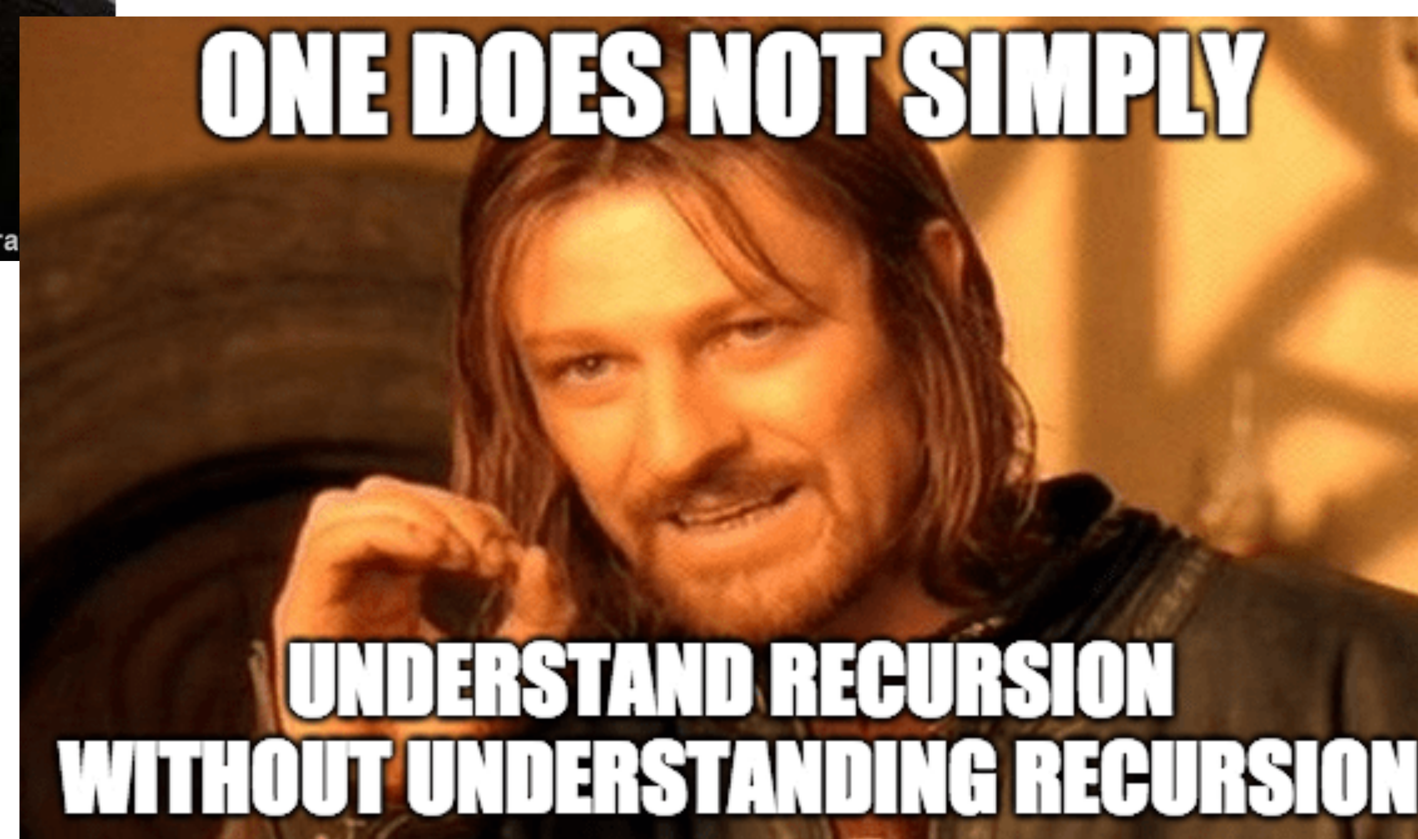
Recursion is **the process of defining a problem (or the solution to a problem)** in terms of (a simpler version of) itself. For example, we can define the operation "find your way home" as: If you are at home, stop moving.

A tricky concept first time you see it… as our brains are wired to think more in an *iterative* way… but maybe memes help…









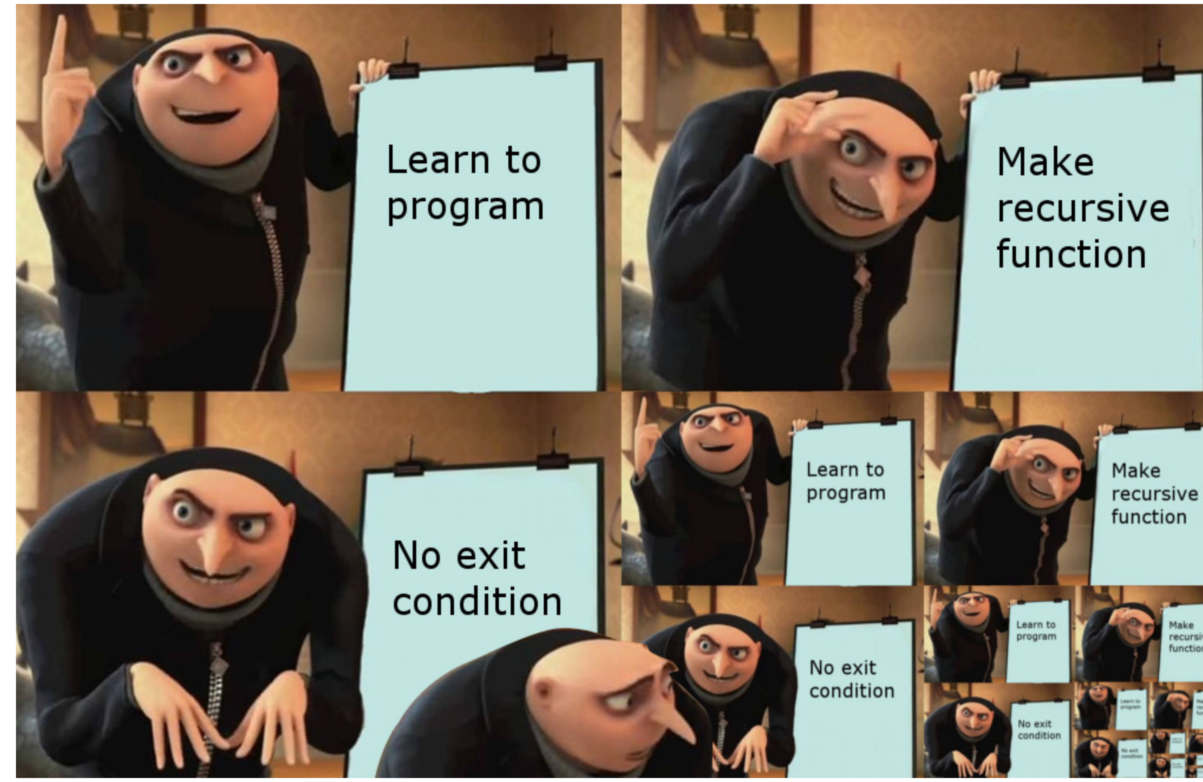
Let’s Try With Math…

- Example: sum all values up to *n*
- *f(n) = 1 + 2 + 3 + 4 + … + (n-2) + (n-1) + n*
    - formally, $f(n) = \sum_{i=1}^{i=n} i$
- eg, *f(5) = 1 + 2 + 3 + 4 + 5 =15*
- Could be implemented with a *for*-loop, where *i* is added to a variable *sum*
- How does a recursive version look like?

Recursive Sum

- Idea: if we can sum up to $n-1$, then, to sum up to n, we can just add n
- $f(n) = n + f(n-1)$
- eg, $f(5) = 5 + f(4)$
  - f(4) = 4 + f(3)
  - f(3) = 3 + f(2)
  - f(2) = 2 + f(1)
  - …

Stopping/Exit Criterion

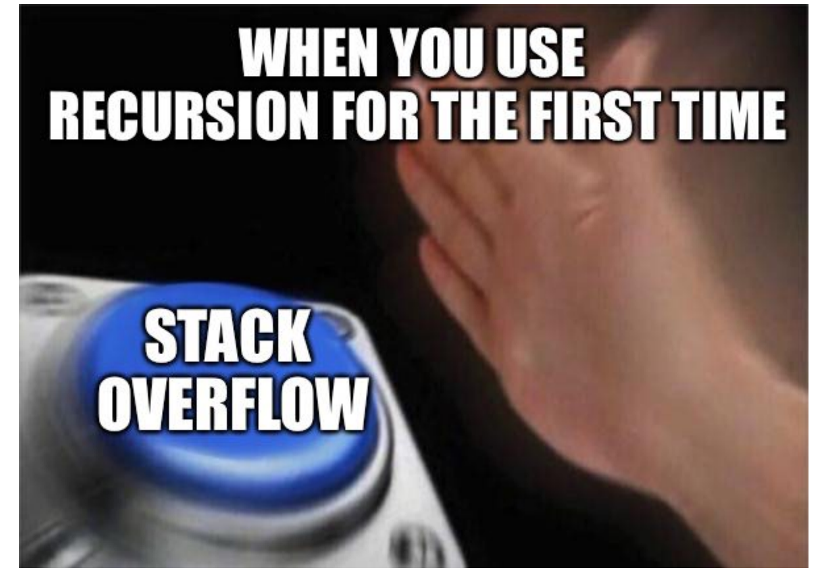


- *f(n) = n + f(n-1)*

- *f(5) = 5 + f(4)*
    - *f(4) = 4 + f(3)*
    - *f(3) = 3 + f(2)*
    - *f(2) = 2 + f(1)*
    - *f(1) = 1 + f(0)*
    - *f(0) = 0 + f(-1)*
    - *f(-1) = -1 + f(-2)*
    - *f(-2) = -2 + f(-3)*
    - *etc.*

- **We must have a stopping criterion**
- Otherwise recursive call will go on "forever"
- We need to define a base value for which we do not make a recursive call
- $f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ n + f(n-1) & otherwise \end{cases}$

What If No Stopping?



- Before *f(n)* is completed, *f(n-1)* must be completed
- But before *f(n-1)* is completed, *f(n-2)* must be completed
- And so on…
- At each function call, we push a frame on the function call stack
- If no stopping criterion, we will push so many frames that we will run out of memory eventually
  - leading so to a *stack overflow* exception

Number of Recursive Calls

- Considering $f(n) = n + f(n-1)$, how many frames will be pushed?
    - assume stopping condition for n<=1
- We would have $O(n)$ frames
- Even if stopping condition, we can get a stack overflow for large enough n
    - eg, $n = 100,000$

Reduced Input

- In each recursive call, we must reduce the input
- How much input is reduced has huge impact on viability of the recursive algorithm
  - we must avoid stack overflows
- If the decrease is linear, then we will have a linear number of frames on stack
  - which will likely lead to a stack overflow for non-small inputs

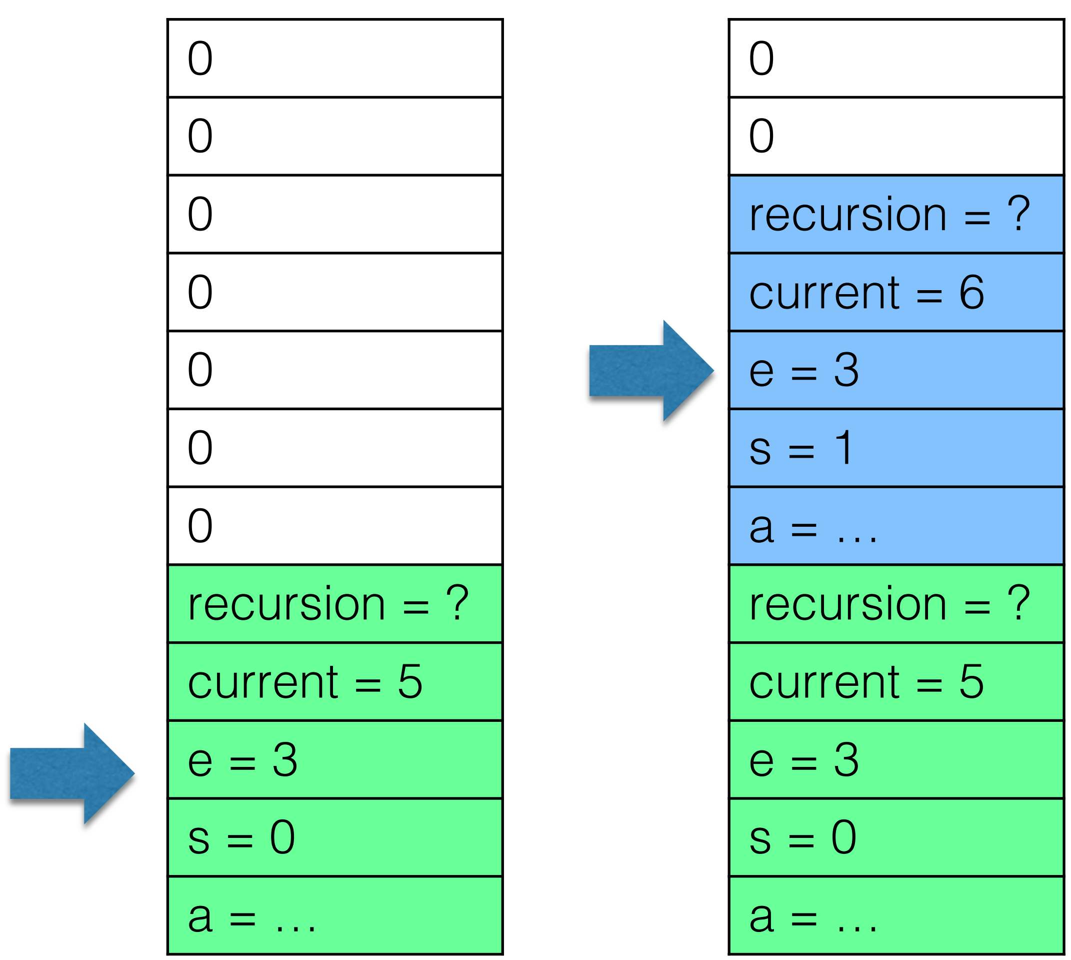
Sum Array Example

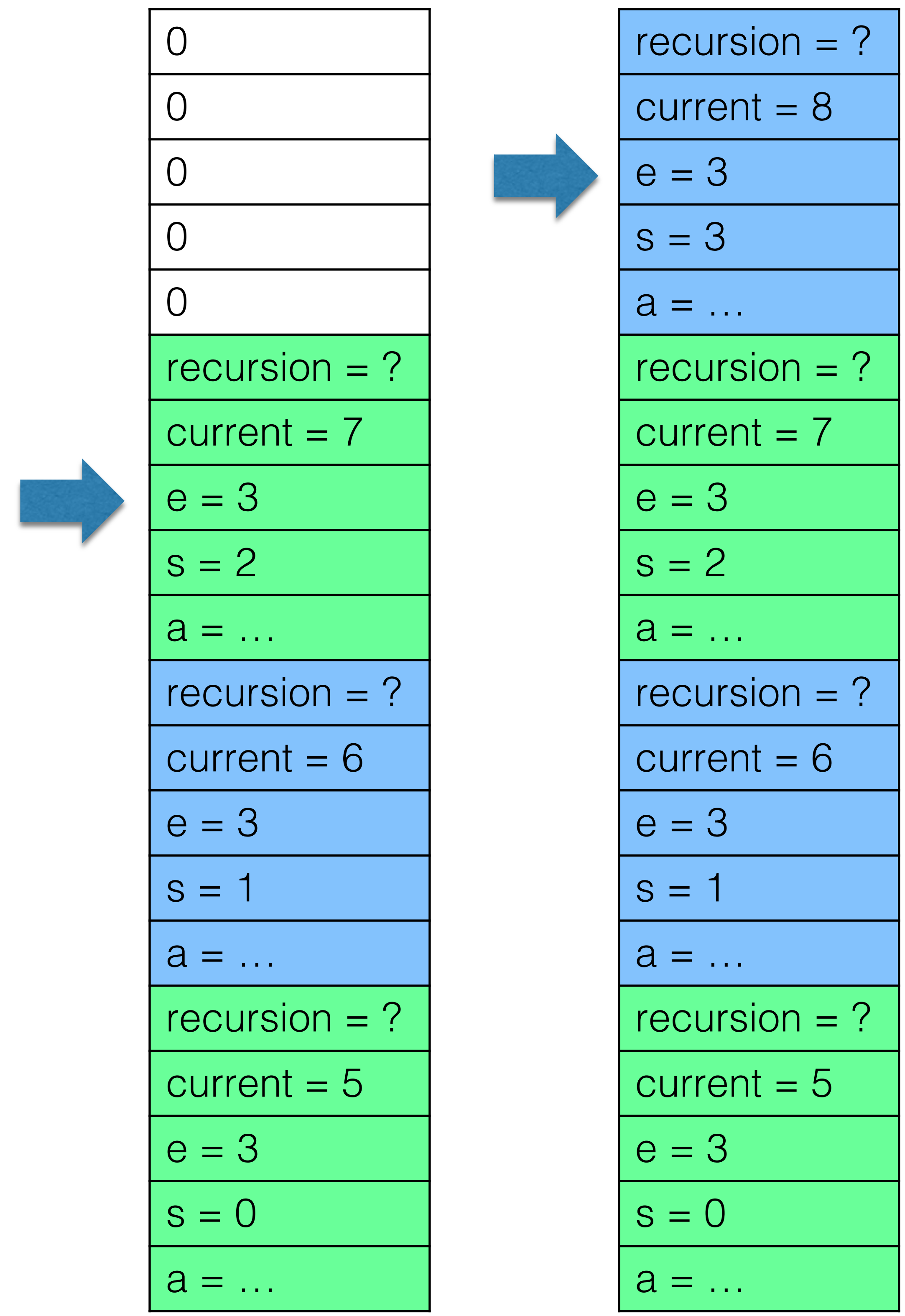
- Let's sum all values in an array *a*, from start index *s* to end index *e*
    - where *sum(a) = sum(a,0,a.length-1)*
- Considering 2 different recursive versions
- *sumX(a,s,e) = a[s] + sumX(a,s+1,e)*
- *sumY(a,s,e) = sumY(a,s,middle) + sumY(a,middle+1,e)*
- Stopping: *s==e -> sum(a,s,e)=a[s]*
- Both give the right answer, but *which version is better?*

sumX(a,s,e) = a[s] + sumX(a,s+1,e)

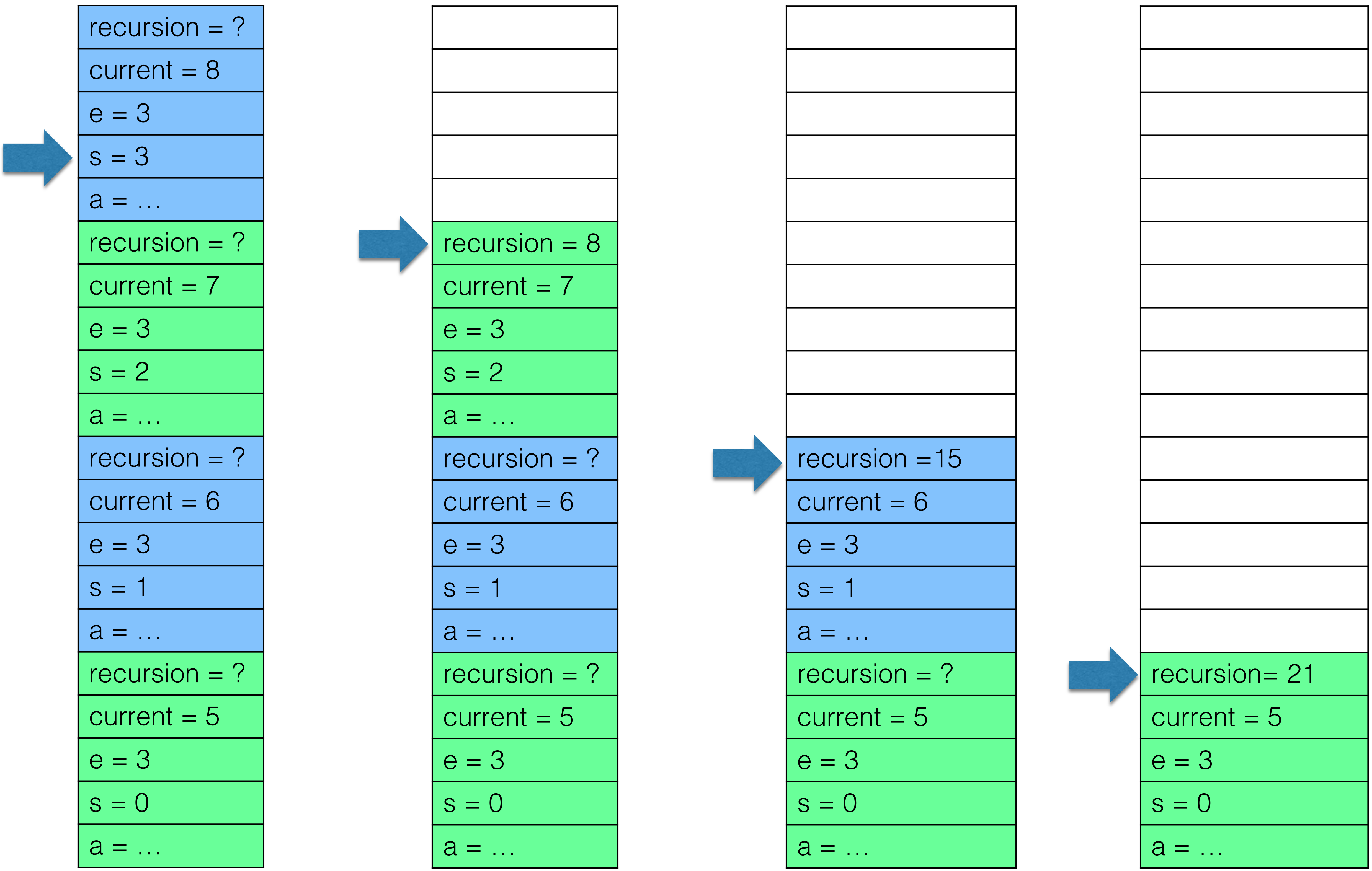
// a = {5,6,7,8}

```
sumX(a,s,e){
    current = a[s]
    s==e ? return current
    recursion = sumX(a,s+1,e)
    return current + recursion
}
```





- Until we reach stopping condition *s==e*, we keep on making recursive calls, and push new frames on stack
- Recall a frame is popped only when its function is terminated



sumY(a,s,e) = sumY(a,s,middle) + sumY(a,middle+1,e)

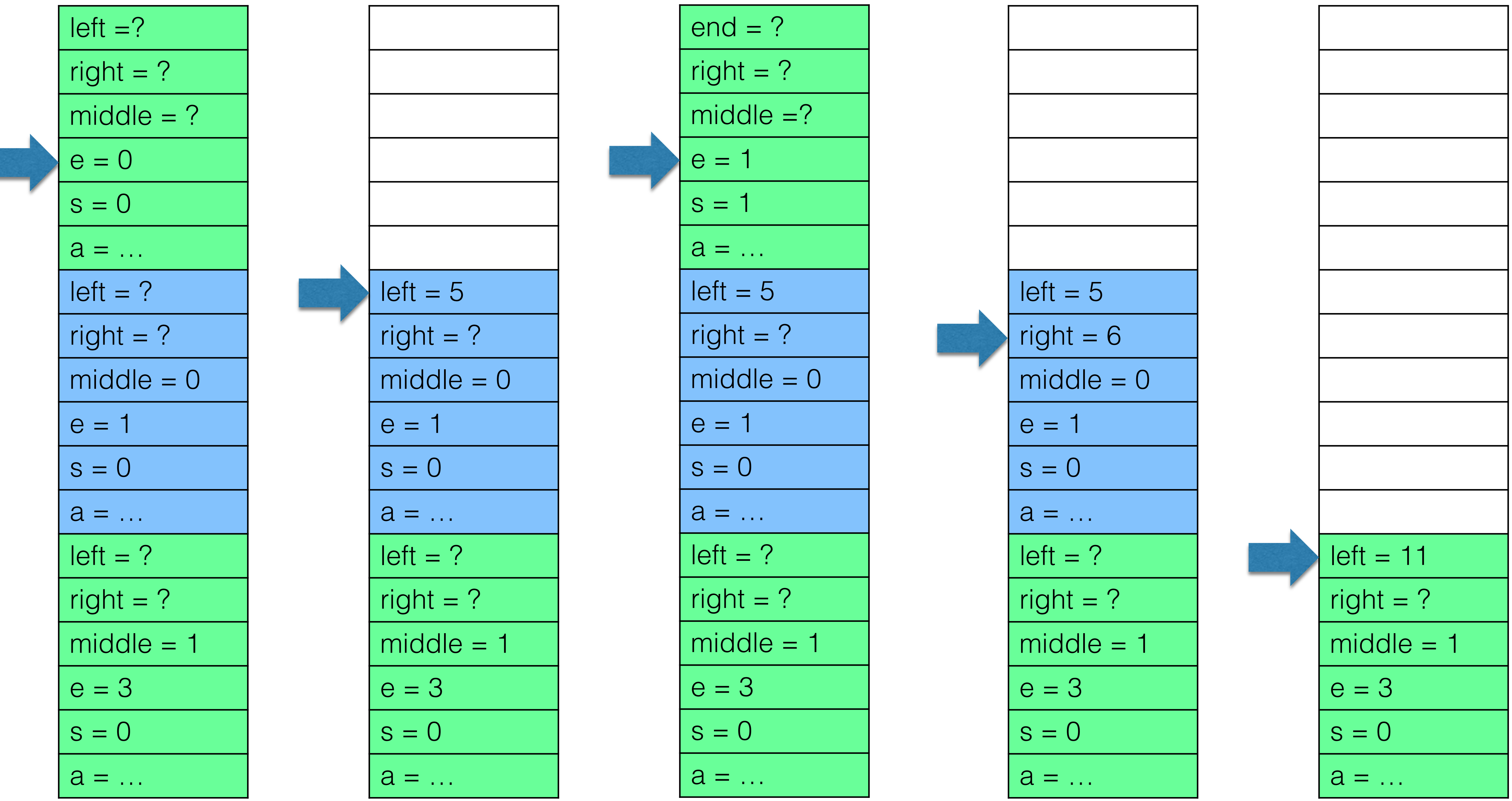
```
// a = {5,6,7,8}

sumY(a,s,e){
    s==e ? return a[s]
    middle = (s+e)/2
    left = sumY(a,s,middle)
    right = sumY(a,middle+1,e)
    return left + right
}
```

0
0
0
0
0
0
left = ?
right = ?
middle = 1
e = 3
s = 0
a = …

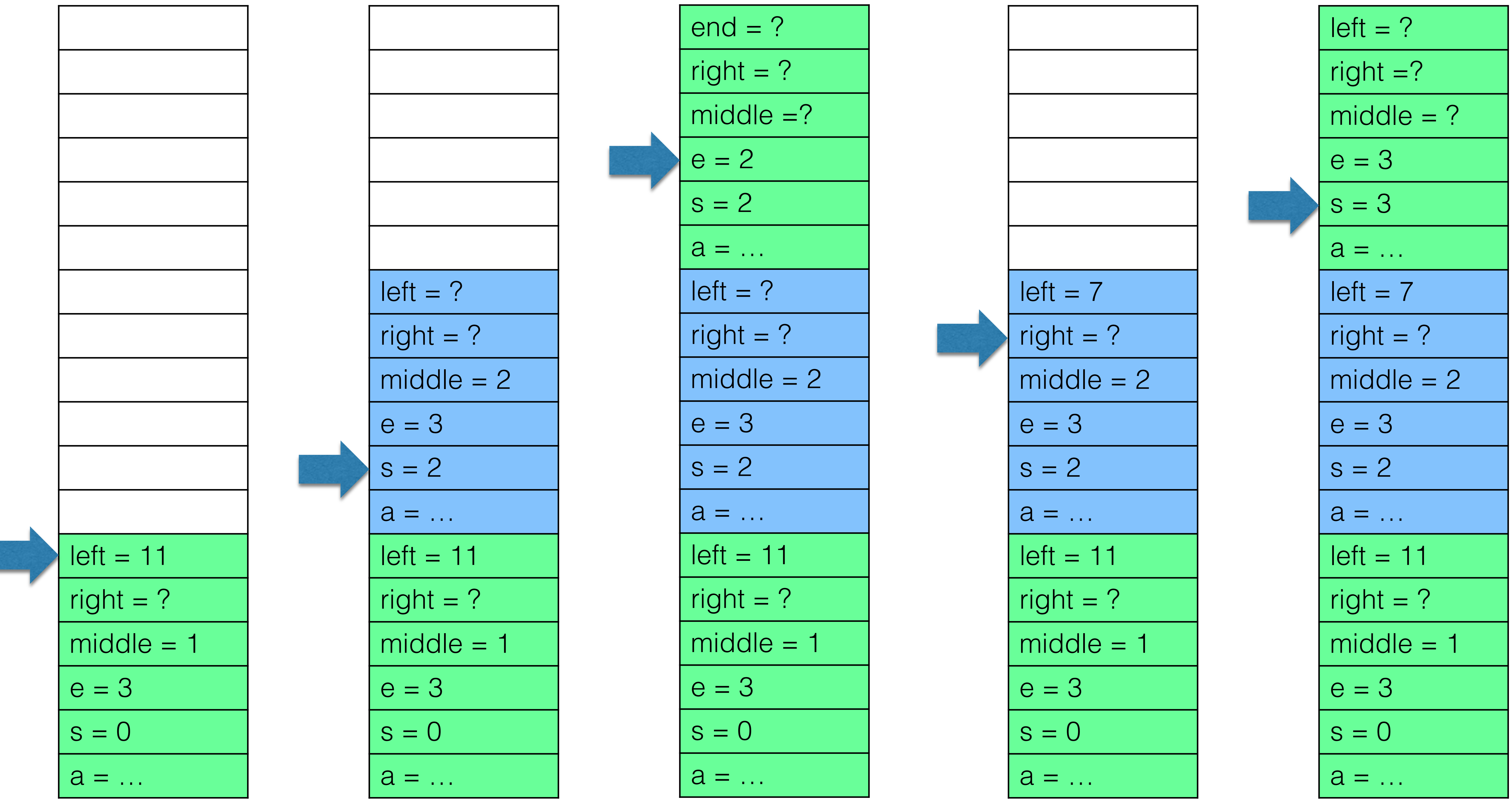
0
0
0
0
0
0
left = ?
right = ?
middle = 0
e = 1
s = 0
a = …
left = ?
right = ?
middle = 1
e = 3
s = 0
a = …

left =?
right = ?
middle = ?
e = 0
s = 0
a = …
left = ?
right = ?
middle = 0
e = 1
s = 0
a = …
left = ?
right = ?
middle = 1
e = 3
s = 0
a = …

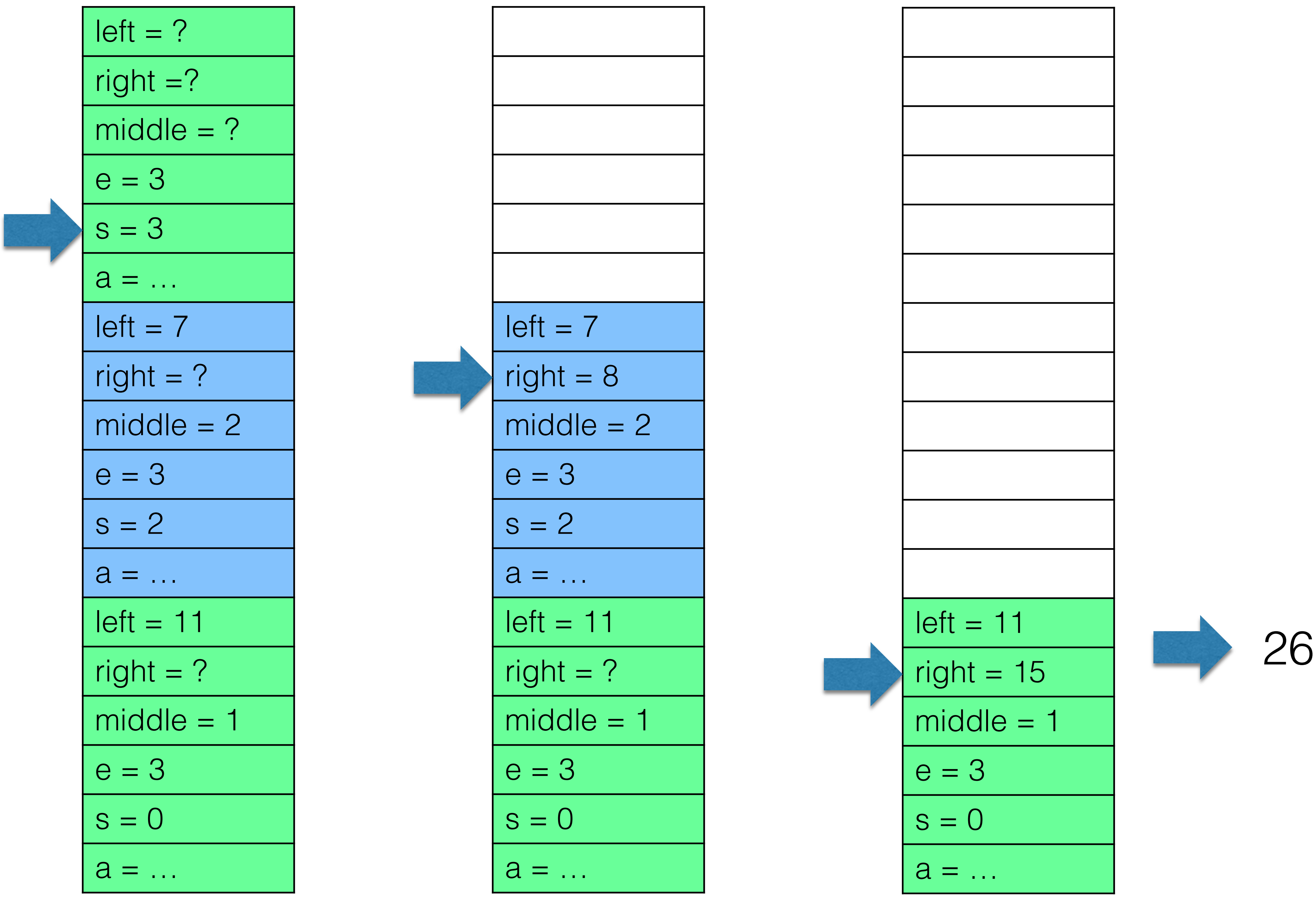
- Before we compute the right half, we wholly compute the left half, including all its internal recursive calls (left and right)



- Once we compute *left=11*, we enter in the call *right=sumY(middle+1,e)*



- Once both *left* and *right* are recursively computed, the final returned result is *11+15 = 26*



Space Complexity

- In these examples, *sumX* had at most 4 frames on stack at same time, whereas *sumY* had only 3
- *space(sumX)* = Θ(n)
  - quite bad… stack is not so big
- *space(sumY)* = Θ(log n)
  - good… but still worse than a Θ(1) of an iterative version using a loop in a single function call
- It can be mathematically proved that, if you halve the space by half at each call, you will get at most *log n* frames on the stack
- When designing a recursive algorithm, should always aim to reduce space by at least half
  - it is not a problem if you then need to make 2 or more recursive calls in the same function

Important to Remember

- 3 main aspects in recursive functions
- Stopping Criterion: otherwise no end, until a stack overflow
- Reduced Input: aim at least at halving it
- Combine Results: once recursive calls finished, combine their outputs together for the final result
    - often it is not as trivial as doing a +

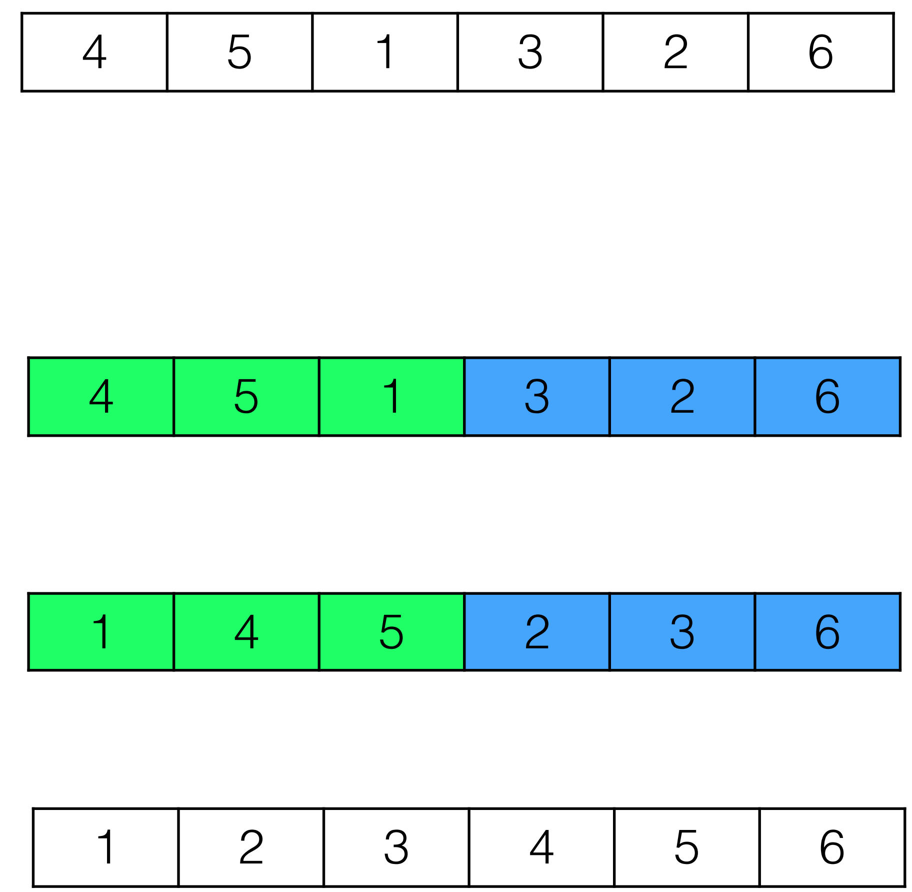
Why Use Recursion???

Recursion vs. Iterative

- Iterative versions of algorithms are "*usually*" better
  - recall that each recursive call has to create and push a new function call frame
- However, there are many algorithms that are *easier* and *more efficient* to write in a recursive form
  - this will become more evident when we will start to work with *Trees*
- Recursive sorting: *Merge Sort* and *Quick Sort*

Merge Sort

- *Divide and Conquer*
  - Recursive implementation
- Split the array in two
- Sort the two parts
- Merge the two parts once sorted



Recursion

- Writing *mergeSort(T[] array)*
- How to sort the 2 halves if my goal was to write a sort algorithm???
- Recursion: reuse function you are writing, but on smaller data

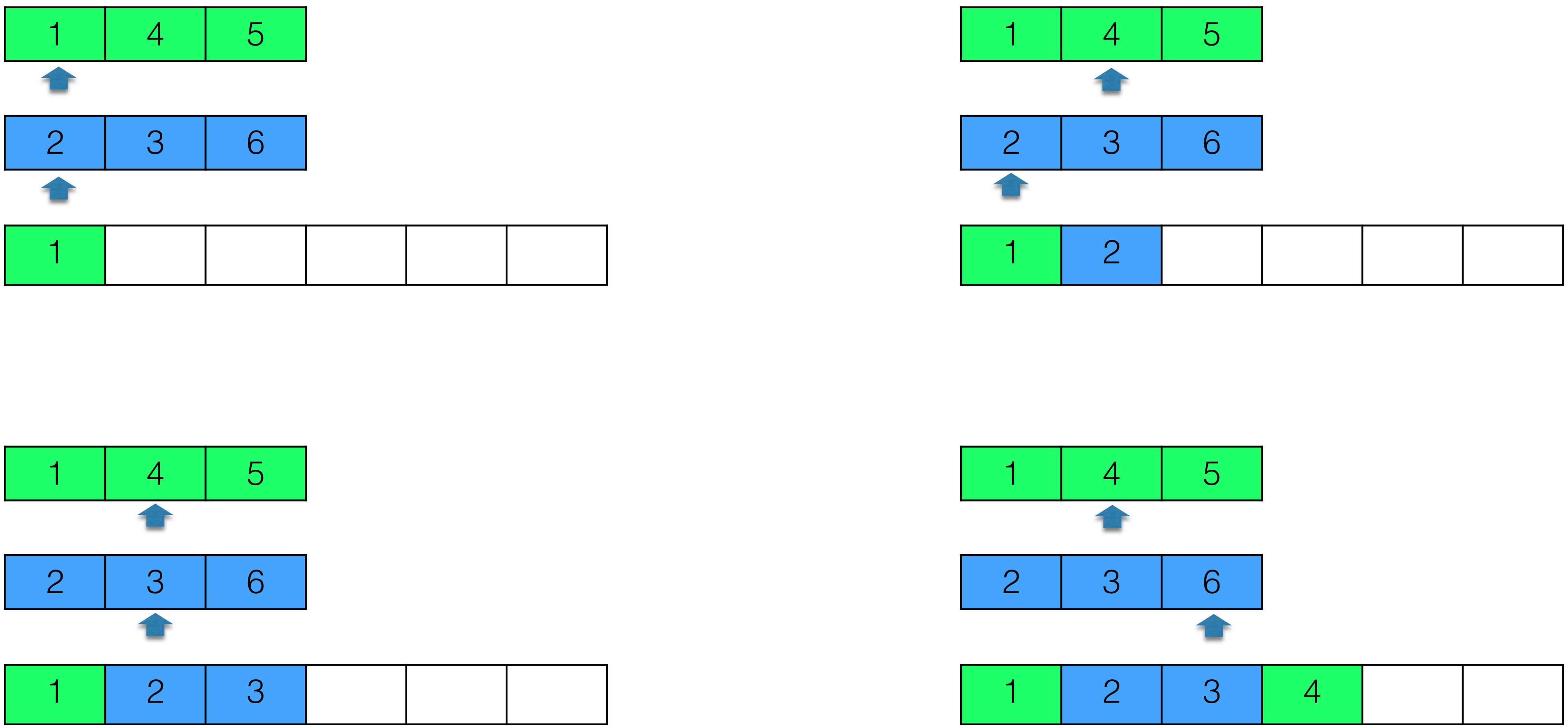
```
mergeSort(T[] array){
    mergeSort(array, 0, array.length / 2)
    mergeSort(array, array.length /2 , array.length)
    mergeHalves(array)
}
```

Recursion End

- To avoid infinite loop, you need to define a stopping condition
- When do I know for certain that an array is sorted?
- Answer: when its size is at most 1
- So, stop recursion when reaching an half of size 1 or less

Merge of Halves

- By recursion, I can sort the 2 halves, but array with 2 sorted halves is not sorted
- Scan the 2 halves, and copy min to a new array
- Between N/2 and N comparisons: once reached end of one half, copy over the other



Cost

- Cost based on 2 recursive calls and then merge
- C(1) = O(1)
- C(n) >= C(n/2) + C(n/2) + n/2
  - Best case for merge, only have to look at one of the halves
  - Eg, all elements in half A are lower than first element in half B
- C(n) <= C(n/2) + C(n/2) + n
  - Worst case for merge, need to look at whole of both halves
- … $C(n) = \Theta(n \log n)$

Considerations

- Asymptotically, does not exist comparison-based sorting better than O(n log n)
- Merge-sort is therefore asymptotically optimal
  - Ie, no instance for which get worse than O(n log n)
- But… more memory, need extra array buffer
- … might be not best on *average*

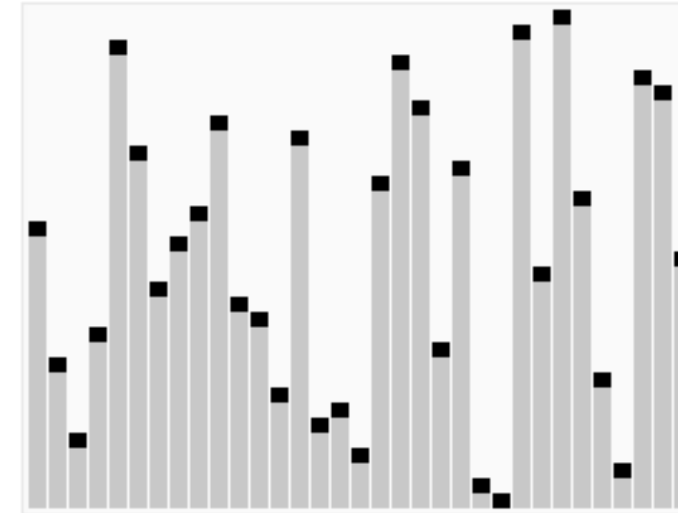
Wikipedia

- Besides book, Wikipedia is good source to read about algorithms and data structures in layman terms
    - https://en.wikipedia.org/wiki/Merge_sort

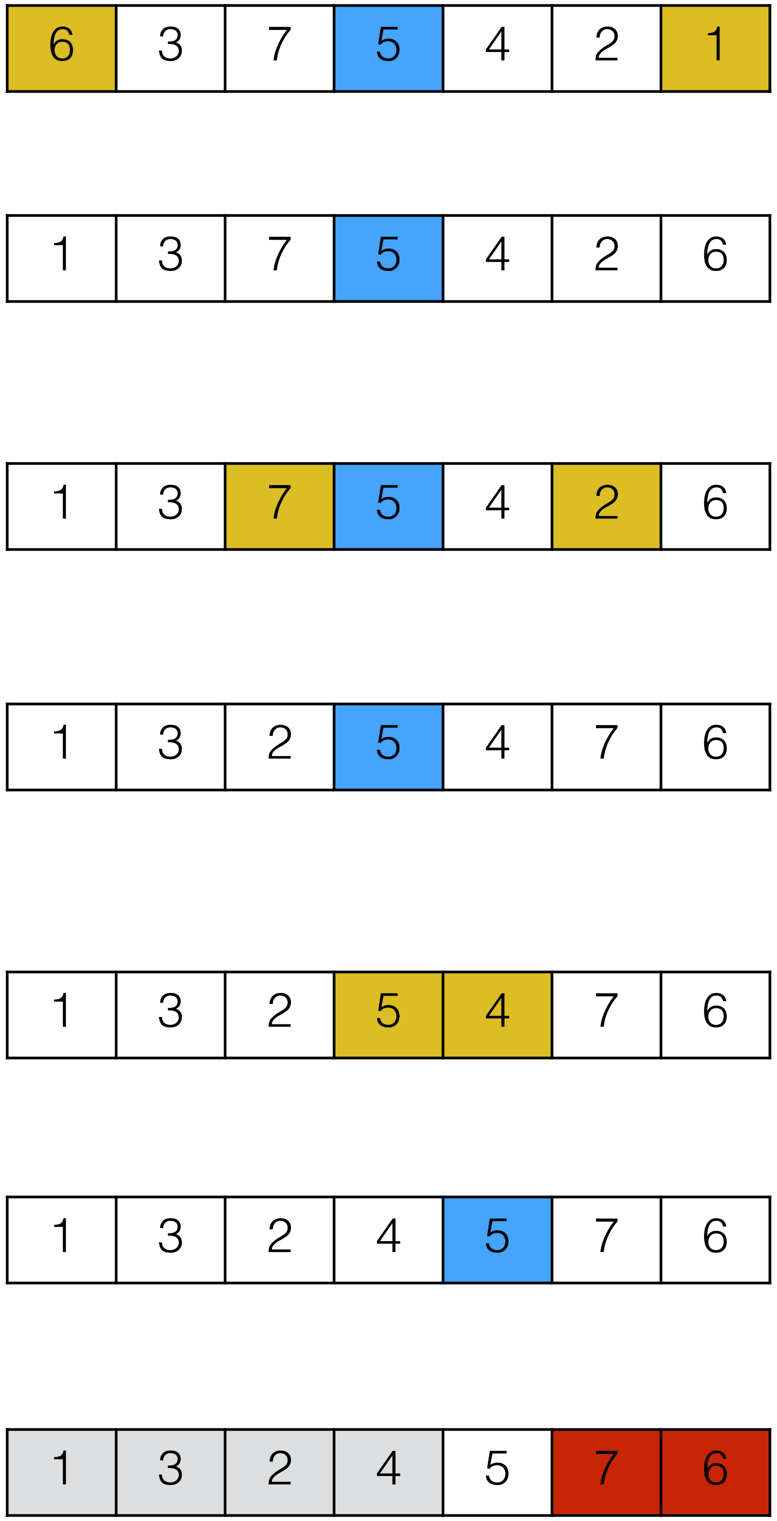
Quick Sort

- One of the most used sorting algorithm
- *Fast on average*, usually “n log n”
  - Better “constant” compared to Merge Sort (eg, no moving data to buffer)
  - But can go till n^2 in worst case
- Minimal memory overhead
- Lot of variants studied during the years

Recursion



- Still Divide and Conquer algorithm, like Merge Sort
- Choose a value X (pivot)
- Move values <X before X, and >X after it
- After one step X is in the correct position
- Apply recursion on subarrays before and after X



- Choose a pivot, eg value 5
- Scan from left till > 5, from right till < 5
- Swap (eg 6 with 1), and continue
- Note how 3 is not touched, as < 5
- At the end, the pivot 5 is in the right position
- On left side, all values < 5
- On right side, all values > 5
- Apply recursively on left and right of pivot

Choice of Pivot

- Choice of pivot is crucial for performance of Quick Sort
  - pivot = array[i]
- For performance, would like a pivot value that gives the 2 partitions of equal size
- How to choose "i"?
  - An option is to take "i" at random
  - Another option is to take middle, which is good when array nearly sorted

1 (and 7) is worst choice

4	3	7	1	5	2	6

4 is best choice

4	3	7	1	5	2	6

Which Sorting Algorithm to Use?

- Unless you have very specific, advanced cases, you use the *default* of what is given by the standard library of the language/framework you use
- Most of the time, it will be a variant of Quick or Merge Sort
  - eg in Java (and Python) the default is TimSort, which is an *hybrid* algorithm based on Merge and Insertion Sort.
    - ★ you can look at code directly at **java.util.TimSort**

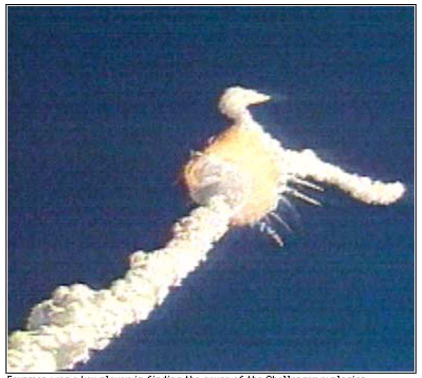
Test Driven Development (TDD)

TDD in One Slide

First write the test cases, which will fail, and then write code to make the test cases pass

What TDD is Not

- It is not a fancy tool you can buy/download
- No cutting edge novel technology
- No silver bullet solving all your software development problems
- *It is just a (relatively simple)* **design** *methodology to improve productivity*



Feynman was a key player in finding the cause of the Challenger explosion 10 years ago (CNN)

Why TDD???

- Bugs in the field can be **extremely** expensive
  - Ariane 5 explosion, Toyota braking system (100k recalled cars), etc
- Aim at improving *quality* (“testing”) within acceptable *cost*
- Lot of *hype* on TDD as possible solution


We received your delivery on time, but we couldn't find the tests
Tests ?
What tests?
You know, the unit tests. The ones for testing the application layer, Quality Assurance and all
Ah, yes, the unit tests...
There aren't any unit tests
No tests ? But what about the guidelines...
How do you check that everything works correctly?
Um...
IT'S MAGIC
CommitStrip.com

Some Facts on TDD

- There isn't much *scientific* evidence that it helps on *large scale* systems
  - Small improvements in *quality*, but also small decrease in *productivity*
- But many anecdotal, success stories
- My *opinion* might be different from what you might hear/read on TDD

The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis

Yahya Rafique and Vojislav B. Mišić, *Senior Member, IEEE*

Abstract—This paper provides a systematic meta-analysis of 27 studies that investigate the impact of Test-Driven Development (TDD) on external code quality and productivity. The results indicate that, in general, TDD has a small positive effect on quality but little to no discernible effect on productivity. However, subgroup analysis has found both the quality improvement and the productivity drop to be much larger in industrial studies in comparison with academic studies. A larger drop of productivity was found in studies where the difference in test effort between the TDD and the control group's process was significant. A larger improvement in quality was also found in the academic studies when the difference in test effort is substantial; however, no conclusion could be derived regarding the industrial studies due to the lack of data. Finally, the influence of developer experience and task size as moderator variables was investigated, and a statistically significant positive correlation was found between task size and the magnitude of the improvement in quality.

Index Terms—Test-driven development, meta-analysis, code quality, programmer productivity, agile software development

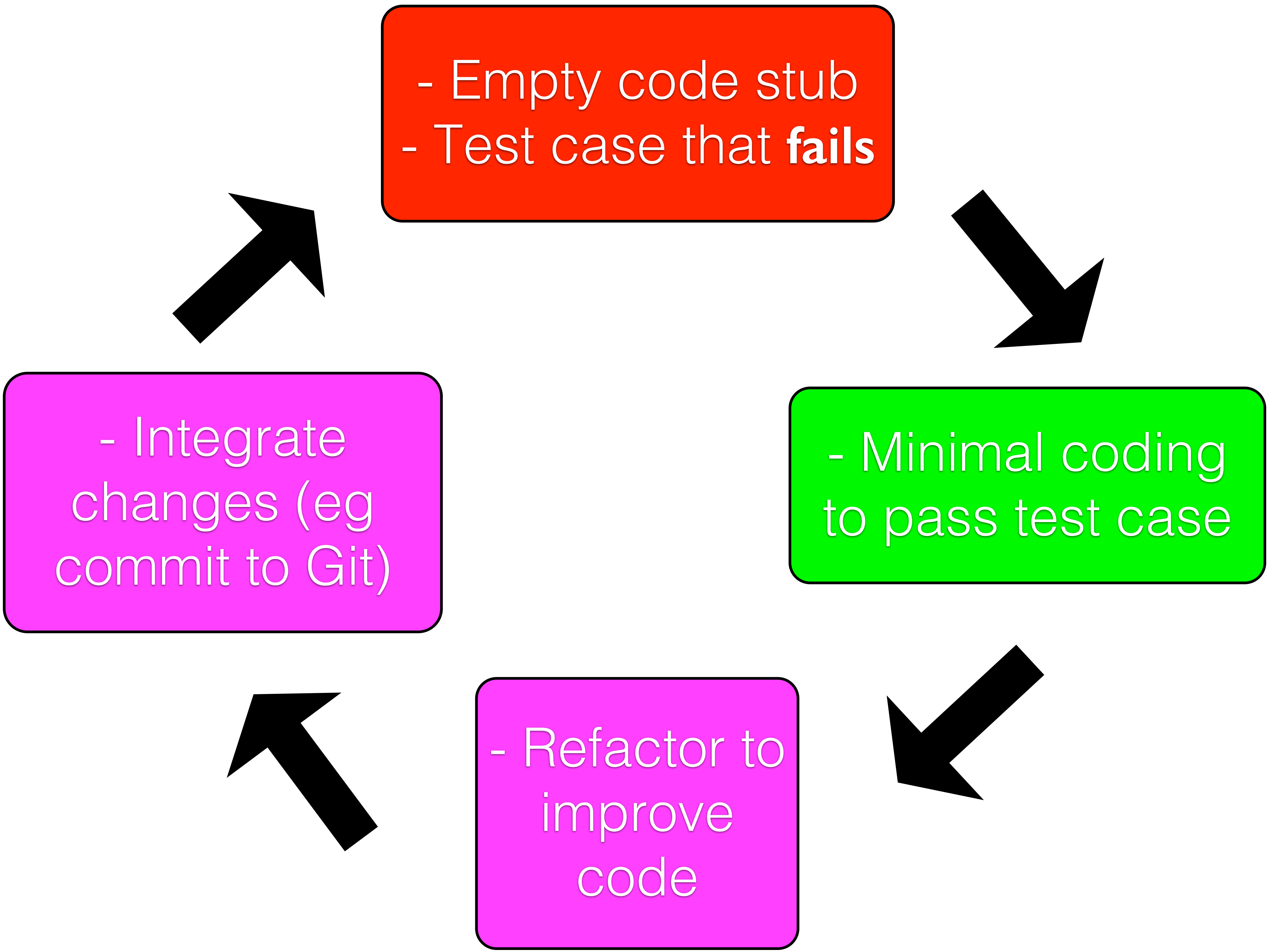
TDD and Algorithms

- Algorithms is a good a place to start with TDD, as having clear functions with clear expected outputs
- Lot of hype on TDD
- However, here we deal with small functions and classes, and not whole applications

Do You Practice What You Preach?

- TDD can be useful, especially for students / junior developers
- But to be honest, I am not using TDD...
  - TDD is a bottom-up approach to software design
- As any technique, its success depends on the context
  - eg, type of software, stage in which TDD is introduced (at the beginning or in a mature project), etc
- Can test software even without TDD
- Success of TDD depends also on *management*

General Principles of TDD


- Empty code stub
- Test case that **fails**
- Minimal coding to pass test case
- Refactor to improve code
- Integrate changes (eg commit to Git)

Example: A List

```
//Start from an "empty" stub that compiles

class List{

    int size(){return -1;  //wrong value}

}
```

First Unit Test

```
void testEmptyList(){
    List list = new List();
    Assert.equals( 0 , list.size());
    // expected empty list
}
```

This will fail!

Fix The Code

```
class List{

    int size(){return 0;}

}
```

Now the test case does not fail

Refactor Step

Improve code

```
class List{

    int counter = 0;

    int size(){return counter;}

}
```

Integration Step

- As all test cases do pass, make changes *permanent*
- Example: commit to repository
  - Git, CVS, SVN, Mercurial, etc.
- Do you really need to committ at each TDD step??? I do not think so...
  - overhead, tedious, side effects if email generated for each commit, etc.

Back to First Step

```
class List{

    int counter = 0;

    void insert(int value){ /* do nothing */}

    int size(){return counter; }

}
```

Write A Second Test

```
void testInsertOneElement(){

    List list = new List();

    list.insert(42);

    Assert.equals( 1 , list.size());

}
```

This will fail!

Minimal Fix

Does not work:

- *testInsertOneElement* does pass
- but *testEmptyList* will fail

```
class List{

    int counter = 1;

    void insert(int value){ }

    int size(){return counter; }

}
```

Still need all passing tests

```
class List{
    int counter = 0;
    void insert(int value){ counter++;}
    int size(){return counter; }
}
```

Refactoring Step

```
class List{
    int[] data = new int[16];
    int counter = 0;
    void insert(int value){ data[counter]=value;
                            counter++;}
    int size(){return counter; }
}
```

Do not do that!
You have no test case checking for internal state!

New Empty Stub

```
class List{

    int counter = 0;

    void insert(int value){ counter++;}

    int size(){return counter; }

    int getLast(){return 0;}

}
```

Write A Third Test

```
void testInsertAndGetOneElement(){

    List list = new List();

    int value = 42;

    list.insert(value);

    Assert.equals( value , list.getLast());

}
```

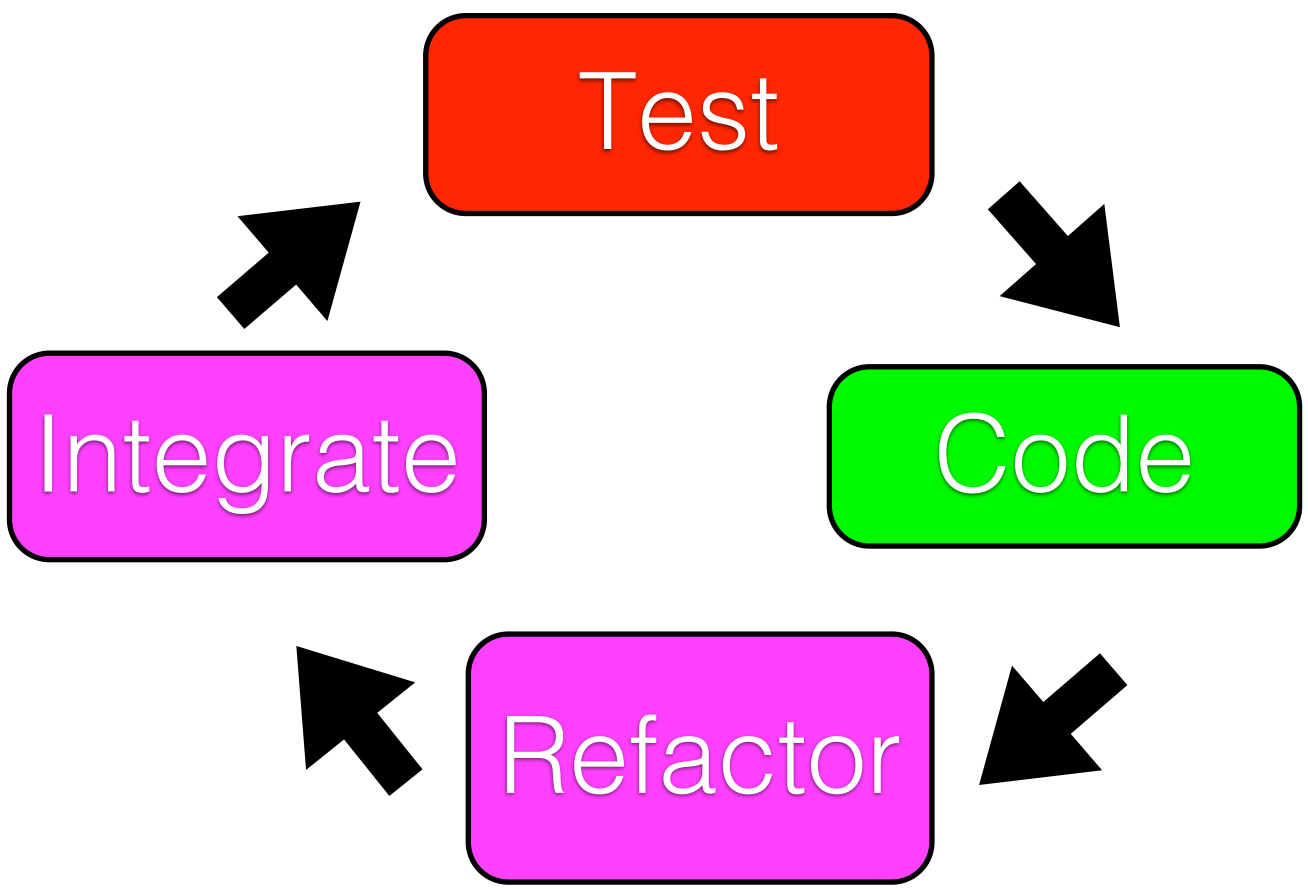
Note: we are **not** testing for “size” here

This will fail!

Minimal Fix Before Refactoring

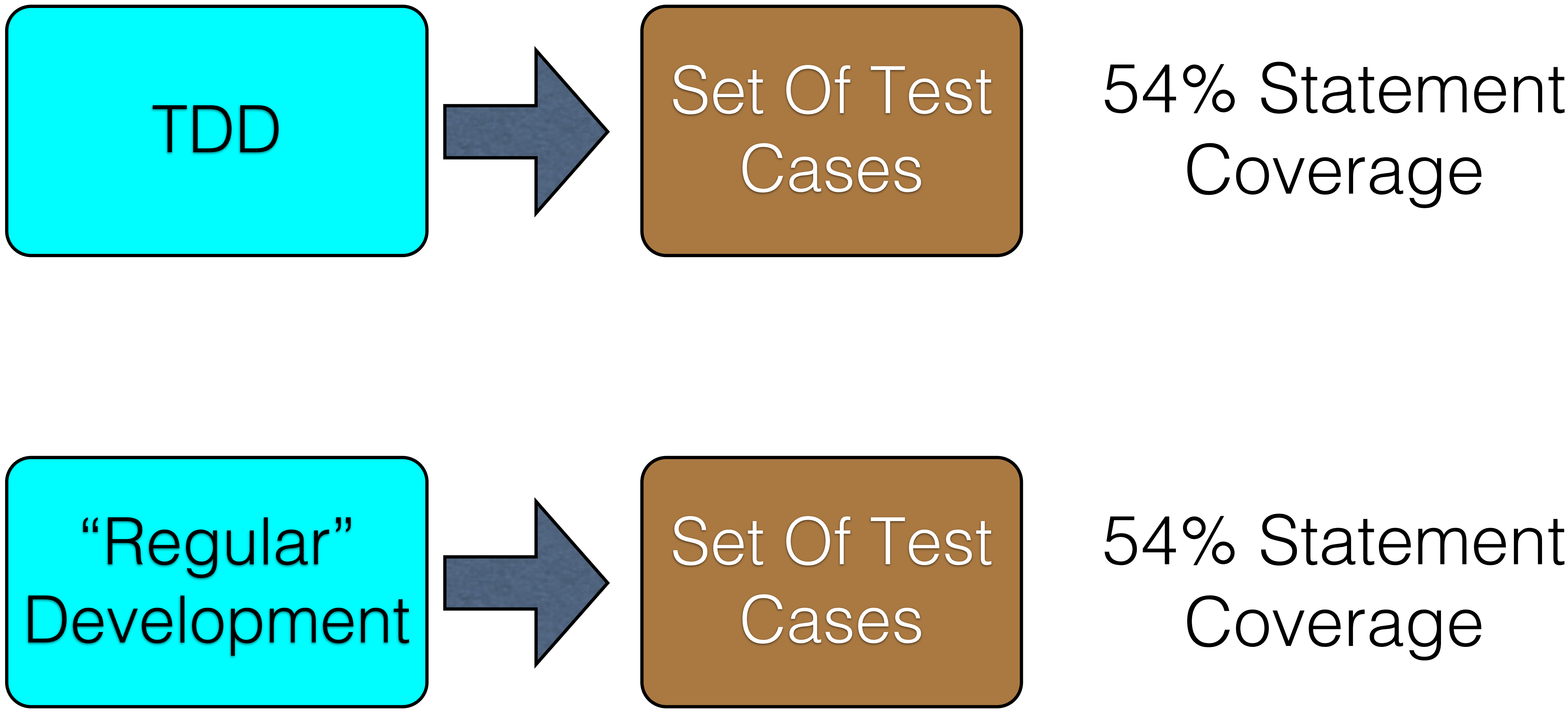
```
class List{

    int counter = 0;

    void insert(int value){ counter++;}

    int size(){return counter; }

    int getLast(){return 42;}

}
```

And so on... keep following the cycle



Measuring Test Effectiveness

- Big issue, regardless of TDD
- #bugs found in QA is a tricky measure
- *Code coverage* is often used


TDD
Set Of Test Cases
54% Statement Coverage
“Regular” Development
Set Of Test Cases
54% Statement Coverage

TDD and Coverage

- TDD is not about more code coverage
- Managers can just set a target (eg >50%) and let engineers decide how to achieve it
- TDD is about *design*

Coverage Does Not Tell You Much

```
void testInsertOneElement(){

    List list = new List();

    list.insert(42);

    int size = list.size();

  // Assert.equals( 1 , size );

}
```

Let's say 15% coverage

Coverage does not change!!!

How Good Are Your Test Assertions?

- Current tools cannot tell you that
  - However, “Mutation Testing” is starting to get usable
- TDD helps you because *force* you to write effective assertions
  - *tests first need to fail*, and then pass

TDD is A **Design** Methodology

- You can write tests even without TDD
- Incremental, *small* steps, each one *verified*
- Can be useful for complex code
- Can apply TDD only on subset of a project
- TDD might lead to write very different code
    - impact on architecture

Homework

- Study Book Chapter 2.2 and 2.3
- Study code in the *org.pg4200.les04* package
- Do exercises in *exercises/ex04*
- Extra: do exercises in the book