Loop Invariants and Binary Search

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.

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

    take step

end loop

Establishing Loop Invariant

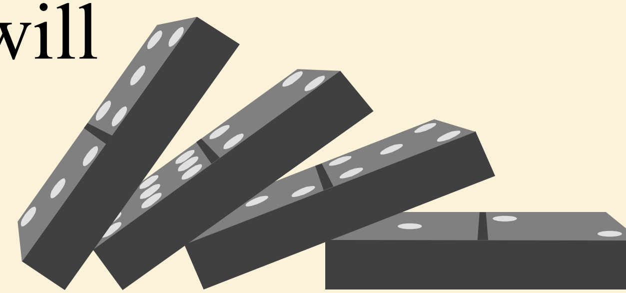
From the Pre-Conditions on the input instance we must establish the loop invariant.

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

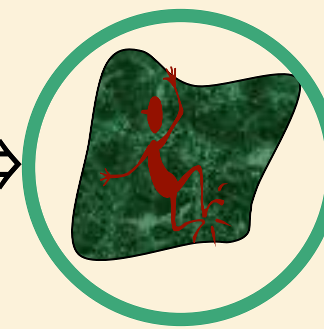
- By Induction the computation will always be in a safe location.



$$\Rightarrow S(0)$$

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

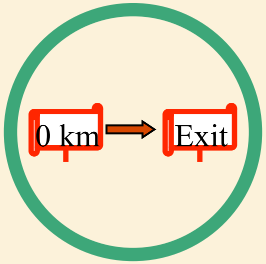
$$\left.\begin{array}{l} \Rightarrow S(0) \\ \Rightarrow \forall i, S(i) \Rightarrow S(i+1) \end{array}\right\} \Rightarrow \forall i, S(i) \Rightarrow$$



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.

End of Lecture

MAR 12, 2015

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

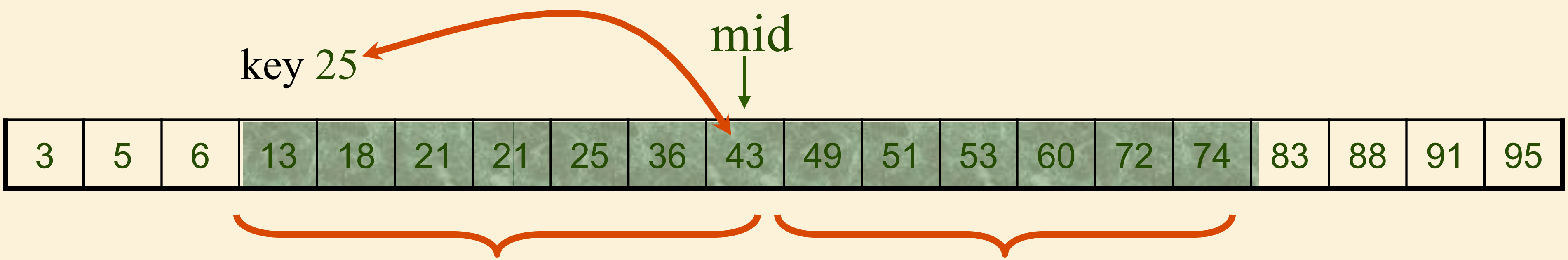
- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

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 $\leq$ 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.

key 43

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

If key $\leq$ 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.

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

79 km

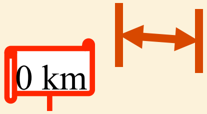
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

75 km

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.

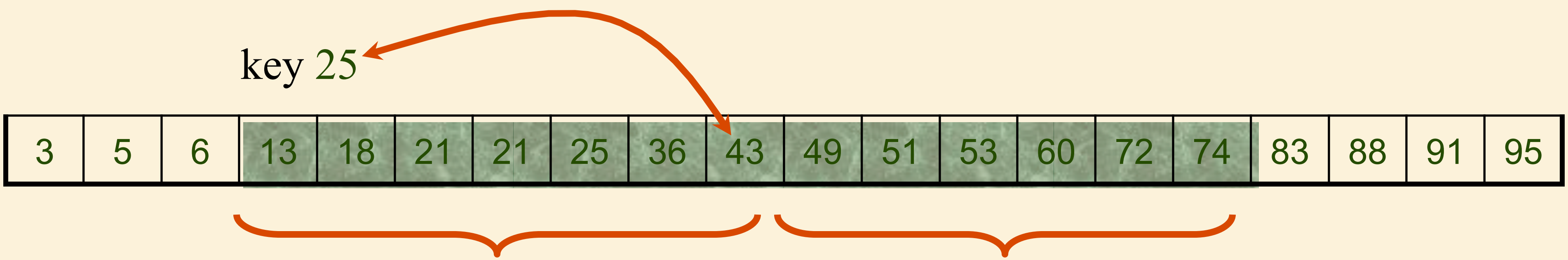
- If element = key, return associated entry.
- Otherwise return false.

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

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

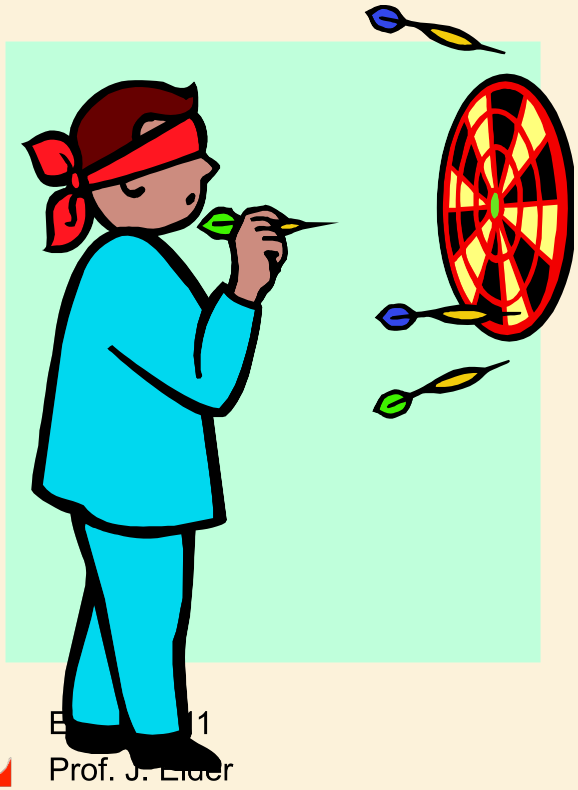
Running Time

- 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.

```
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
    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

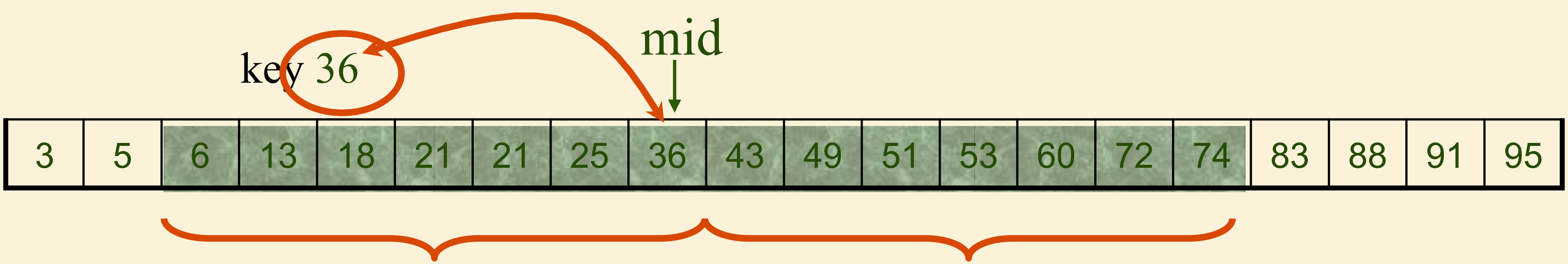
if $key \le A[mid]$
    $q = mid$
else
    $p = mid + 1$
end

or

~~if $key < A[mid]$~~
    ~~$q = mid$~~
~~else~~
    ~~$p = mid + 1$~~
~~end~~

What condition will break the loop invariant?

Boundary Conditions



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

Bug!!

Boundary Conditions

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

OK

if $key < A[mid]$
    $q = mid - 1$
else
    $p = mid$
end

OK

if $key < A[mid]$
    $q = mid$
else
    $p = mid + 1$
end

Not OK!!

Boundary Conditions

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

key 25

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

Shouldn't matter, right?

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

Boundary Conditions

if $key \leq A[mid]$

  $q = mid$

else

  $p = mid + 1$

end

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

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

Boundary Conditions

if $key \le A[mid]$

$q = mid$

else

$p = mid + 1$

end

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

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

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

- Another bug!

No progress toward goal: Loops Forever!

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

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 $\le$ 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?

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

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

if $key \le A[mid]$

or if $key < A[mid]$?

$q = mid$

else

$p = mid + 1$

end

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.

Assignment 3 Q2: kth Smallest of Union

- e = kthSmallestOfUnion(k)
  - e.g., kthSmallestOfUnion(6) = 7
- Observation: e must be in first k positions of A1 or A2, i.e.,

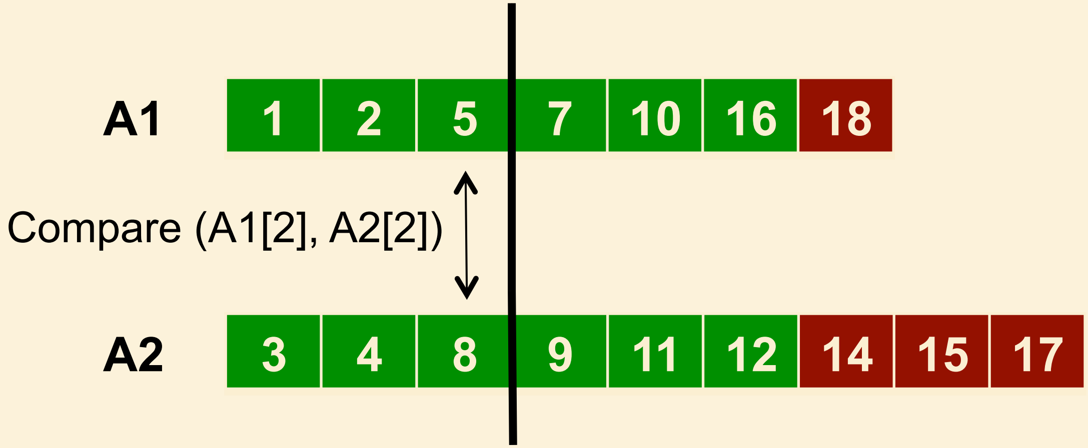
$$e \in A_1[0 \ldots k-1] \cup A_2[0 \ldots k-1]$$

- → Step 1: Truncate A1 and A2 to length k.

A1	1	2	5	7	10	16	18		
A2	3	4	8	9	11	12	14	15	17

Assignment 3 Q2: kth Smallest of Union

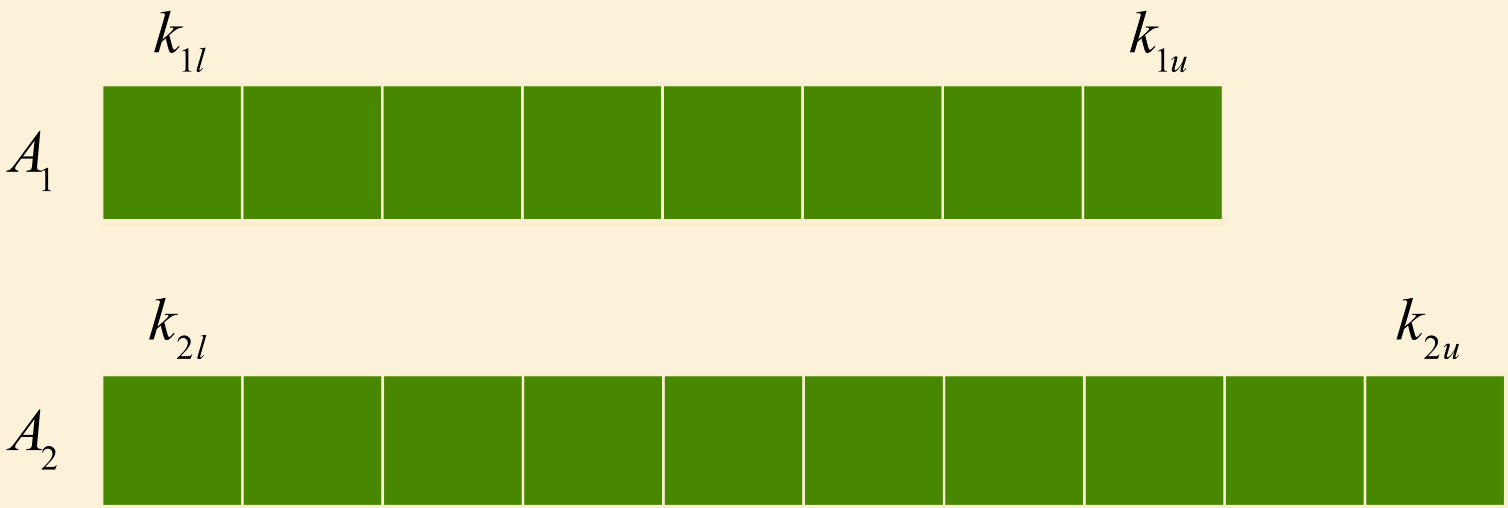
- e = kthSmallestOfUnion(k)
  - e.g., kthSmallestOfUnion(6) = 7
- Step 2: Divide and Conquer!
  - Case 1: A1[2] > A2[2]. In what intervals must the kth smallest lie?
  - Case 2: A1[2] < A2[2]. In what intervals must the kth smallest lie?



Assignment 3 Q2: kth Smallest of Union

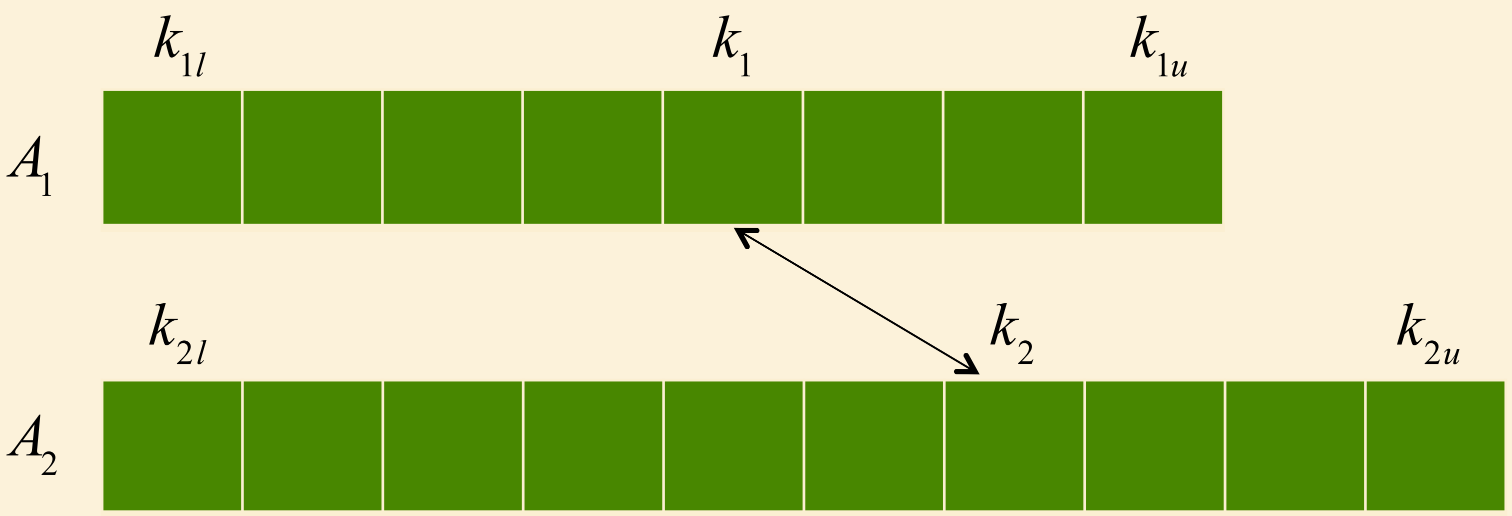
- More generally: maintain the loop invariant that the kth smallest key is stored in

$$A_1\left[k_{1l}\ldots k_{1u}\right]\cup A_2\left[k_{2l}\ldots k_{2u}\right]$$



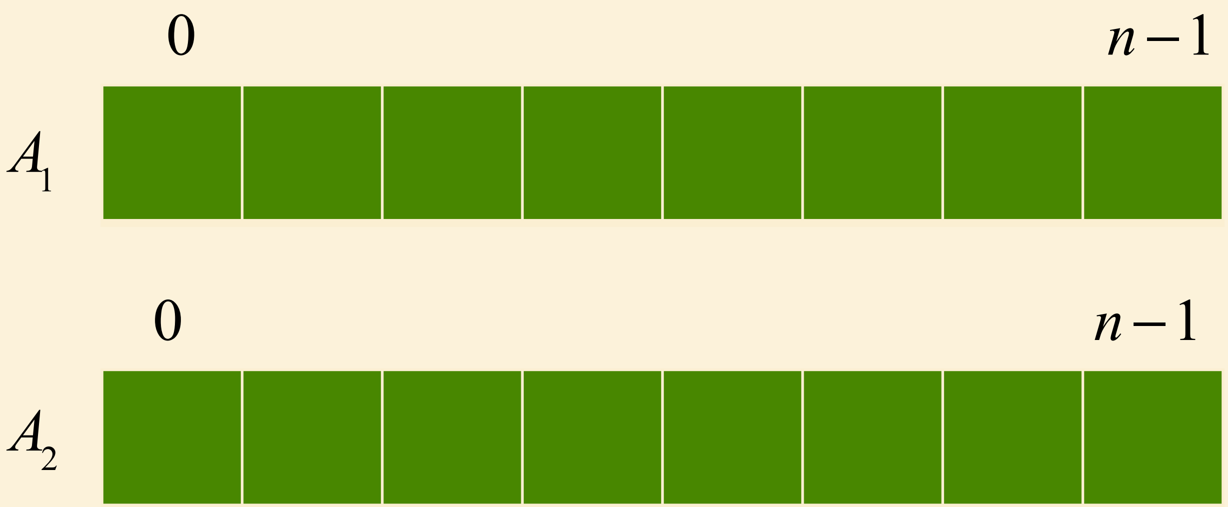
Assignment 3 Q2: kth Smallest of Union

- Now bisect A_1: $k_1 = \lfloor (k_{1l} + k_{1u})/2 \rfloor$ and define $k_2 = k - k_1 - 1.$
- Note that $k_1 + k_2 = k - 1$
- Now compare $A_1[k_1]$ and $A_2[k_2]$.
- What sub-intervals can you safely rule out?
- Now update $k_{1l}, k_{1u}, k_{2l}, k_{2u}$ accordingly, and iterate!



Assignment 3 Q2: kth Smallest of Union

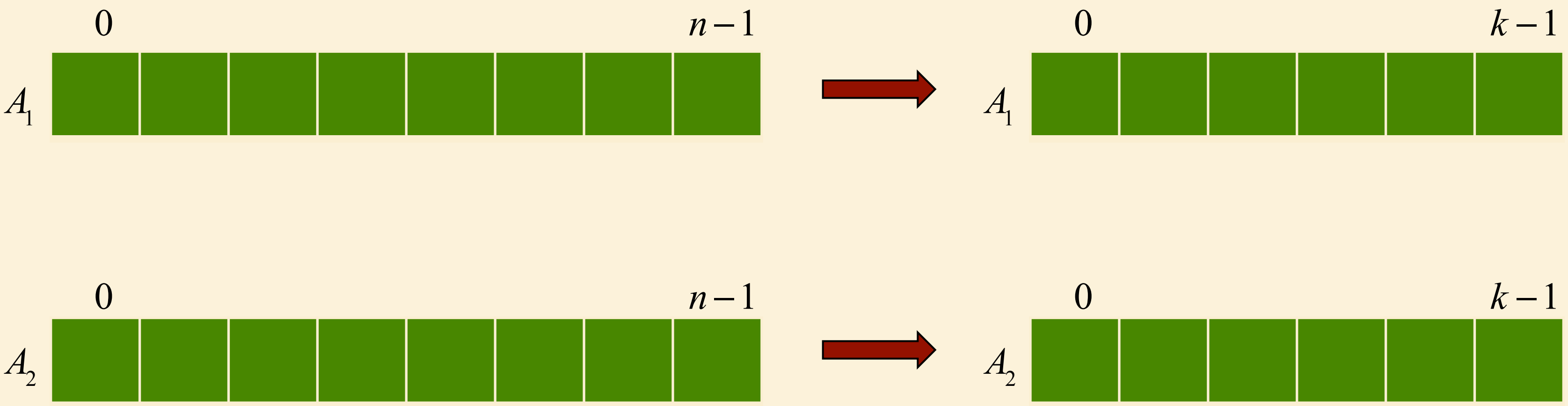
- To simplify the problem, assume that original input arrays are of the same length.
- Note that k < 2n, or a RankOutOfRangeException is thrown.



Compare (A1[2], A2[2])

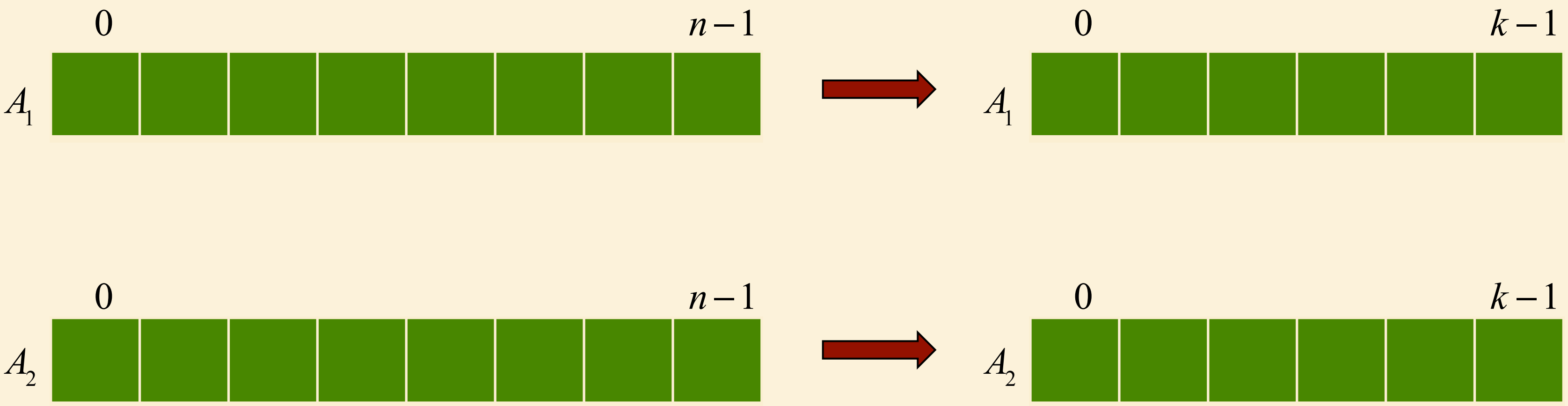
Assignment 3 Q2: kth Smallest of Union

- What if k < n?
- Then we first truncate both arrays to be of length k.



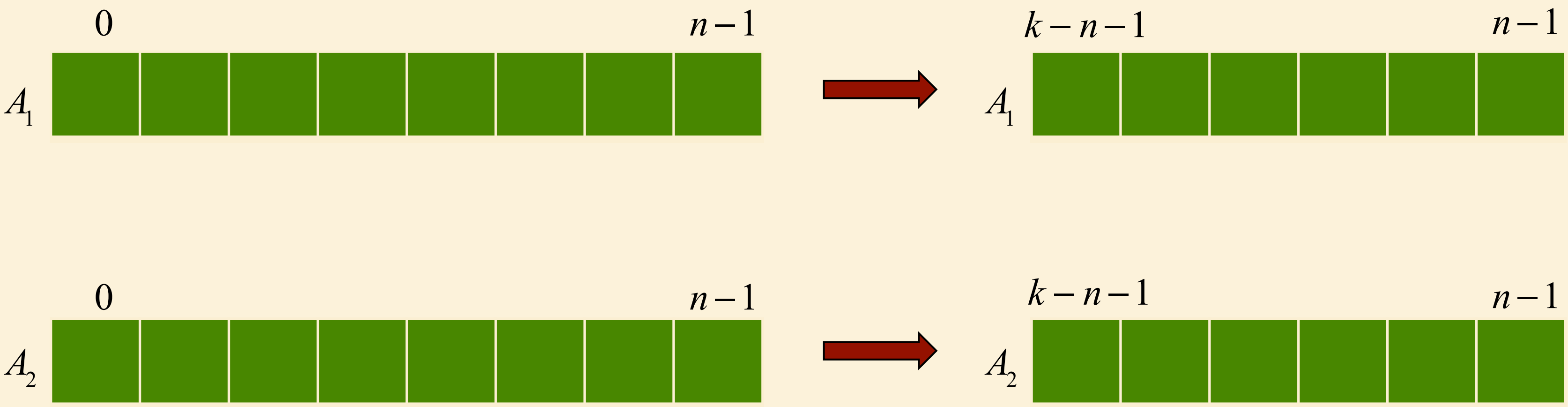
Assignment 3 Q2: kth Smallest of Union

- What if k > n?
- Then we first trim the tails of the arrays so they are of length k.



Assignment 3 Q2: kth Smallest of Union

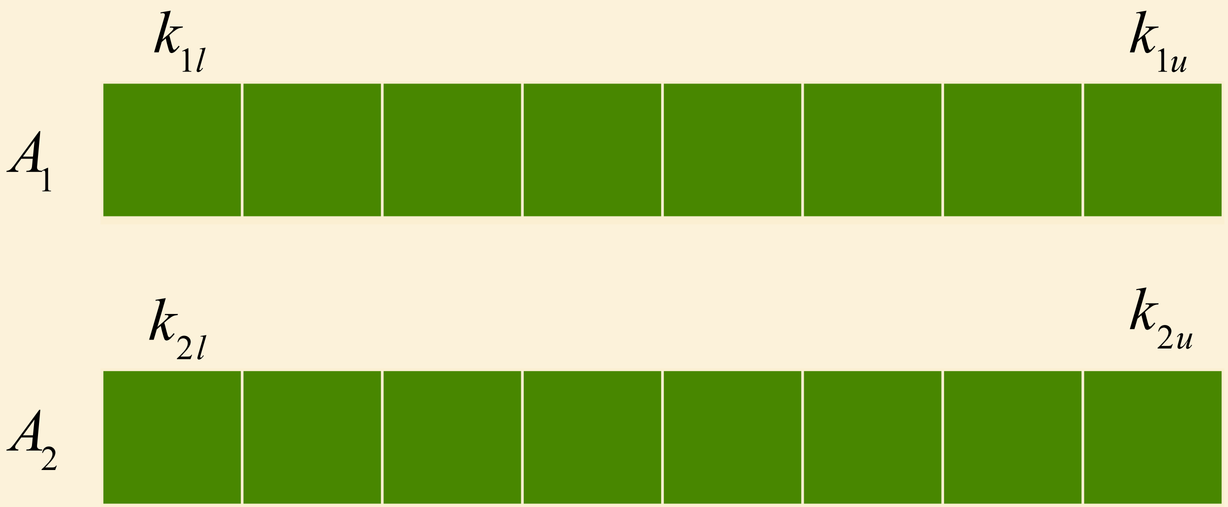
- What if k > n + 1?
- Then we first trim the beginning of both arrays so they are of length n – (k – n -1) + 1 = 2n – k + 1.



Assignment 3 Q2: kth Smallest of Union

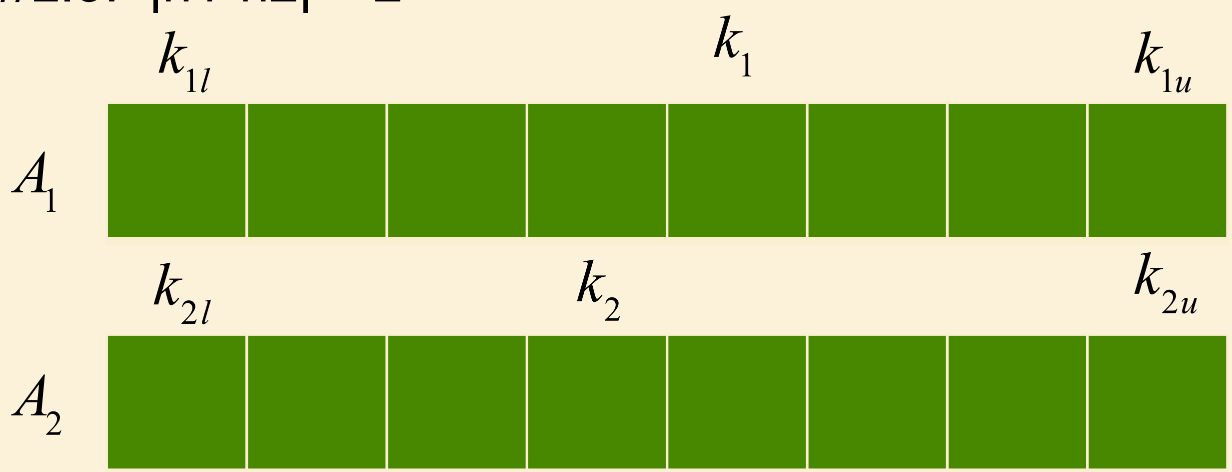
- Thus at the beginning of the loop, we have the kth smallest element in

$$A_1\left[k_{1l} \ldots k_{1u}\right] \cup A_2\left[k_{2l} \ldots k_{2u}\right]$$



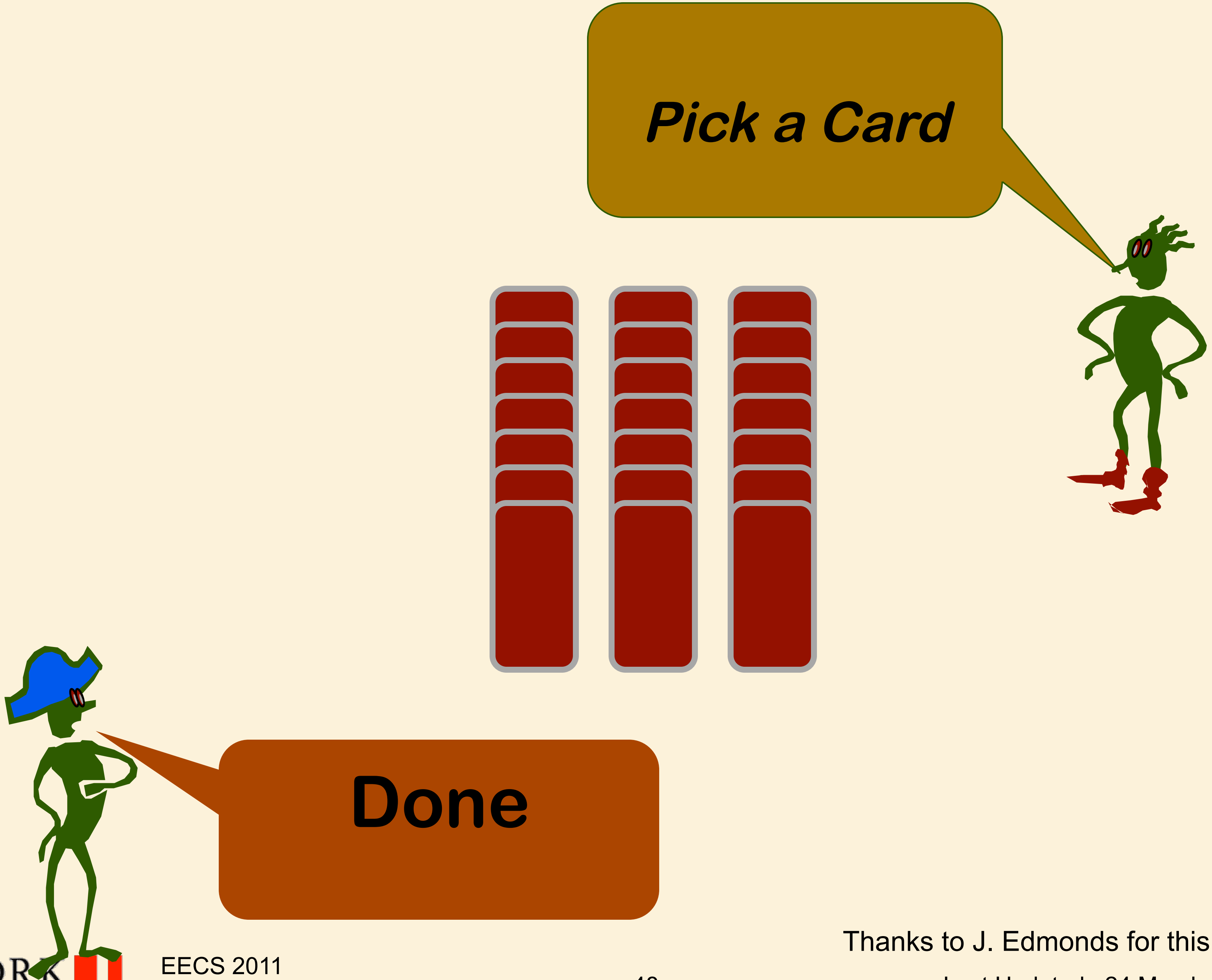
Assignment 3 Q2: kth Smallest of Union

- Now let $k_1 = \left\lceil \left(k_{1l} + k_{1u}\right)/2 \right\rceil$ and $k_2 = \left\lfloor \left(k_{2l} + k_{2u}\right)/2 \right\rfloor$
- Then we have that $k_1 + k_2 = k - 1$.
- In the loop we will compare $A_1\left[k_1\right]$ and $A_2\left[k_2\right]$, and update, while preserving 3 loop invariants:
- //LI1: kth smallest is in A1[k1l...k1u] or A2[k2l...k2u]
- //LI2: k1 + k2 = k-1
- //LI3: |n1-n2| < 2



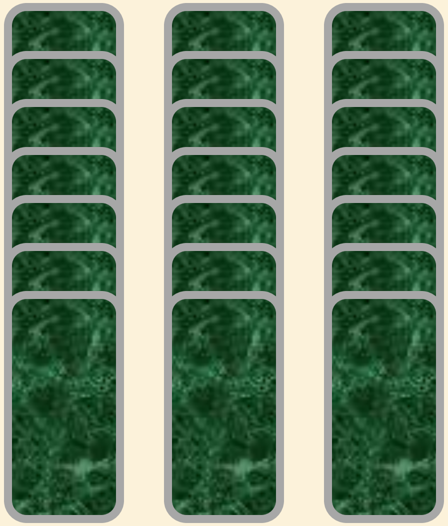
Card Trick

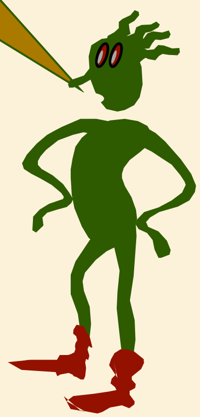
- A volunteer, please.

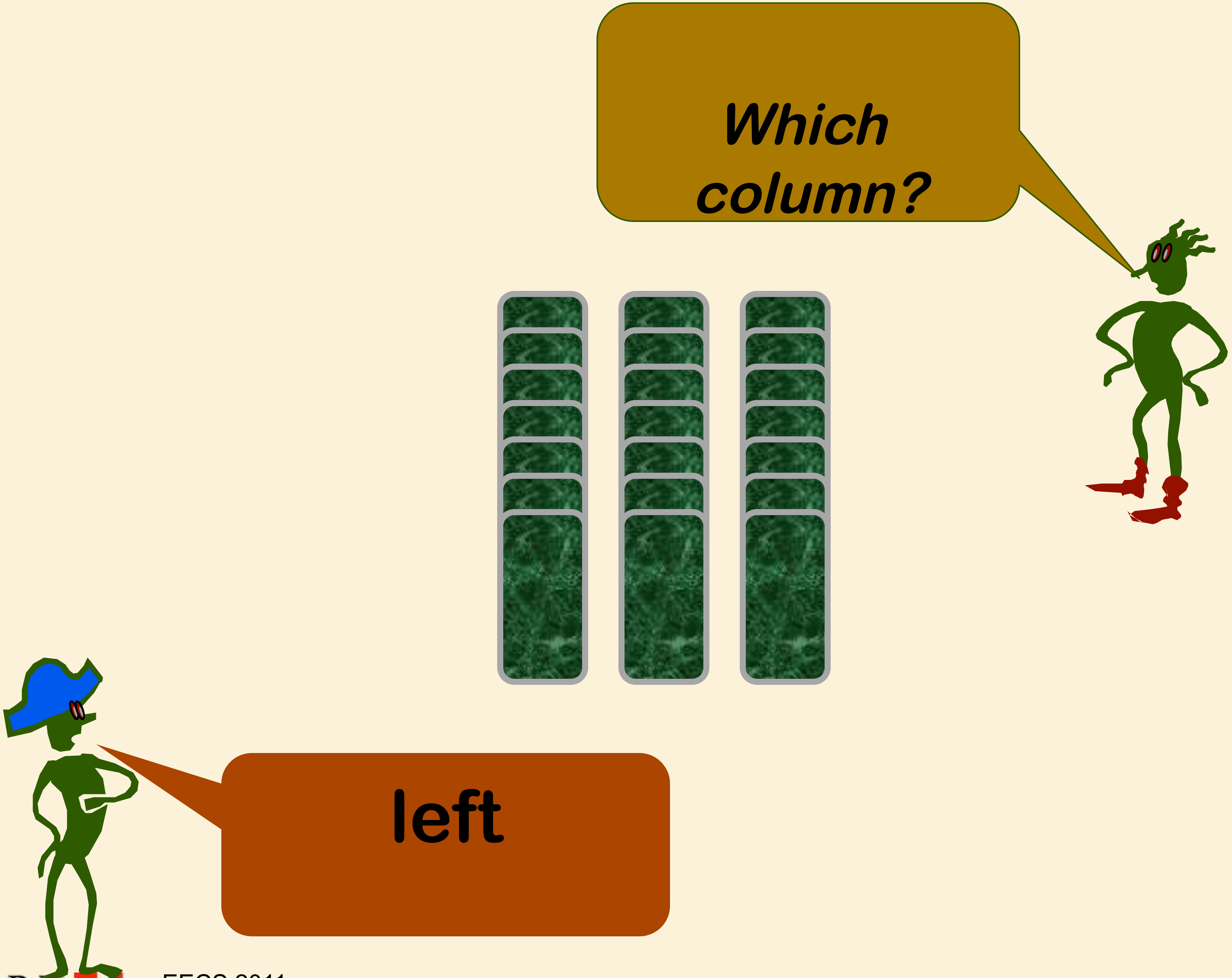


Thanks to J. Edmonds for this example.

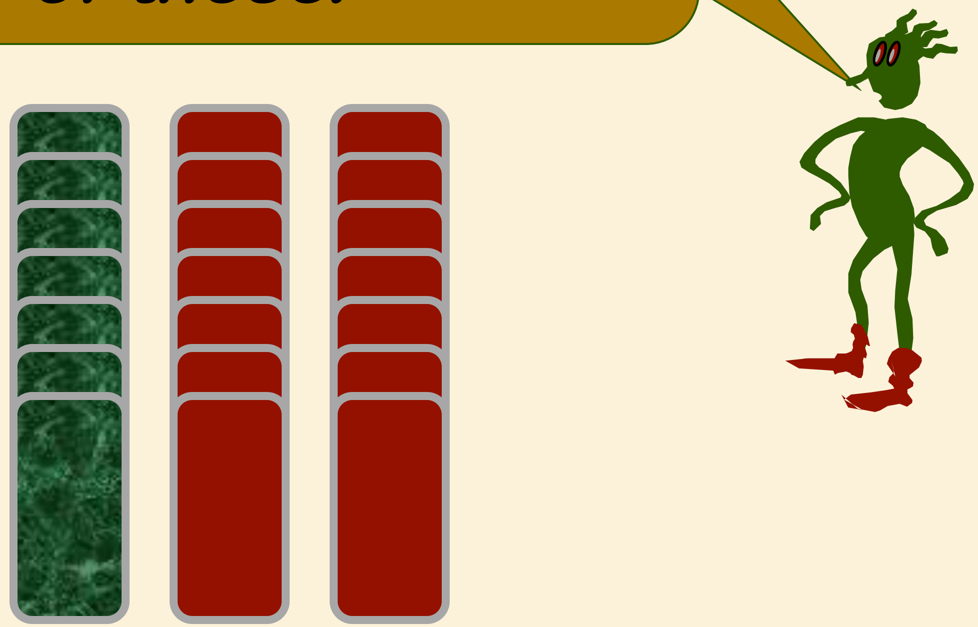
Loop Invariant:
The selected card is one of these.



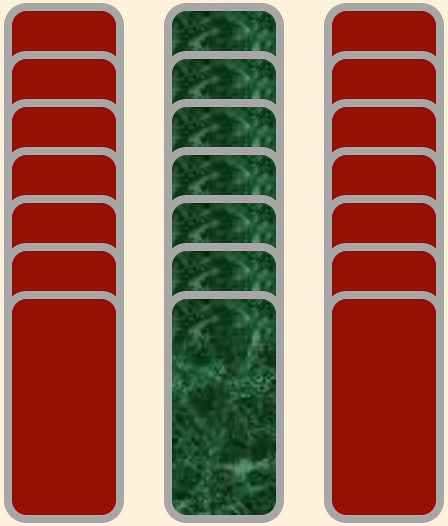


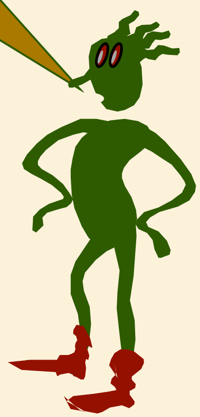

Which column?
left

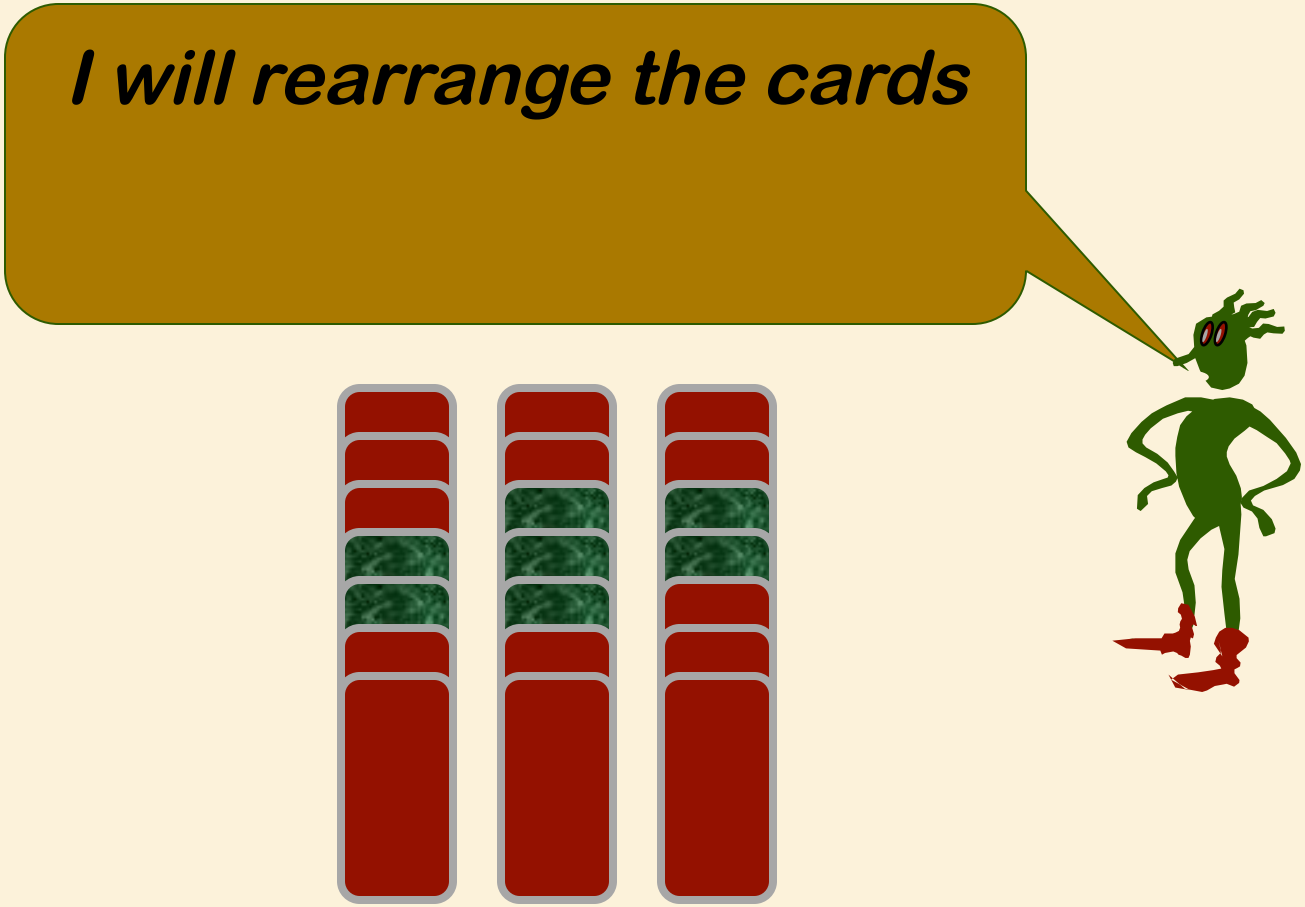
Loop Invariant: The selected card is one of these.



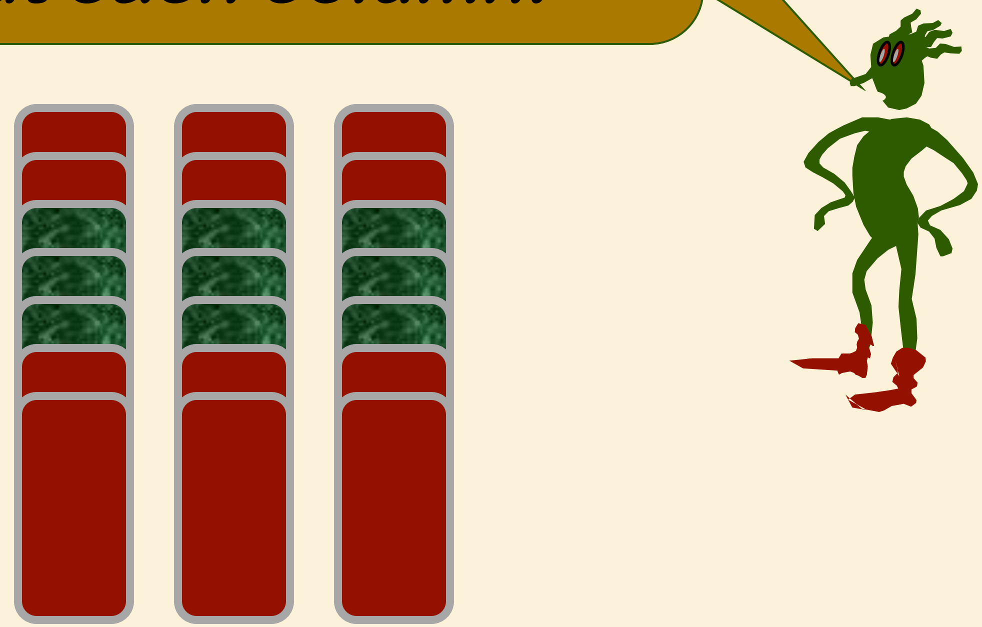
Selected column is placed in the middle

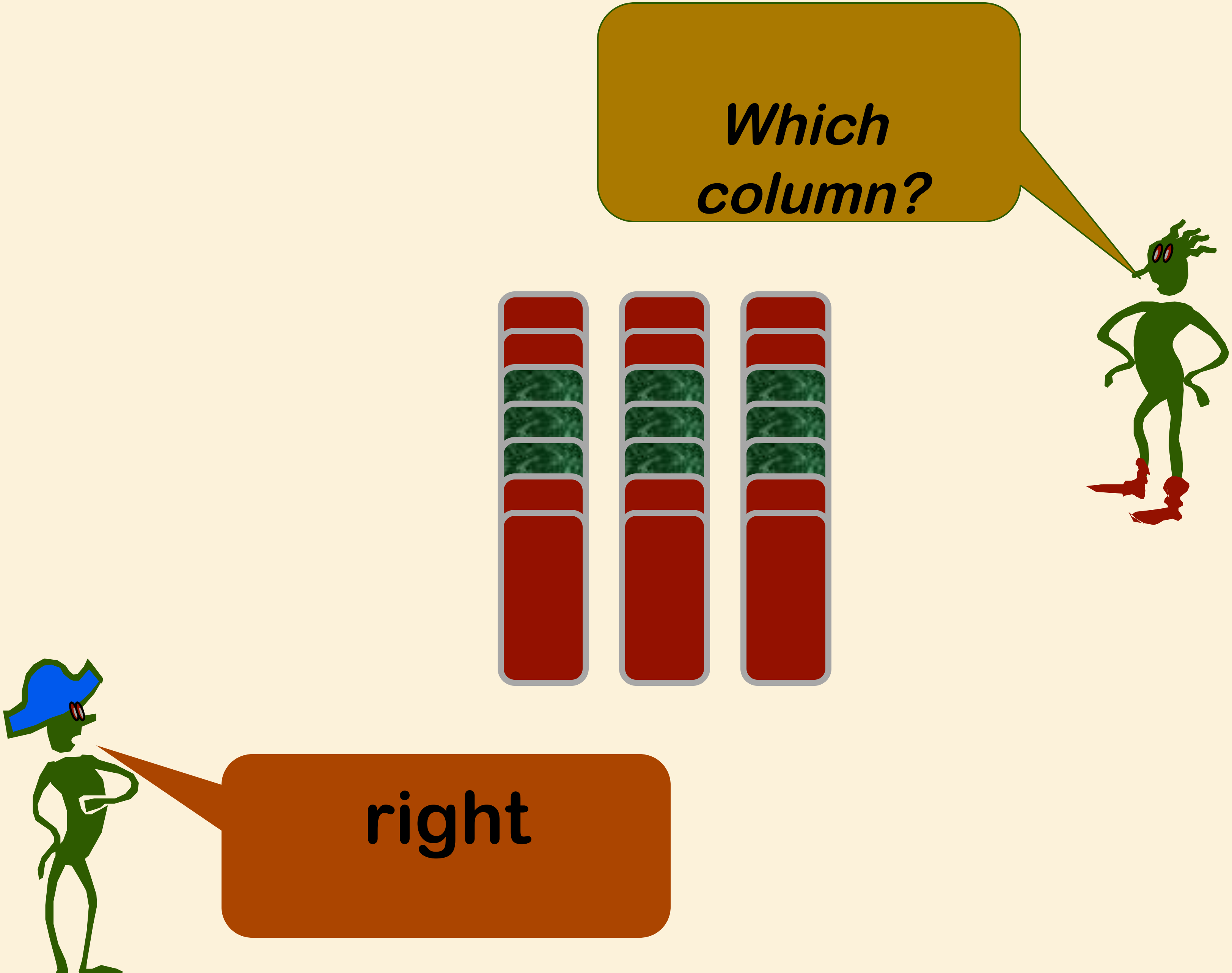




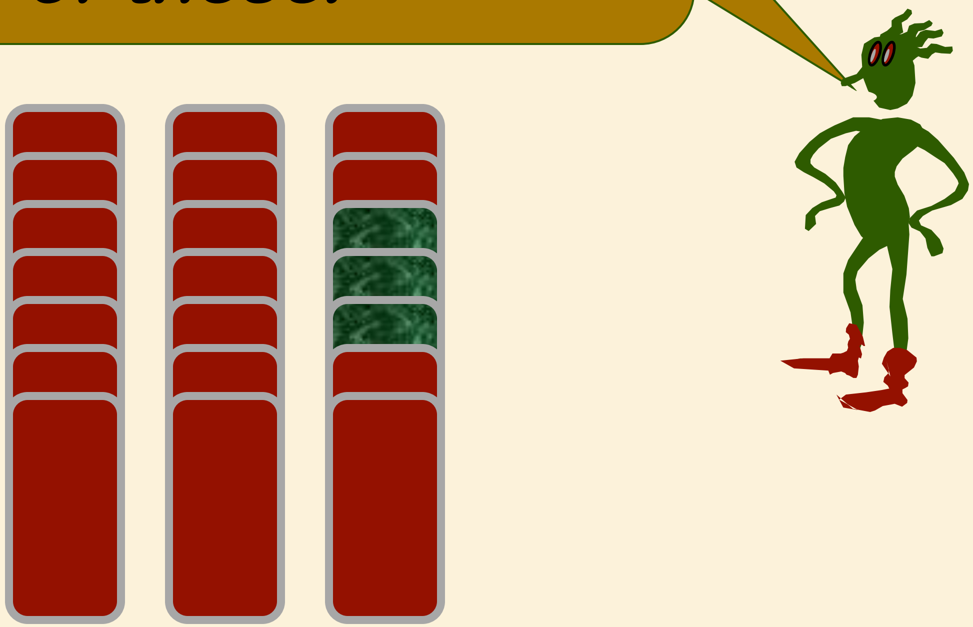

I will rearrange the cards

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

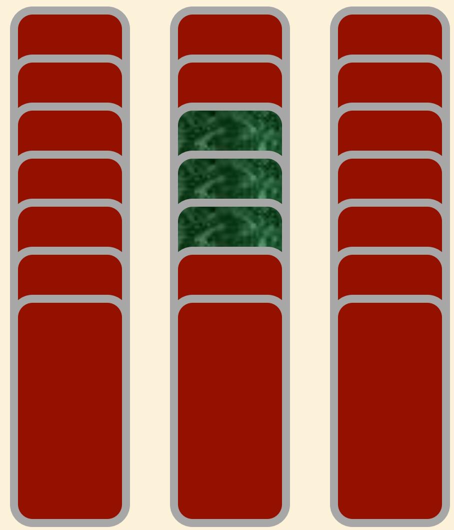


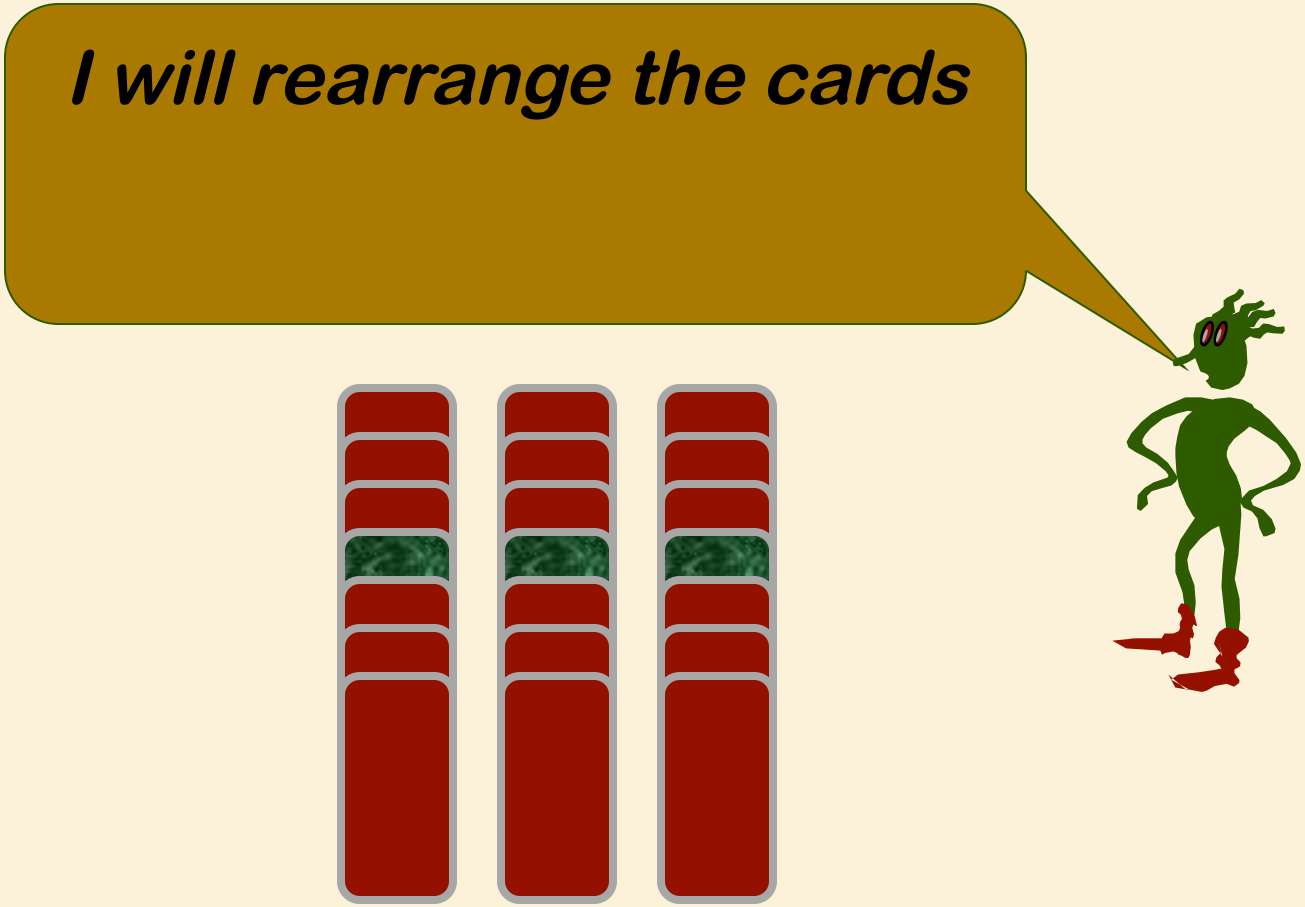

Which column?
right

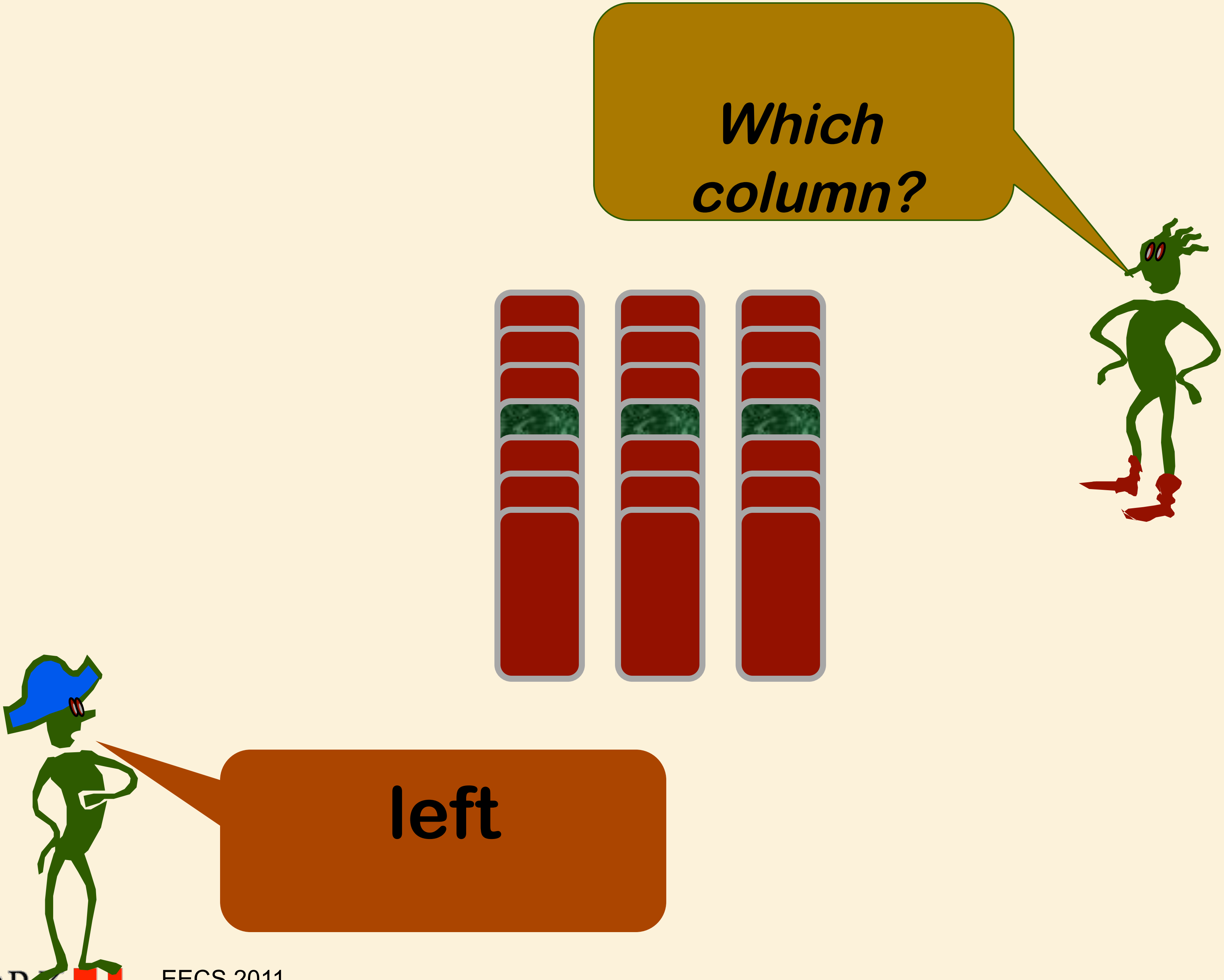
Loop Invariant:
The selected card is one of these.



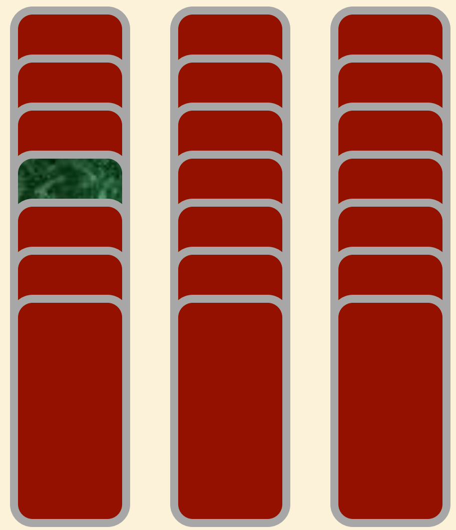
Selected column is placed in the middle

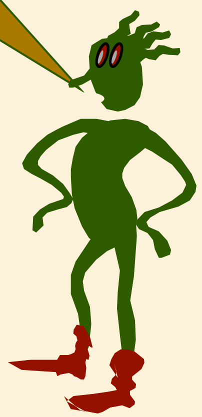



I will rearrange the cards

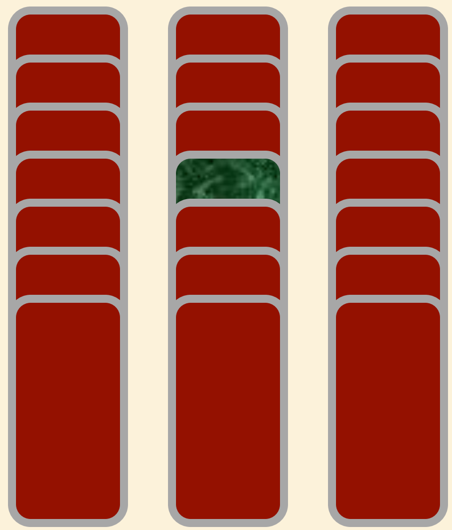

Which column?
left

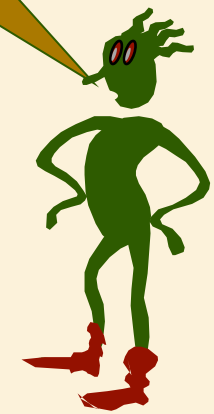
Loop Invariant:
The selected card is one of these.

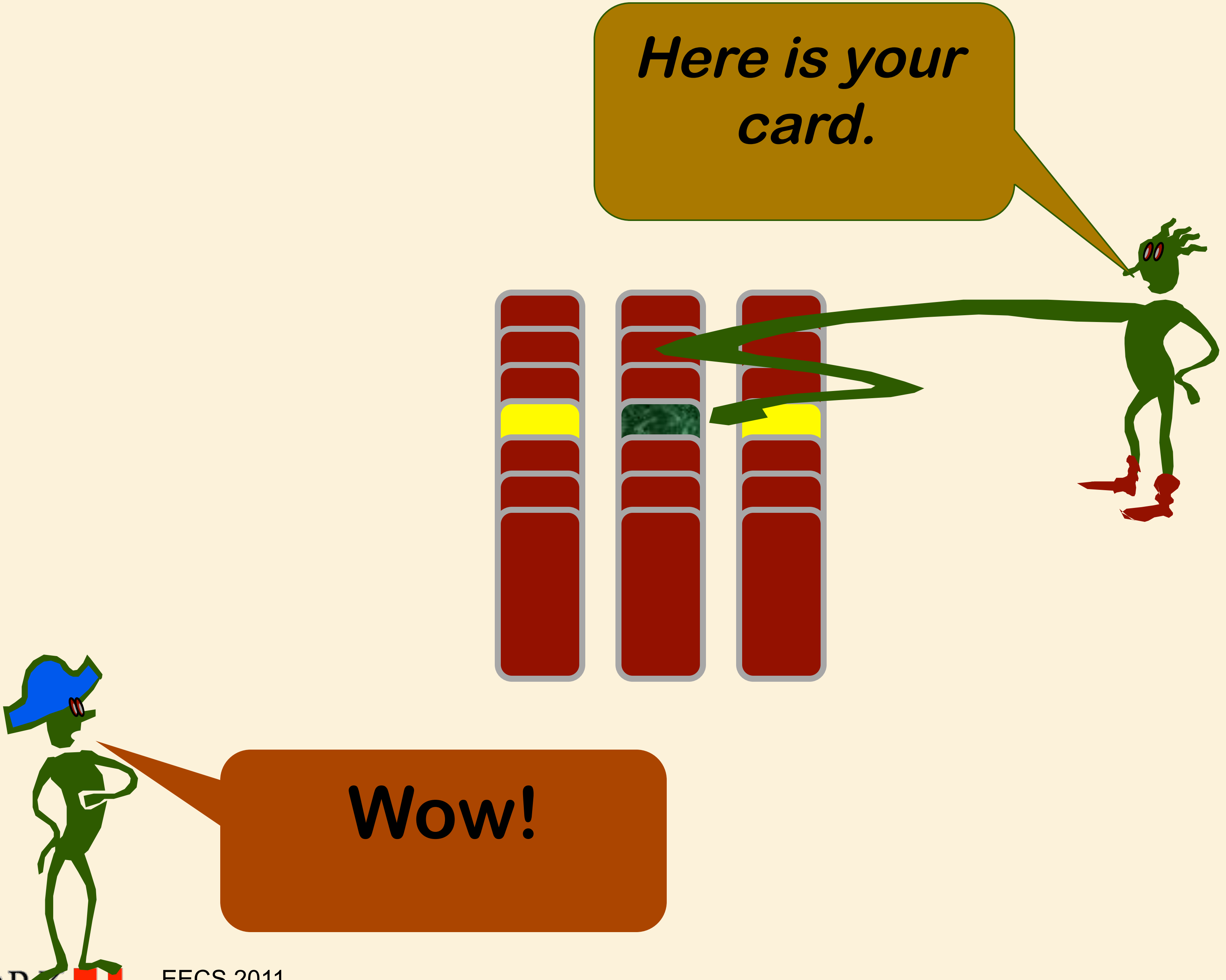




Selected column is placed in the middle






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?

Outline

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

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.