Loop Invariants and Binary Search

Chapter 4.3.3 and 9.3.1

Outline

- Iterative Algorithms, Assertions and Proofs of Correctness
- Binary Search: A Case Study

Outline

- **Iterative Algorithms, Assertions and Proofs of Correctness**
- Binary Search: A Case Study

Assertions

- An **assertion** is a statement about the state of the data at a specified point in your algorithm.
- An assertion is not a task for the algorithm to perform.
- You may think of it as a comment that is added for the benefit of the reader.

Loop Invariants

- Binary search can be implemented as an **iterative algorithm** (it could also be done recursively).
- **Loop Invariant:** An **assertion** about the current state useful for designing, analyzing and proving the correctness of iterative algorithms.

Other Examples of Assertions

- Preconditions: Any assumptions that must be true about the input instance.
- Postconditions: The statement of what must be true when the algorithm/program returns.
- Exit condition: The statement of what must be true to exit a loop.

Iterative Algorithms

Take one step at a time

towards the final destination

```
loop (done)
    take step
end loop
```

Establishing Loop Invariant

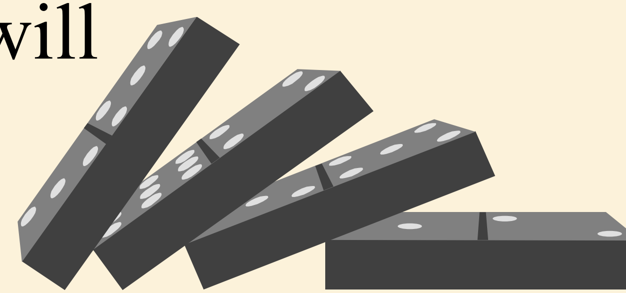
From the Pre-Conditions on the input instance we must establish the loop invariant.

Maintain Loop Invariant

- Suppose that
  - We start in a safe location (pre-condition)
  - If we are in a safe location, we always step to another safe location (loop invariant)
- Can we be assured that the computation will always be in a safe location?
- By what principle?

Maintain Loop Invariant

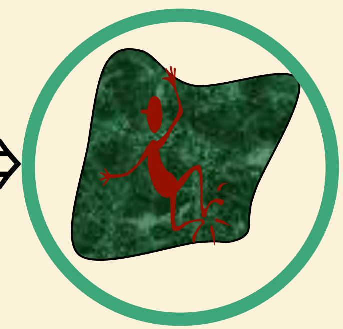
- By Induction the computation will always be in a safe location.



$\Rightarrow S(0)$

$\Rightarrow \forall i, S(i) \Rightarrow S(i+1)$

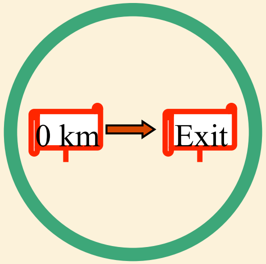
$$\left.\begin{matrix} \Rightarrow S(0) \\ \Rightarrow \forall i, S(i) \Rightarrow S(i+1) \end{matrix}\right\} \Rightarrow \forall i, S(i) \Rightarrow$$



Ending The Algorithm

- Define Exit Condition

- Termination: With sufficient progress, the exit condition will be met.



- When we exit, we know
  - exit condition is true
  - loop invariant is true

  from these we must establish the post conditions.

Definition of Correctness

<PreCond> & <code> → <PostCond>

If the input meets the preconditions,

then the output must meet the postconditions.

If the input does not meet the preconditions, then nothing is required.

Outline

- Iterative Algorithms, Assertions and Proofs of Correctness
- **Binary Search: A Case Study**

Define Problem: Binary Search

- PreConditions
  - Key 25
  - Sorted List

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- PostConditions
  - Find key in list (if there).

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Loop Invariant

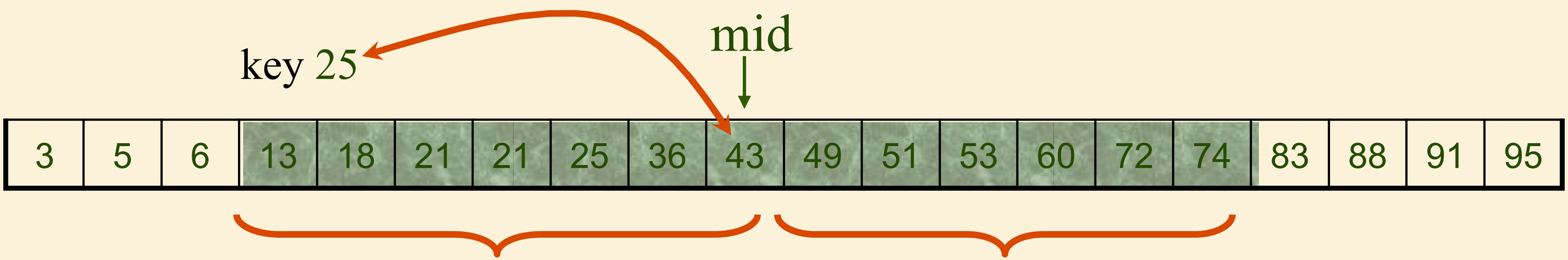
- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

key 25

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Step

- Cut sublist in half.
- Determine which half the key would be in.
- Keep that half.

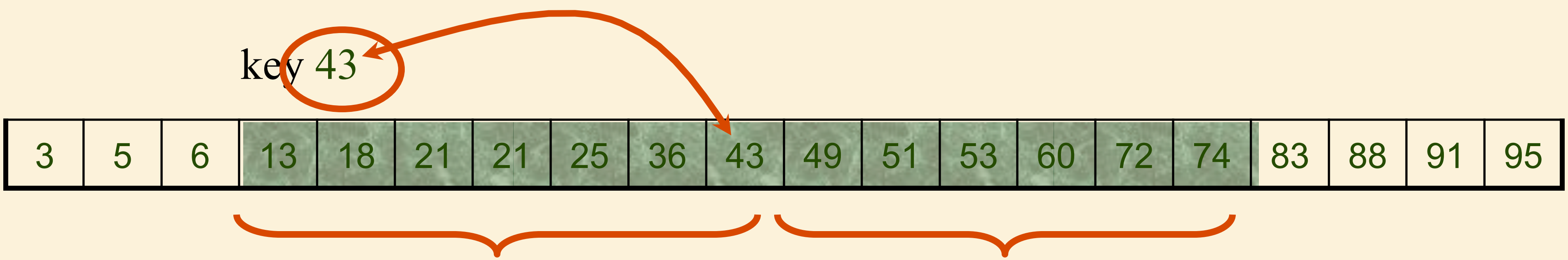


If key $\le$ mid, then key is in left half.

If key $>$ mid, then key is in right half.

Define Step

- It is faster not to check if the middle element is the key.
- Simply continue.

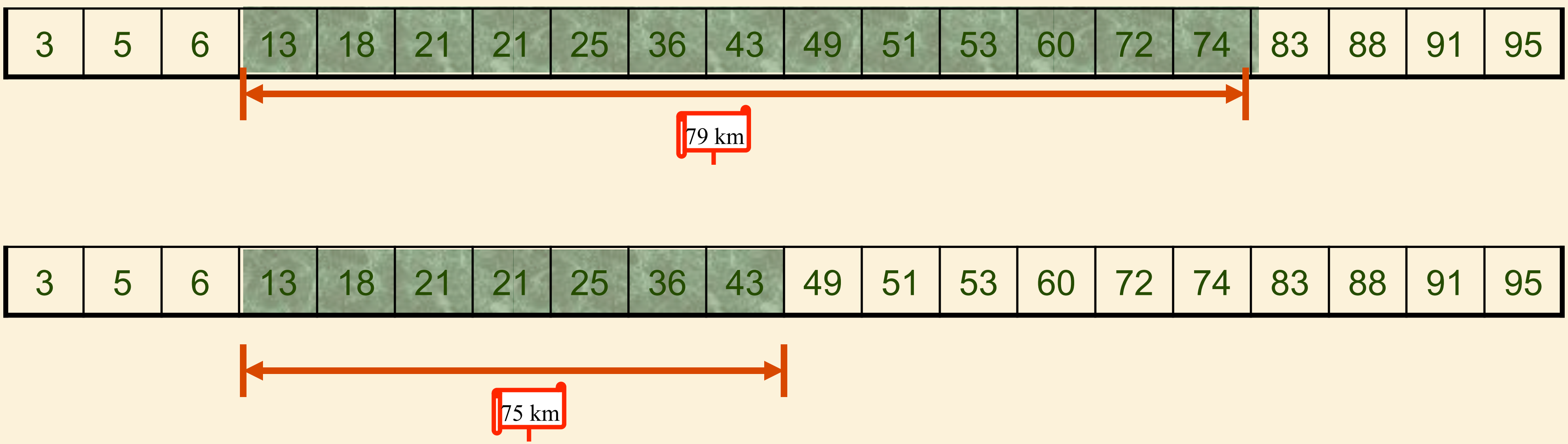


If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

Make Progress

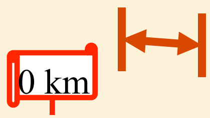
- The size of the list becomes smaller.



Exit Condition

key 25

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95



- If the key is contained in the original list,

  then the key is contained in the sublist.

- Sublist contains one element.

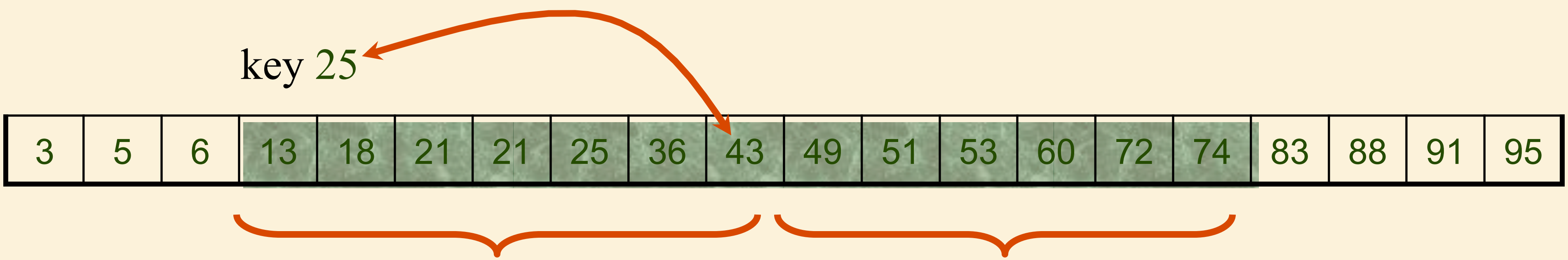
- If element = key, return associated entry.
- Otherwise return false.

Running Time

The sublist is of size n, $^n/_2$, $^n/_4$, $^n/_8$,…,1

Each step O(1) time.

Total = O(log n)



If key ≤ mid,
then key is in
left half.

If key > mid,
then key is in
right half.

Running Time

- Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature.
- It is common to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers.

END OF LECTURE
FEB 13, 2014

BinarySearch(A[1..n], *key*)

<precondition>: A[1..n] is sorted in non-decreasing order

<postcondition>: If *key* is in A[1..n], algorithm returns its location

$p = 1,\ q = n$

while $q > p$

    < loop-invariant>: If *key* is in A[1..n], then *key* is in A[p..q]

    $mid = \left\lfloor \frac{p+q}{2} \right\rfloor$

    if $key \le A[mid]$

        $q = mid$

    else

        $p = mid + 1$

    end

end

if $key = A[p]$

    return(p)

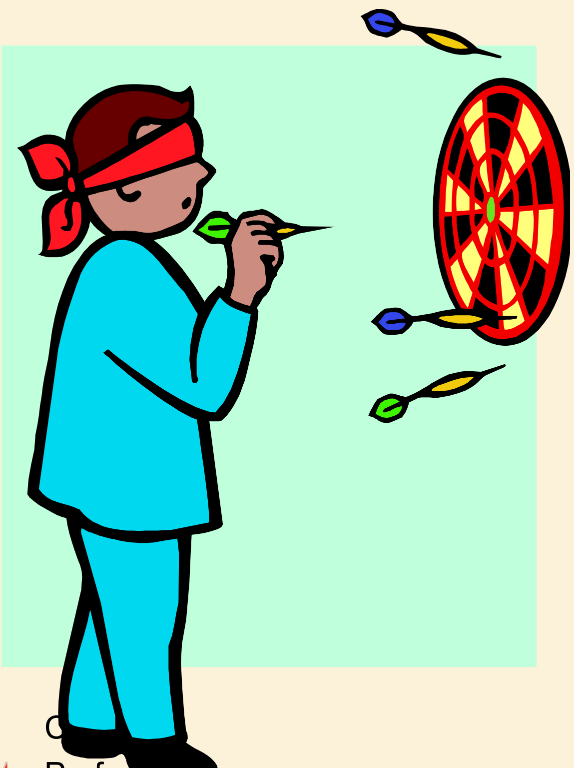
else

    return("Key not in list")

end

Simple, right?

- Although the concept is simple, binary search is notoriously easy to get wrong.
- Why is this?



Boundary Conditions

- The basic idea behind binary search is easy to grasp.
- It is then easy to write pseudocode that works for a ‘typical’ case.
- Unfortunately, it is equally easy to write pseudocode that fails on the *boundary conditions*.

Boundary Conditions

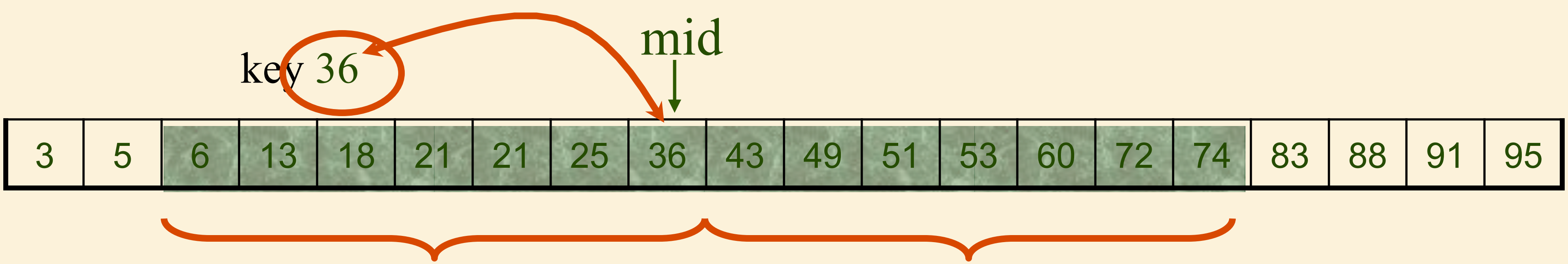
$$\text{if } key \le A[mid]$$
$$\quad q = mid$$
$$\text{else}$$
$$\quad p = mid + 1$$
$$\text{end}$$

or

~~$\text{if } key < A[mid]$~~
~~$\quad q = mid$~~
~~else~~
~~$\quad p = mid + 1$~~
~~end~~

What condition will break the loop invariant?

Boundary Conditions



Code: key ≥ A[mid] → select right half

Bug!!

Boundary Conditions

$$\begin{aligned}&\text{if } key \le A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid + 1\\&\text{end}\end{aligned}$$

OK

$$\begin{aligned}&\text{if } key < A[mid]\\&\quad q = mid - 1\\&\text{else}\\&\quad p = mid\\&\text{end}\end{aligned}$$

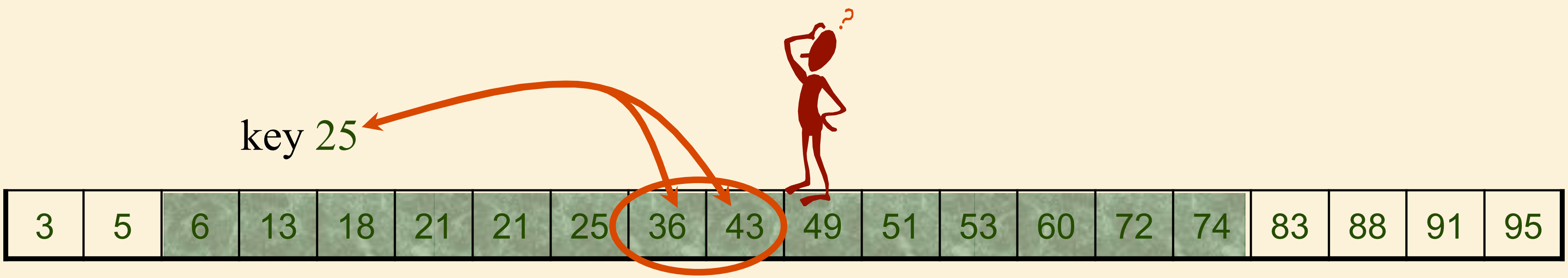
OK

$$\begin{aligned}&\text{if } key < A[mid]\\&\quad q = mid\\&\text{else}\\&\quad p = mid + 1\\&\text{end}\end{aligned}$$

Not OK!!

Boundary Conditions

$$\text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor \quad \text{or} \quad \text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$

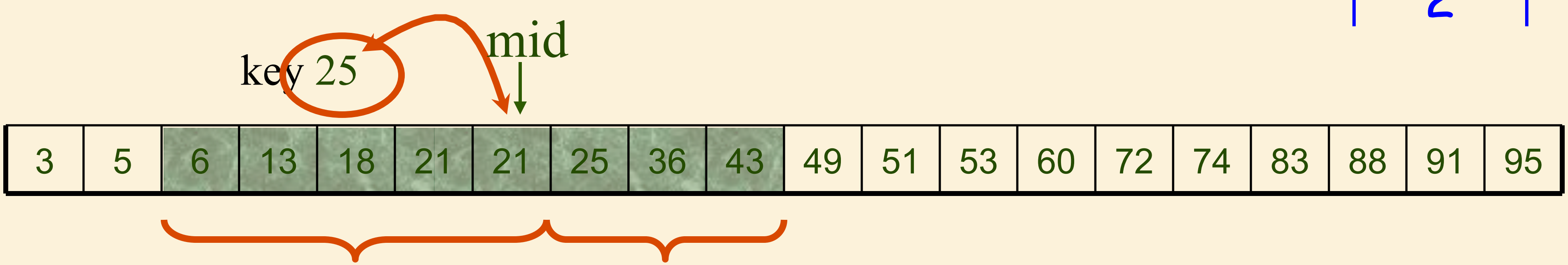


Shouldn't matter, right?

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$

Boundary Conditions

Select $mid = \left\lceil \frac{p+q}{2} \right\rceil$

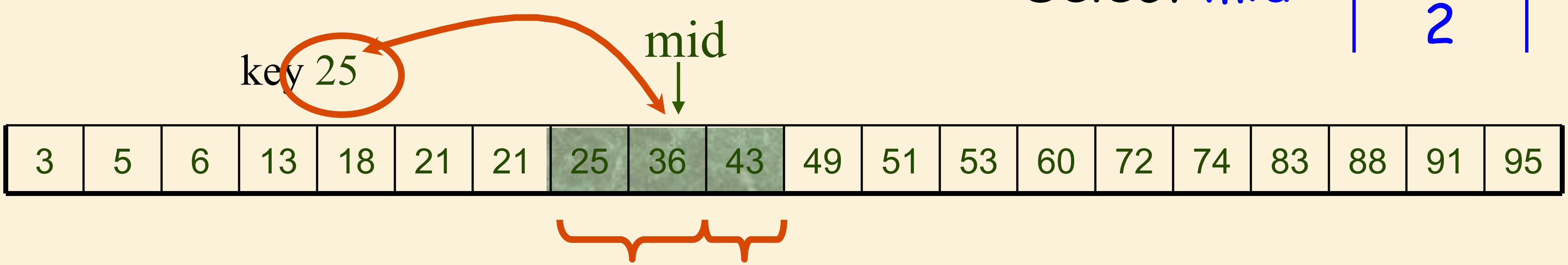


If key $\leq$ mid,
then key is in
left half.

If key $>$ mid,
then key is in
right half.

Boundary Conditions

Select $mid = \left\lceil \frac{p+q}{2} \right\rceil$



If key $\leq$ mid, then key is in left half.

If key $>$ mid, then key is in right half.

Boundary Conditions

No progress toward goal: Loops Forever!

- Another bug!

Select $\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$

key 25

mid

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

If key ≤ mid, then key is in left half.

If key > mid, then key is in right half.

Boundary Conditions

$$\text{mid} = \left\lfloor \frac{p+q}{2} \right\rfloor$$
if $key \le A[mid]$
    $q = mid$
else
    $p = mid + 1$
end

OK

$$\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$
if $key < A[mid]$
    $q = mid - 1$
else
    $p = mid$
end

OK

$$\text{mid} = \left\lceil \frac{p+q}{2} \right\rceil$$
if $key \le A[mid]$
    $q = mid$
else
    $p = mid + 1$
end

Not OK!!

Getting it Right

- How many possible algorithms?
- How many **correct** algorithms?
- Probability of **guessing** correctly?

$$mid = \left\lfloor \frac{p+q}{2} \right\rfloor$$

or $mid = \left\lceil \frac{p+q}{2} \right\rceil$?

if $key \le A[mid]$

$q = mid$

else

$p = mid + 1$

end

or if $key < A[mid]$?

or $q = mid - 1$

else

$p = mid$

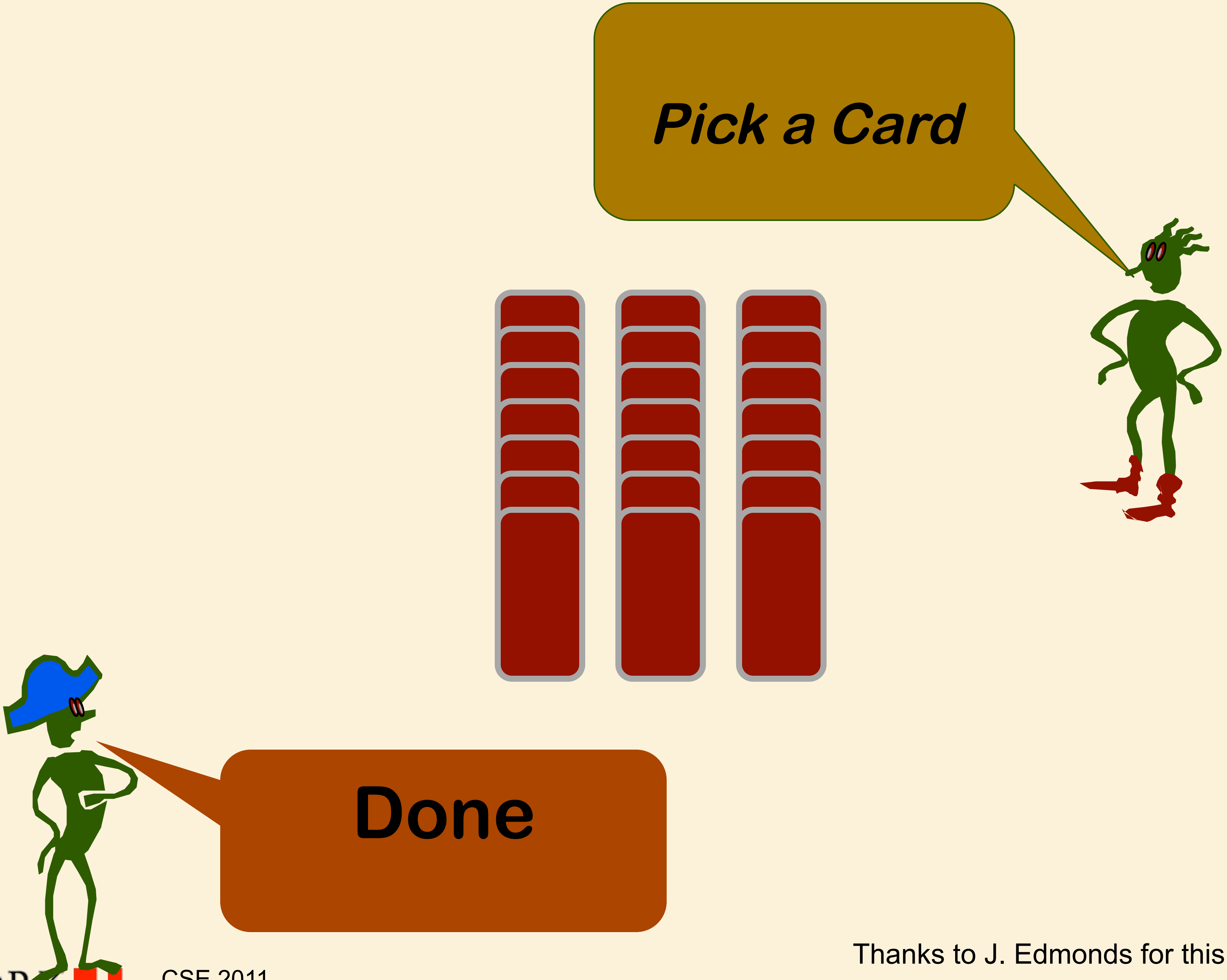
end

Alternative Algorithm: Less Efficient but More Clear

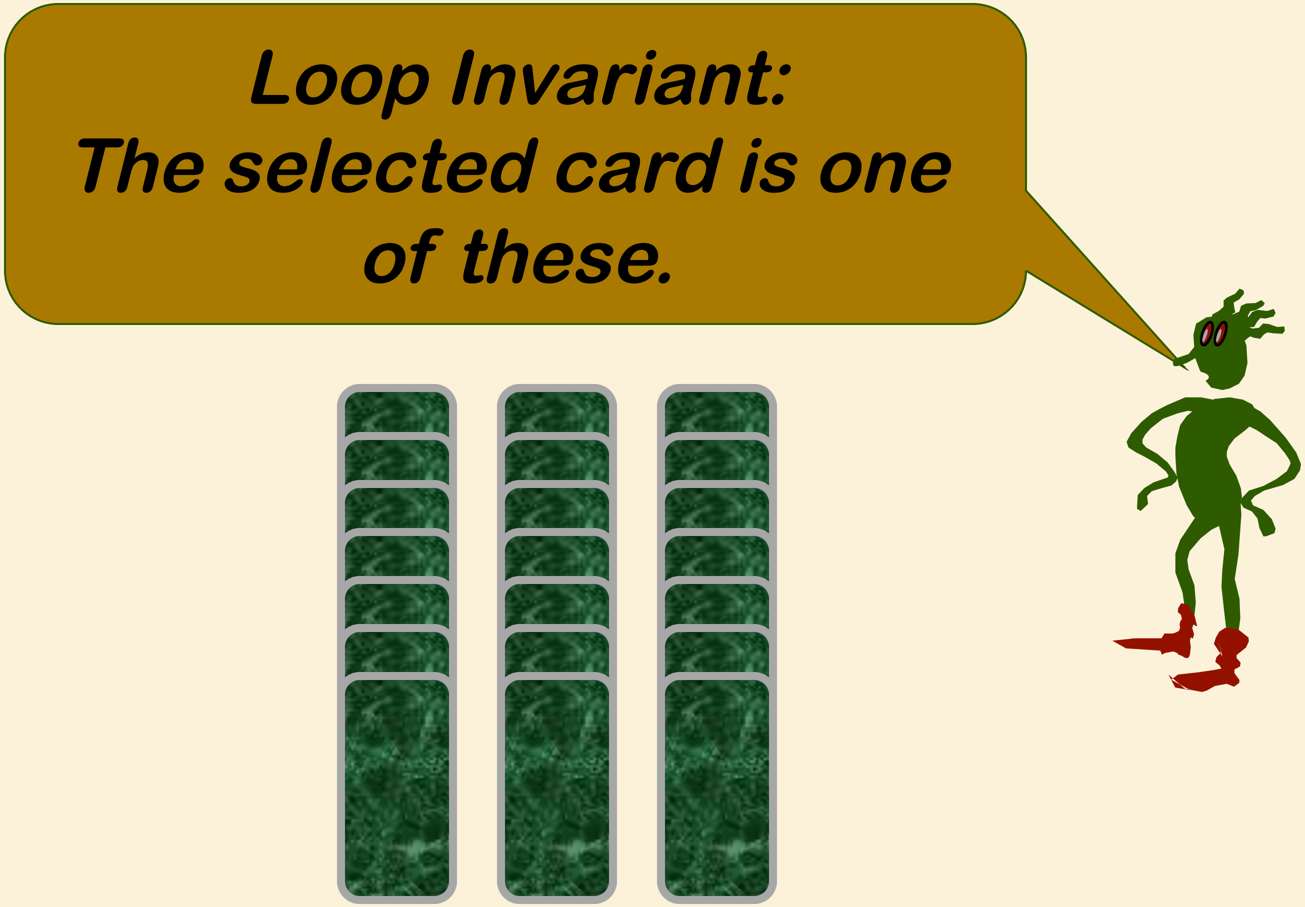
```
BinarySearch(A[1..n], key)
<precondition>:  A[1..n] is sorted in non-decreasing order
<postcondition>: If key is in A[1..n], algorithm returns its location
p = 1, q = n
while q ≥ p
    < loop-invariant>: If key is in A[1..n], then key is in A[p..q]
    mid = ⌊(p+q)/2⌋
    if key < A[mid]
        q = mid − 1
    else if key > A[mid]
        p = mid + 1
    else
        return(mid)
    end
end
return("Key not in list")
```

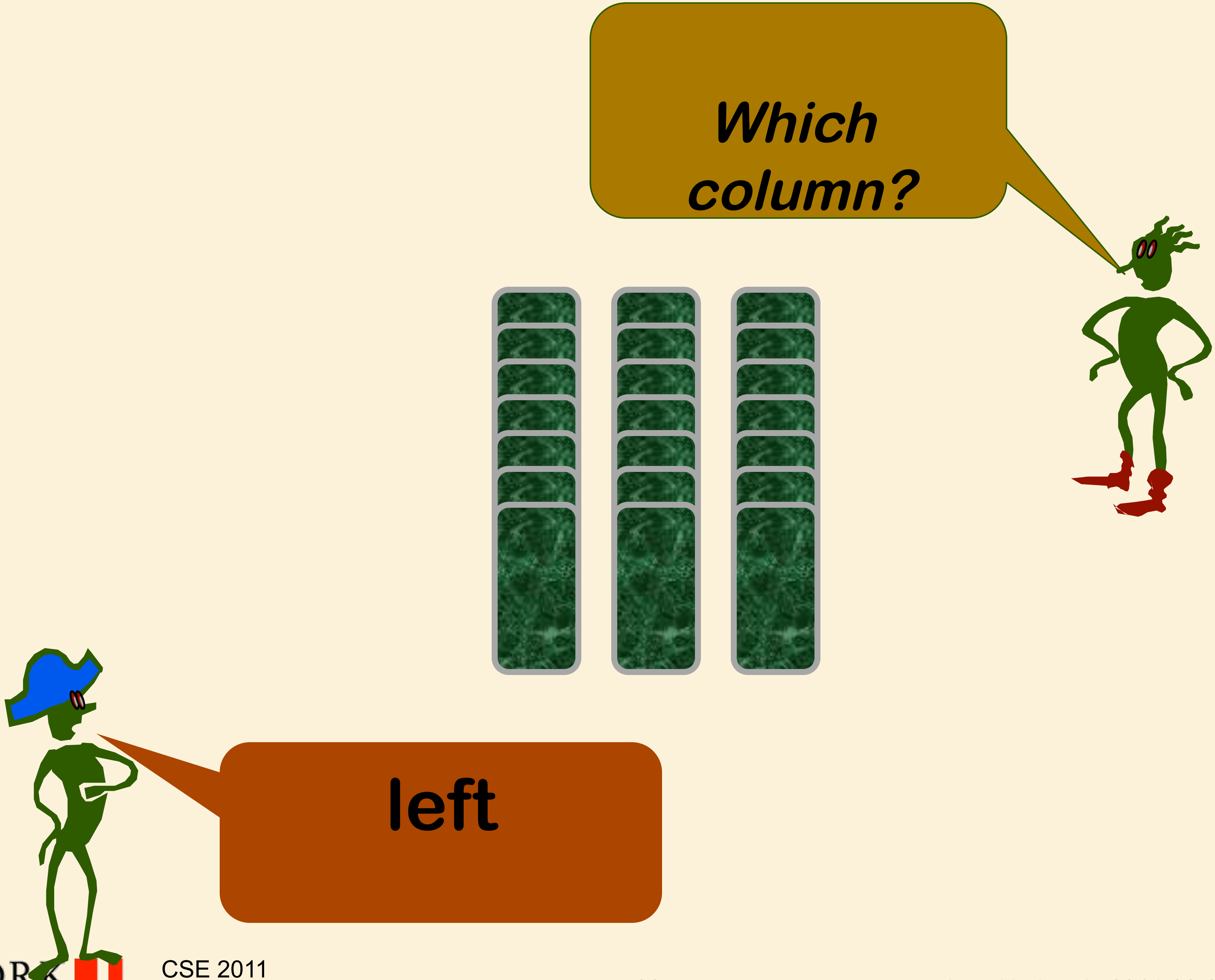
Still $\Theta(\log n)$, but with slightly larger constant.

Card Trick

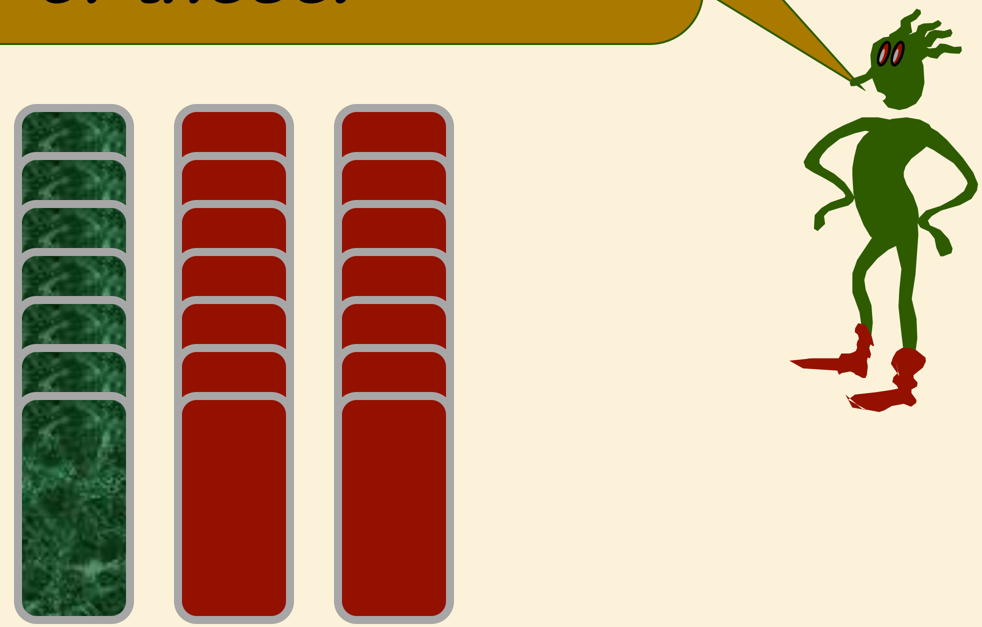


Thanks to J. Edmonds for this example.

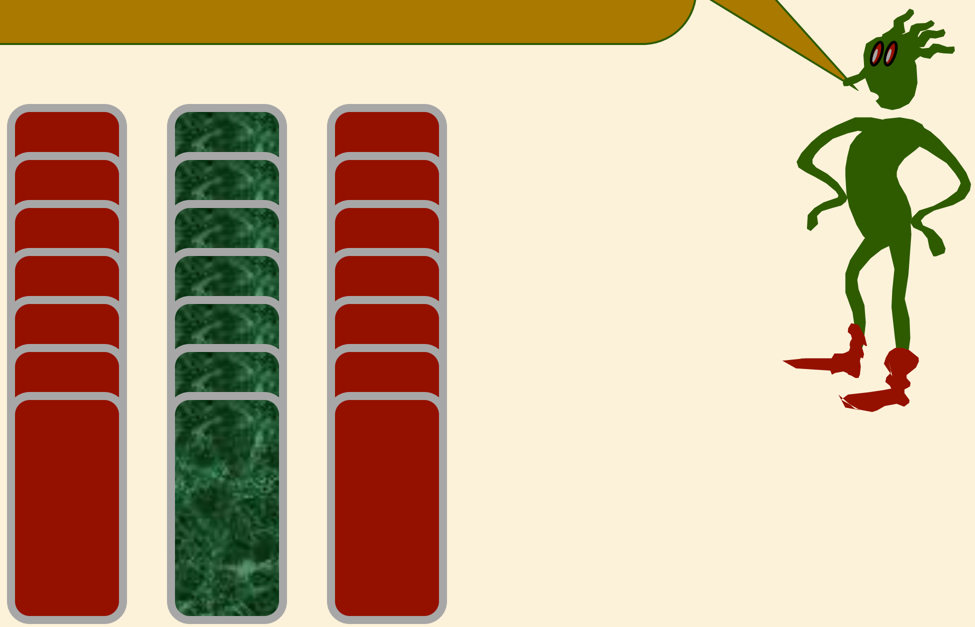

Loop Invariant:
The selected card is one of these.

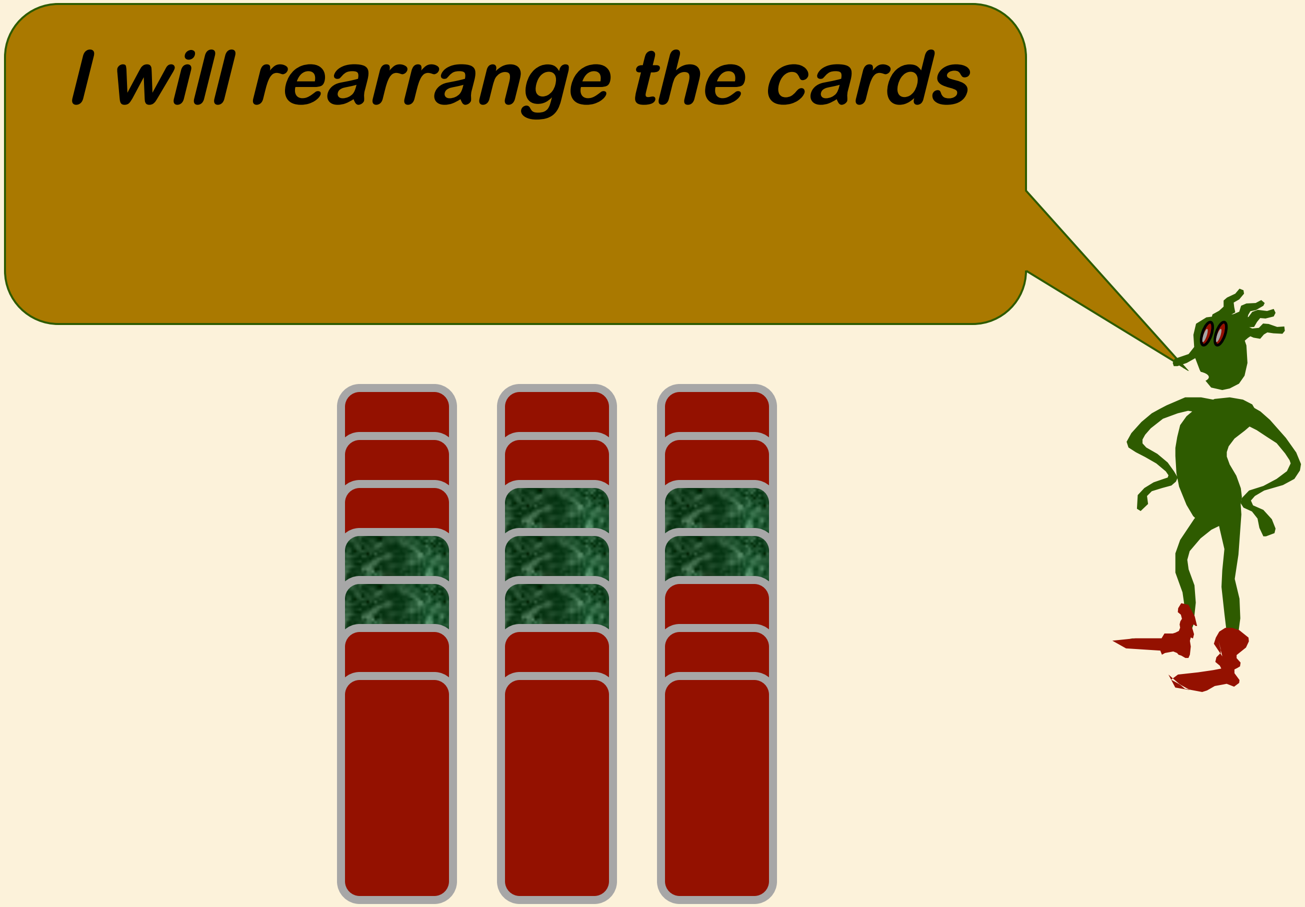

Which column?
left

Loop Invariant: The selected card is one of these.

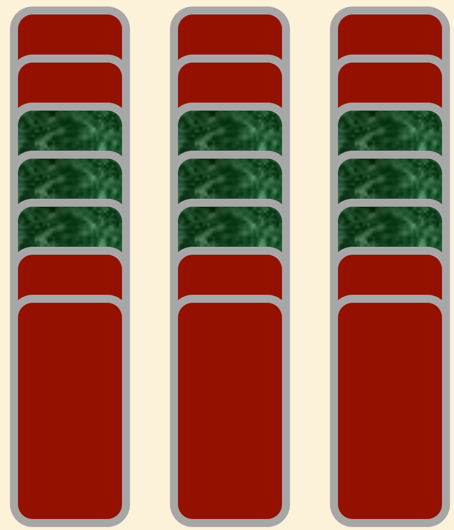


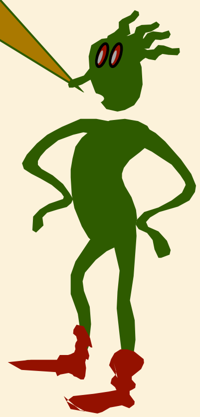
Selected column is placed in the middle

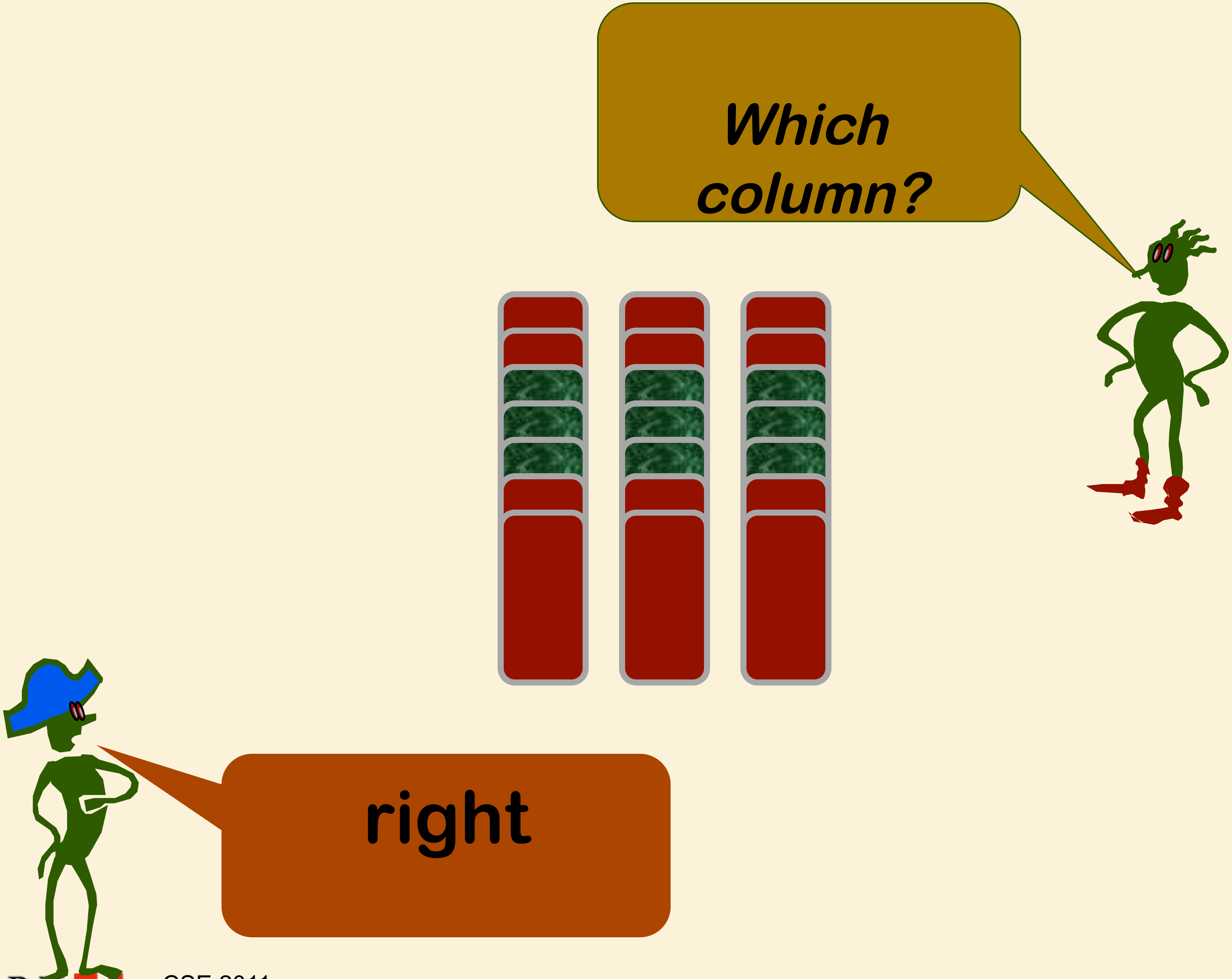


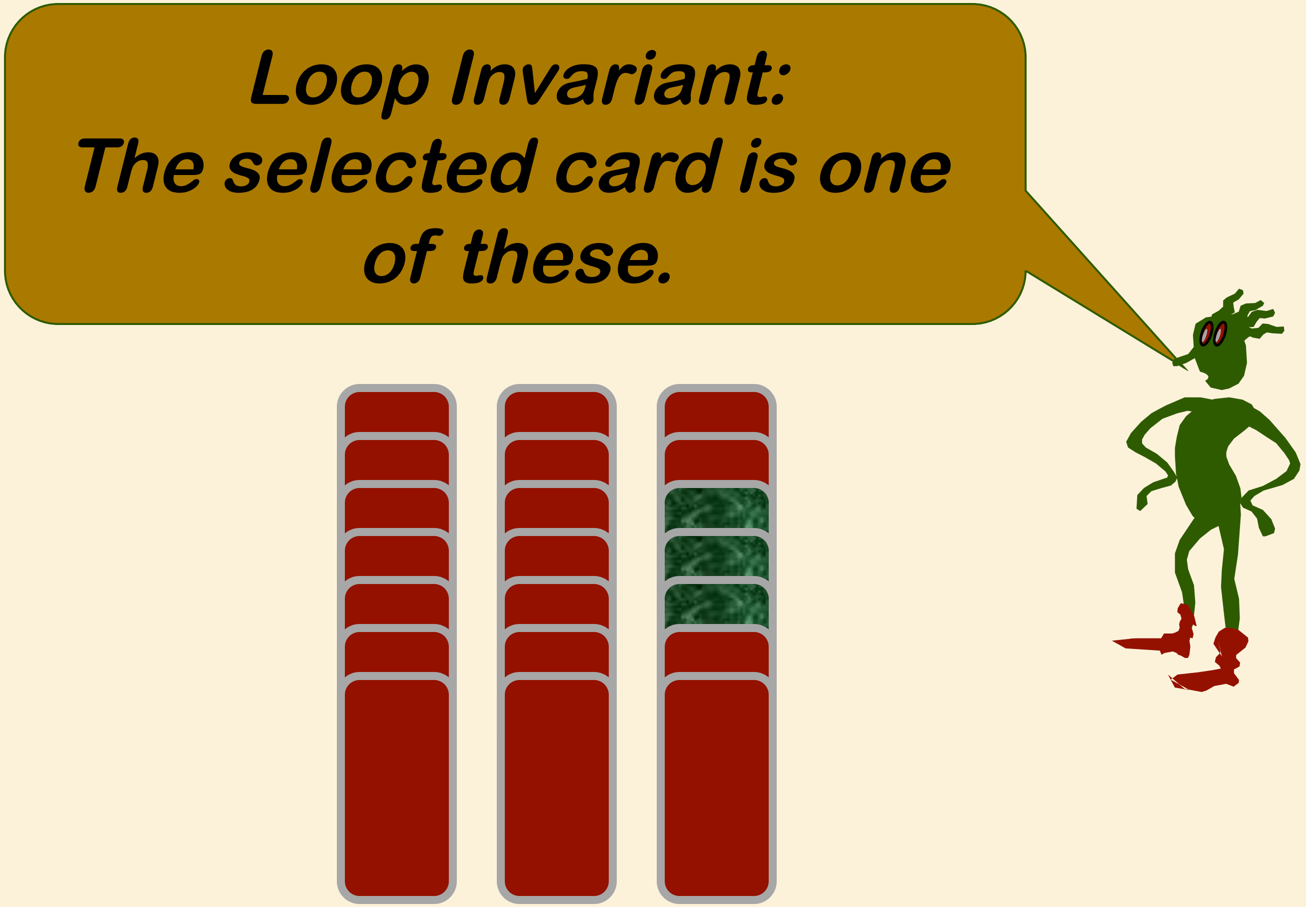

I will rearrange the cards

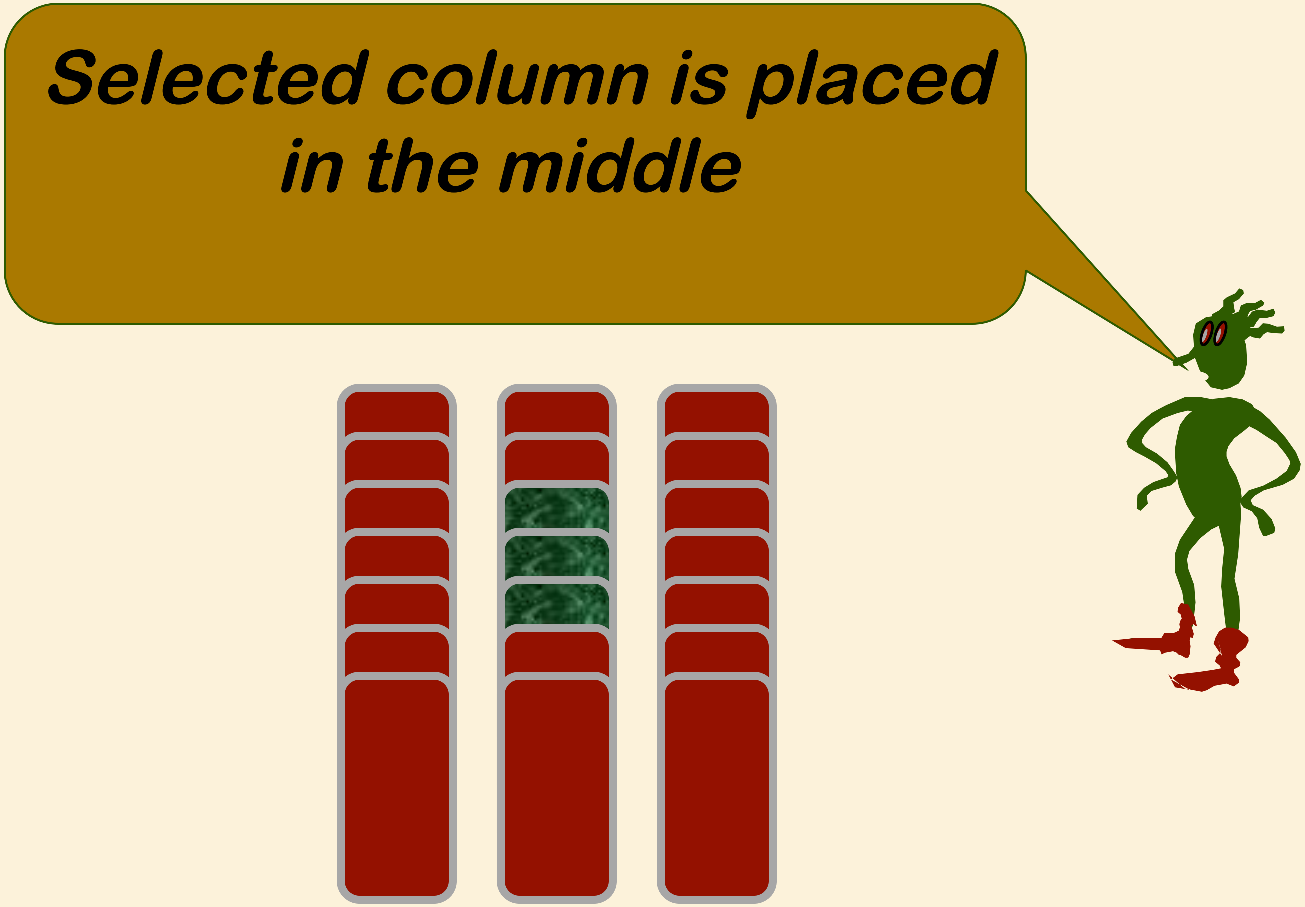
Relax Loop Invariant:
I will remember the same about each column.



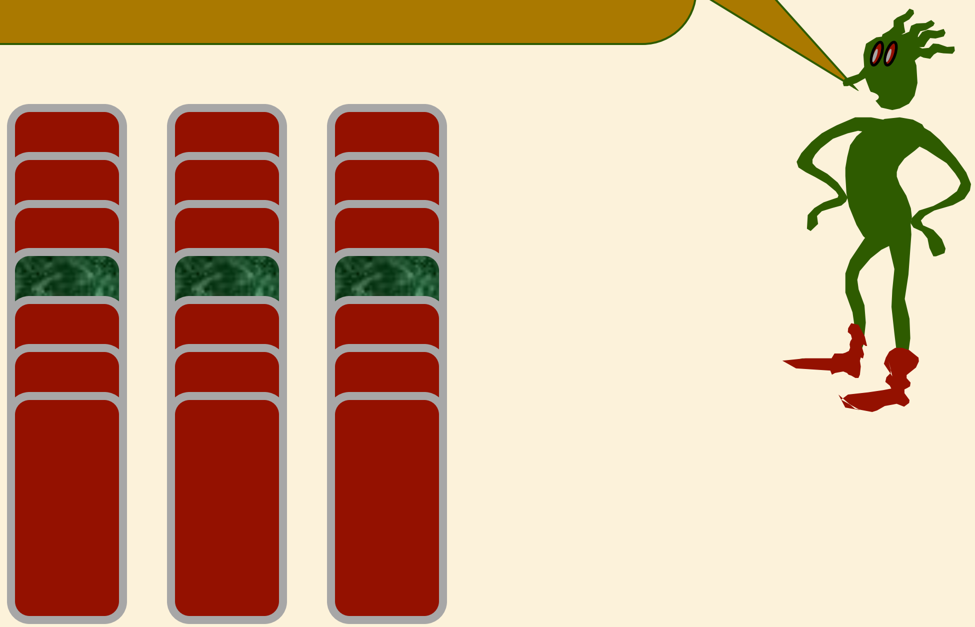


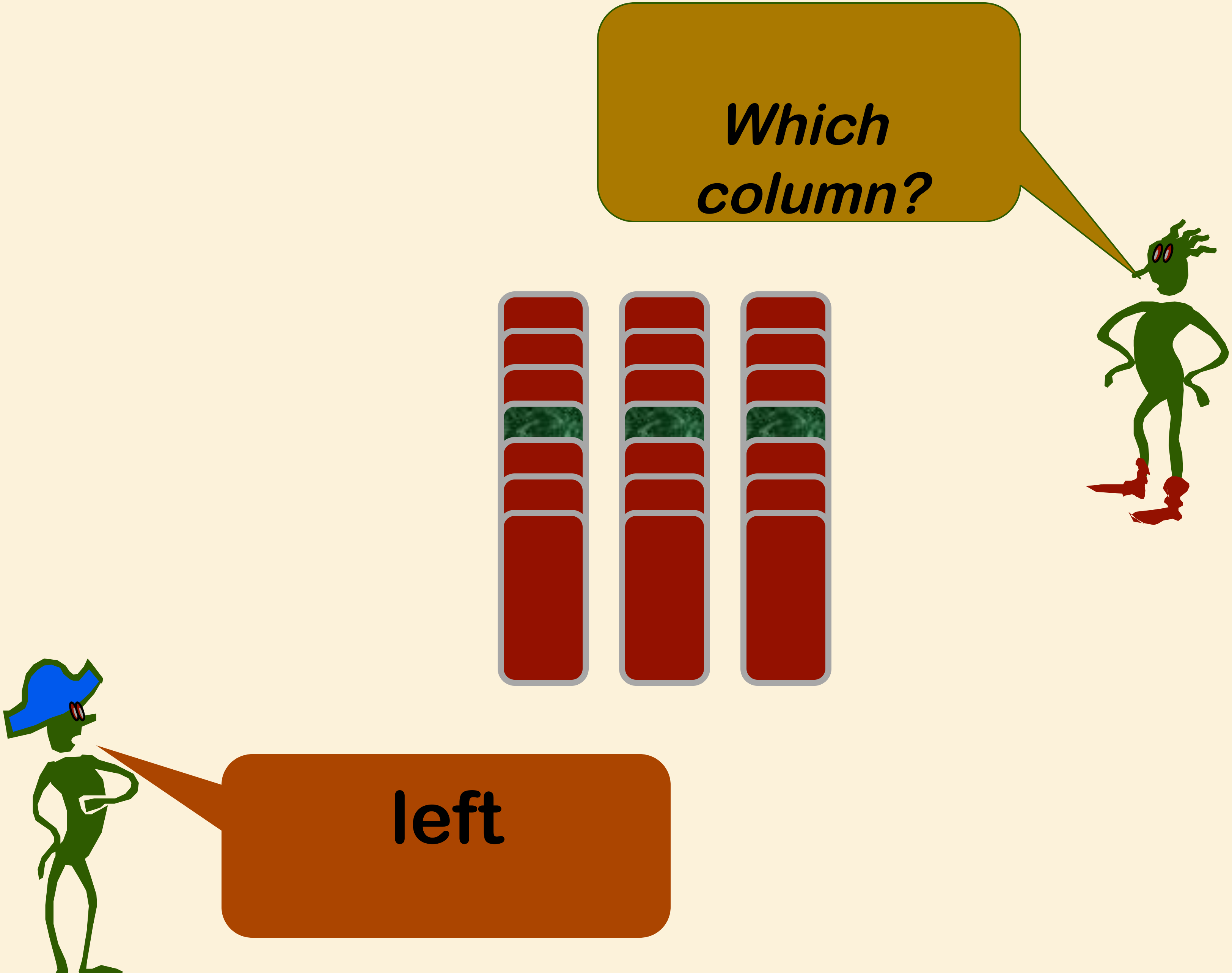

Which column?
right


Loop Invariant:
The selected card is one of these.

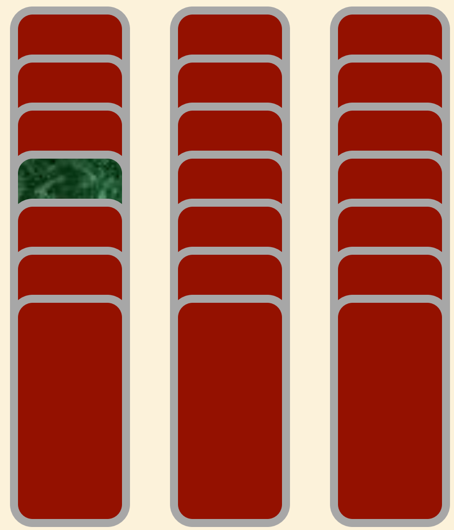

Selected column is placed in the middle

I will rearrange the cards

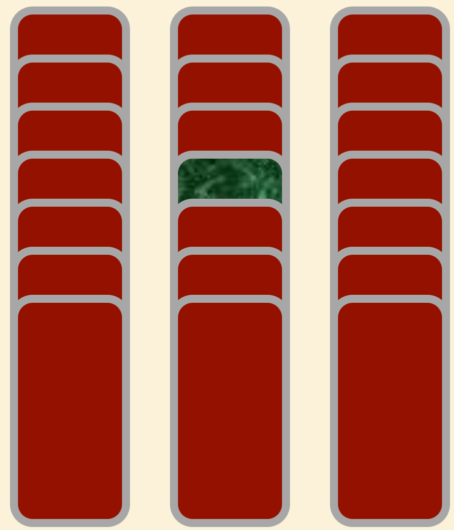


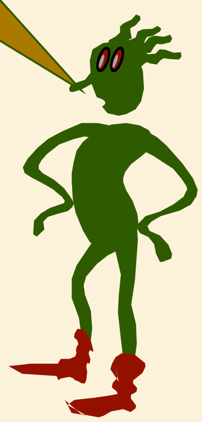

Which column?
left

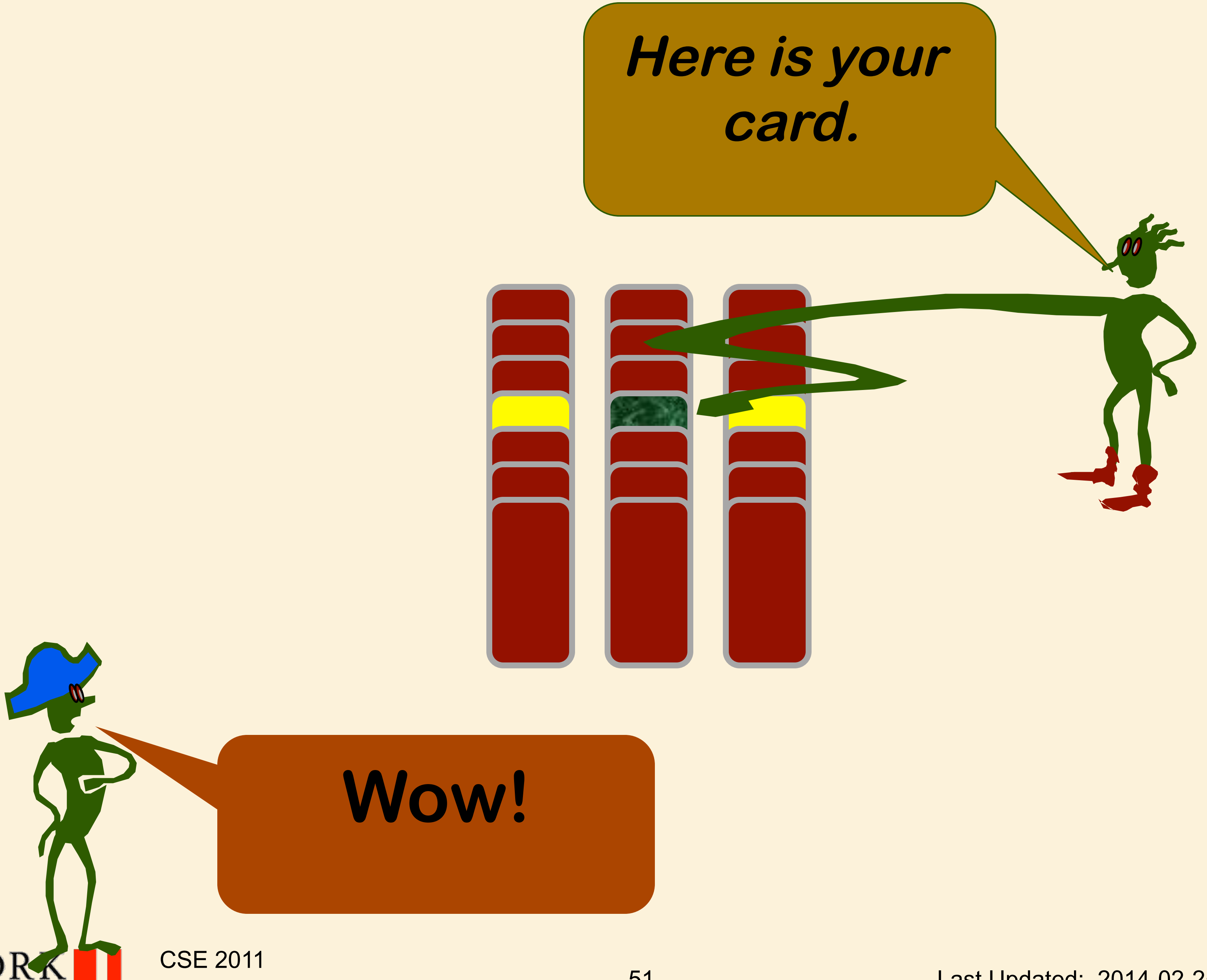
Loop Invariant:
The selected card is one of these.



Selected column is placed in the middle






Here is your card.
Wow!

Ternary Search

- Loop Invariant: selected card in central subset of cards

Size of subset = $\lceil n/3^{i-1} \rceil$

where

$n =$ total number of cards

$i =$ iteration index

- How many iterations are required to guarantee success?

Learning Outcomes

- From this lecture, you should be able to:
  - Use the loop invariant method to think about iterative algorithms.
  - Prove that the loop invariant is established.
  - Prove that the loop invariant is maintained in the ‘typical’ case.
  - Prove that the loop invariant is maintained at all boundary conditions.
  - Prove that progress is made in the ‘typical’ case
  - Prove that progress is guaranteed even near termination, so that the exit condition is always reached.
  - Prove that the loop invariant, when combined with the exit condition, produces the post-condition.
  - Trade off efficiency for clear, correct code.