Central Algorithmic Techniques

Iterative Algorithms

Code
Representation of an Algorithm

```
class InsertionSortAlgorithm extends SortAlgorithm {
 void sort(int a[]) throws Exception {
          for (int i = 1; i < a.length; i++) {
              int j = i;
              int B = a[i];
              while ((j > 0) && (a[j-1] > B)) {
                        a[j] = a[j-1];
                        j--;  }
              a[j] = B;
           }}
```

Pros and Cons?

Code
Representation of an Algorithm

Pros:

- Runs on computers
- Precise and succinct

Cons:

- I am not a computer
- I need a higher level of intuition.
- Prone to bugs
- Language dependent

Two Key Types of Algorithms

- Iterative Algorithms
- Recursive Algorithms

Iterative Algorithms

Take one step at a time

towards the final destination

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

Loop Invariants

A good way to structure many programs:

- Store the key information you currently know in some data representation.
- In the main loop,
  - take a step forward towards destination
  - by making a simple change to this data.

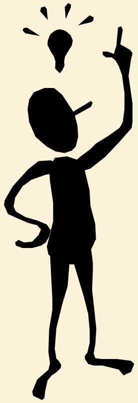
The Getting to School Problem

Problem Specification

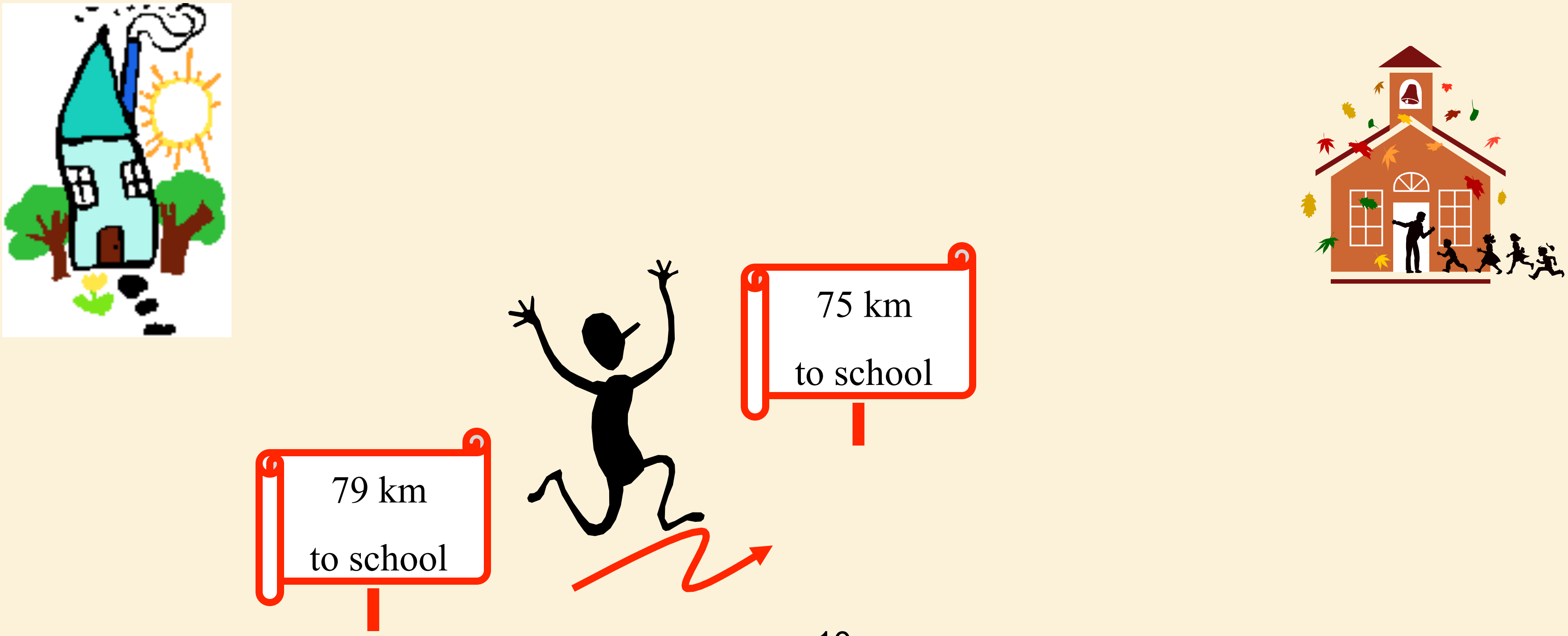
- Pre condition: location of home and school
- Post condition: Traveled from home to school

General Principle

- Do not worry about the entire computation.
- Take one step at a time!

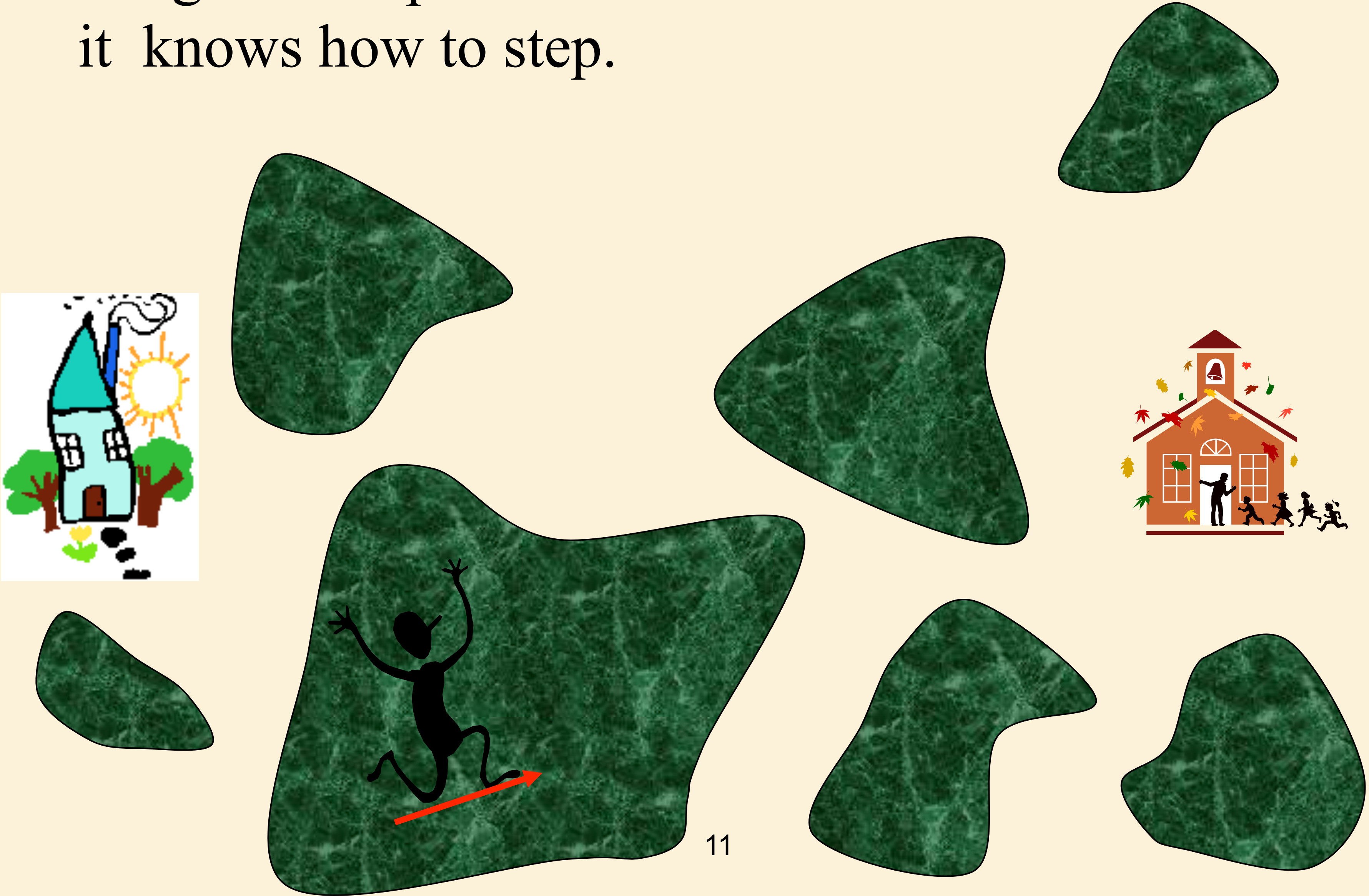


A Measure of Progress



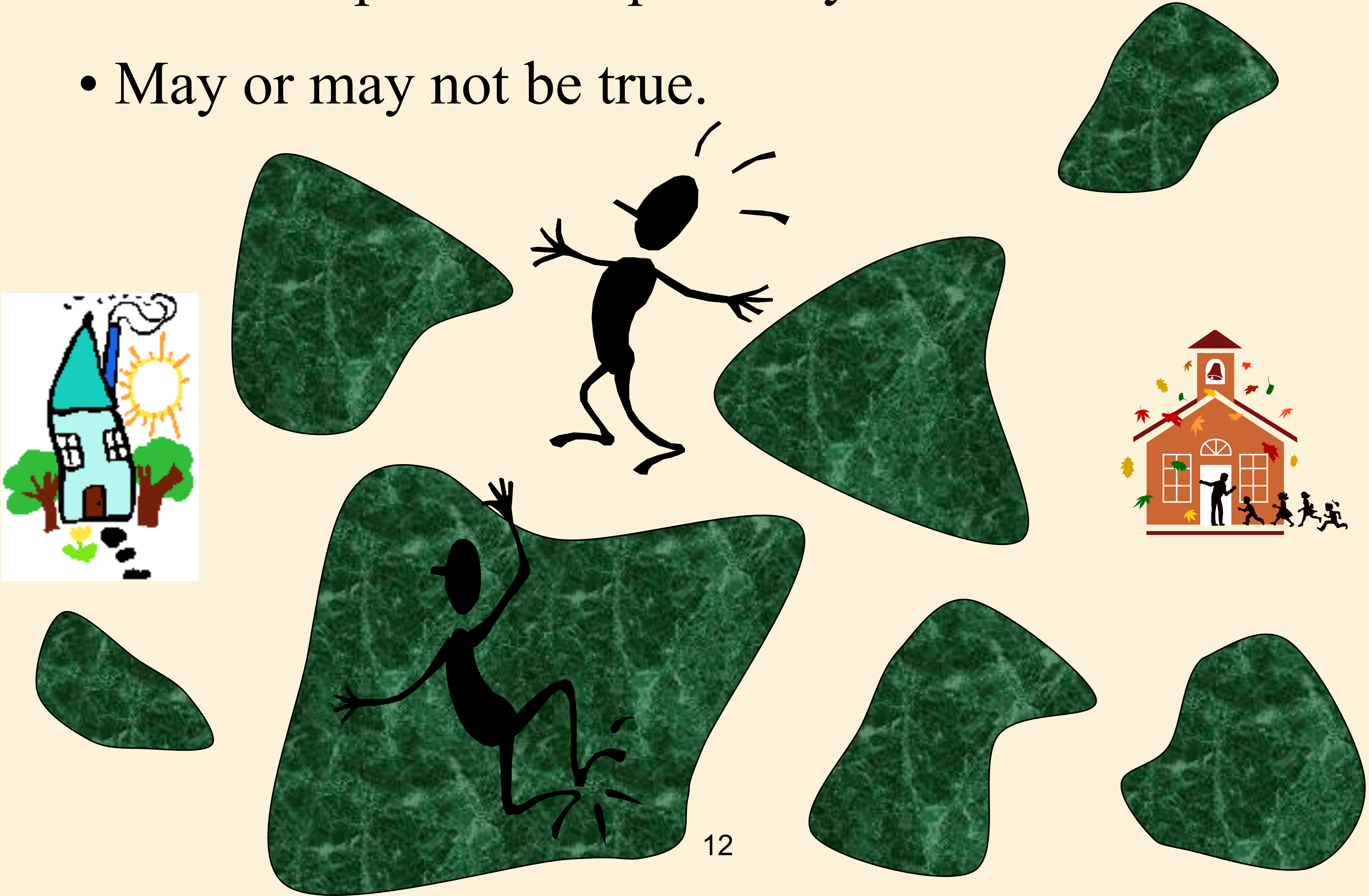
Safe Locations

- Algorithm specifies from which locations it knows how to step.



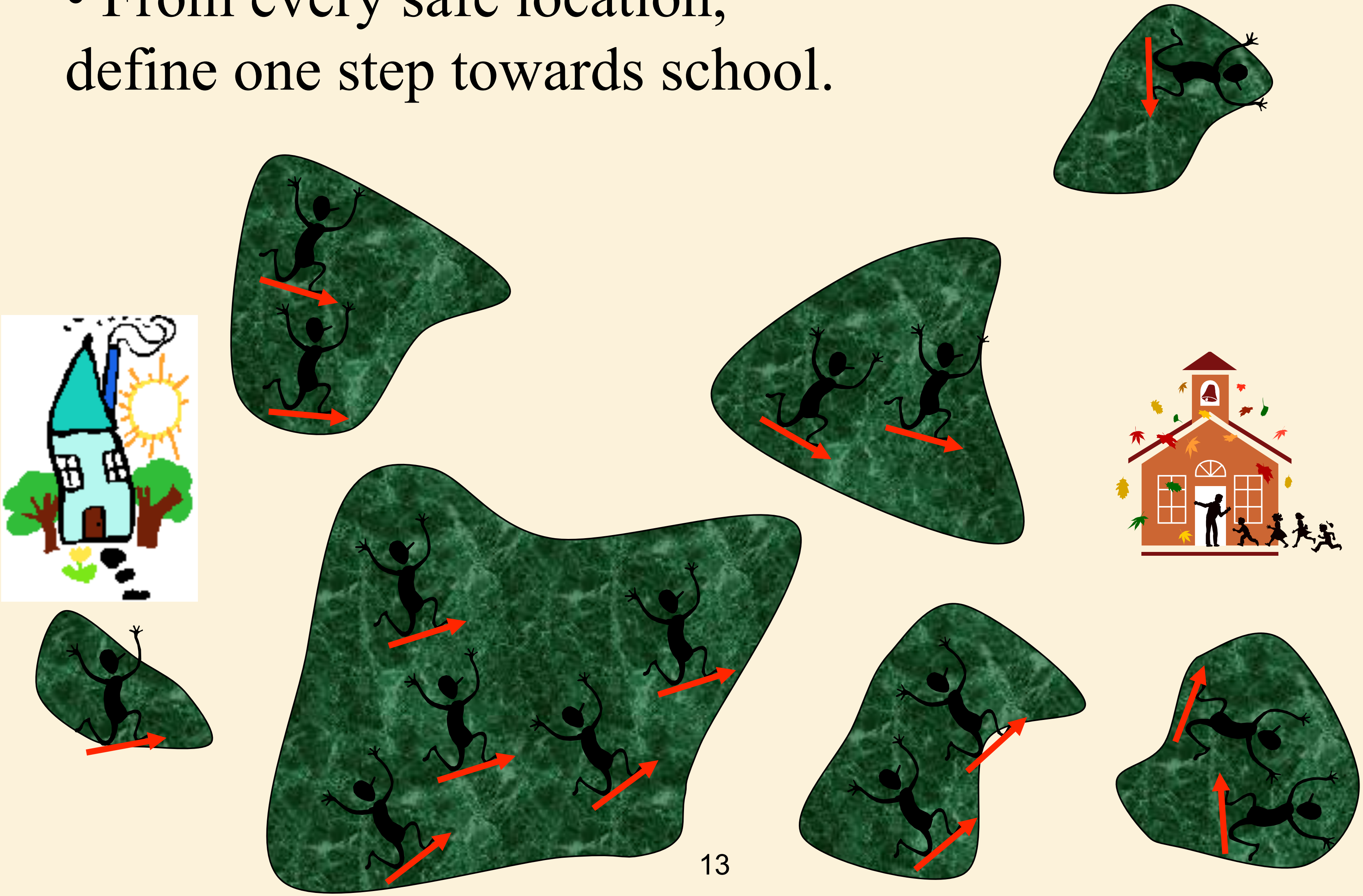
Loop Invariant

- “The computation is presently in a safe location.”
- May or may not be true.



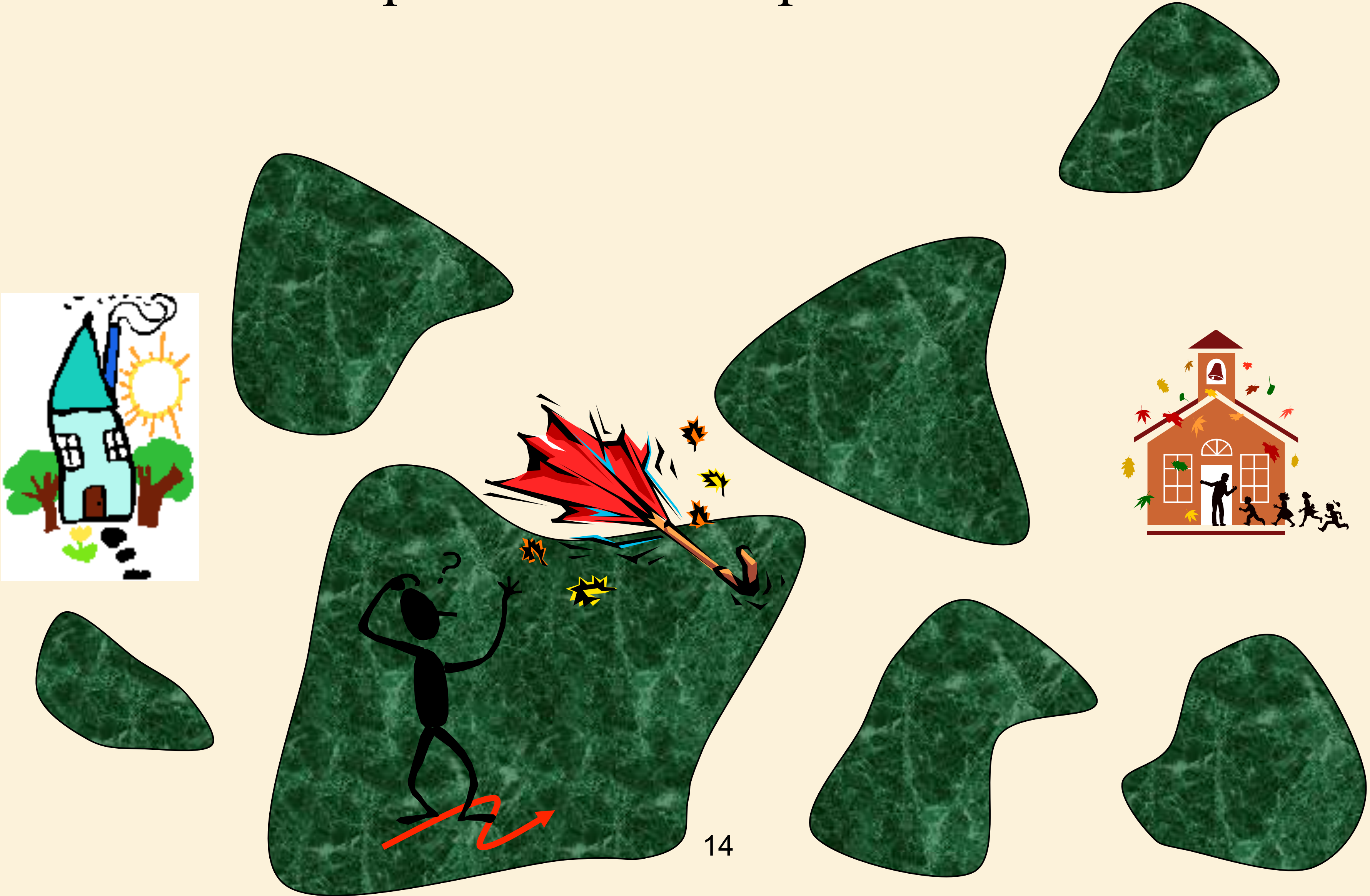
Defining Algorithm

- From every safe location, define one step towards school.



Take a step

- What is required of this step?



Maintain Loop Invariant

- If the computation is in a safe location, it does not step into an unsafe one.

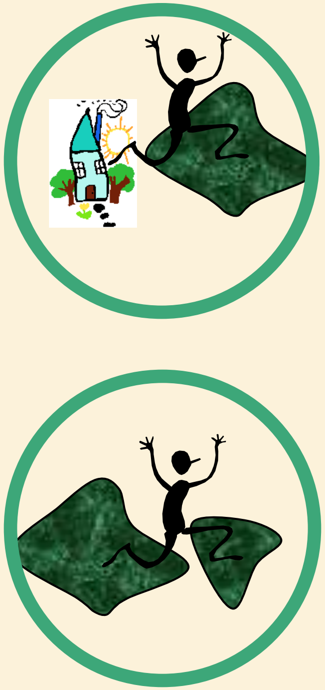
- Can we be assured that the computation will always be in a safe location?

No. What if it is not initially true?

Establishing Loop Invariant

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

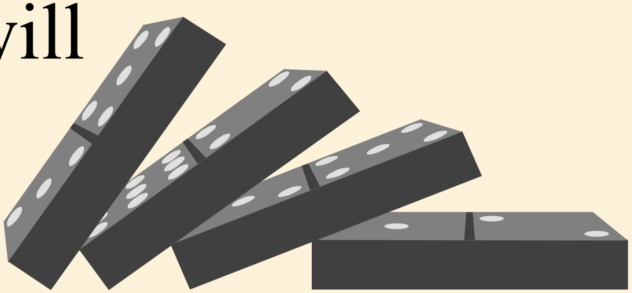
Maintain Loop Invariant



- 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.

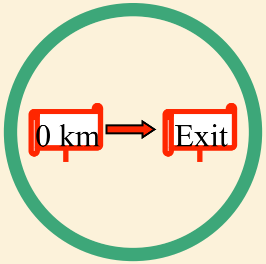


$$\left.\begin{array}{l}\Rightarrow S(0)\\ \\ \Rightarrow \forall i, S(i) \Rightarrow S(i+1)\end{array}\right\} \Rightarrow \forall i, S(i) \Rightarrow$$

Ending The Algorithm

- Define Exit Condition

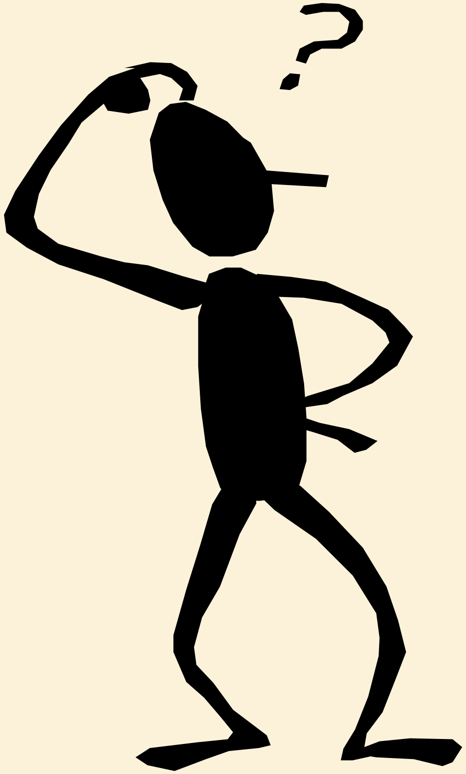
- Termination: With sufficient progress, the exit condition will be met.



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

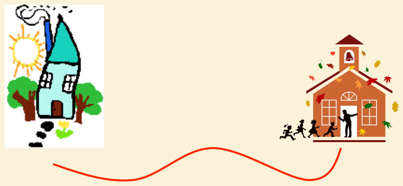
from these we must establish the post conditions.

Let's Recap



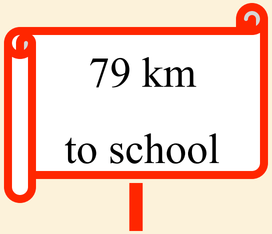
Designing an Algorithm

Define Problem

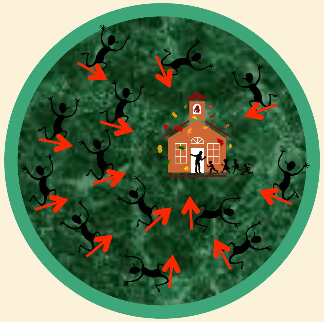


Define Loop Invariants

Define Measure of Progress



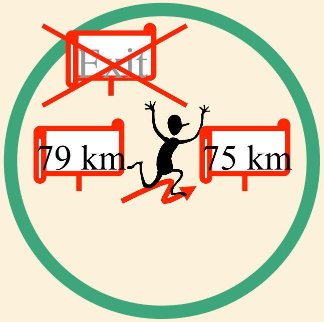
Define Step



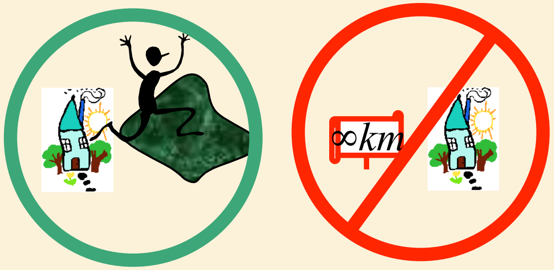
Define Exit Condition

Maintain Loop Inv

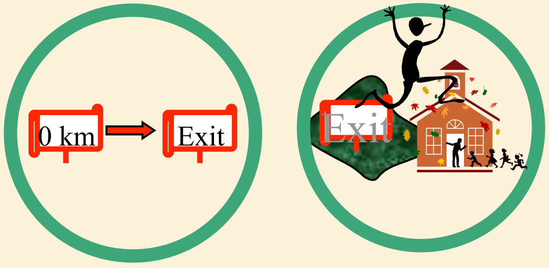
Make Progress



Initial Conditions



Ending



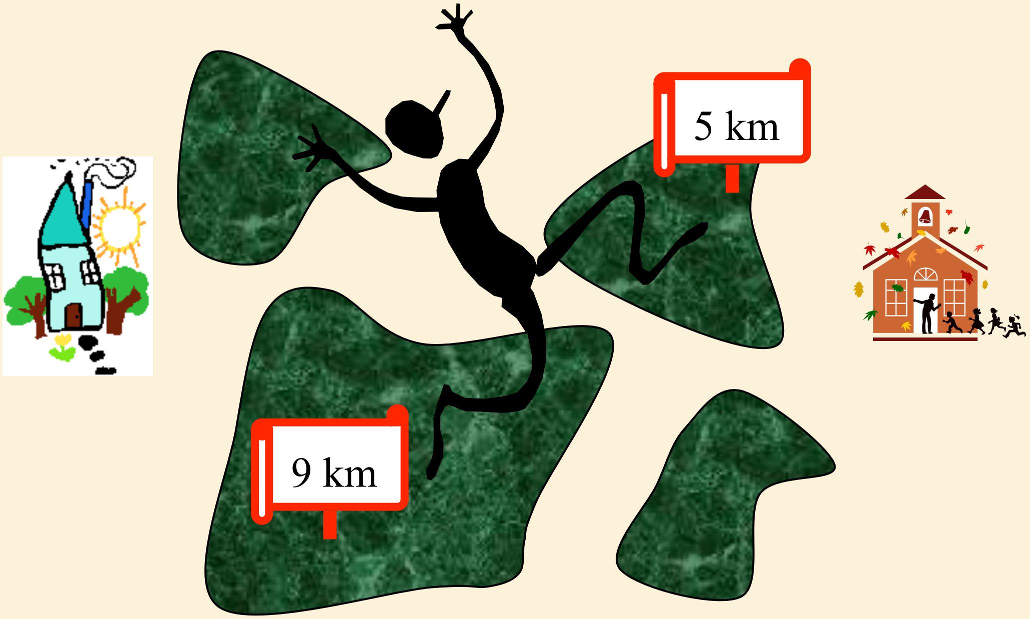
Simple Example

Insertion Sort Algorithm

Code
Representation of an Algorithm

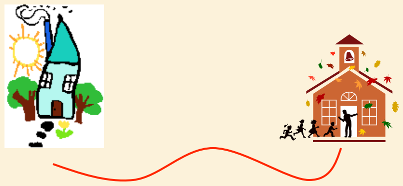
```
class InsertionSortAlgorithm extends SortAlgorithm {
 void sort(int a[]) throws Exception {
           for (int i = 1; i < a.length; i++) {
              int j = i;
              int B = a[i];
              while ((j > 0) && (a[j-1] > B)) {
                      a[j] = a[j-1];
                      j--;  }
              a[j] = B;
              }}
```

Higher Level Abstract View Representation of an Algorithm



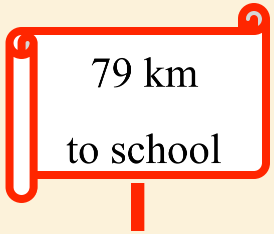
Designing an Algorithm

Define Problem

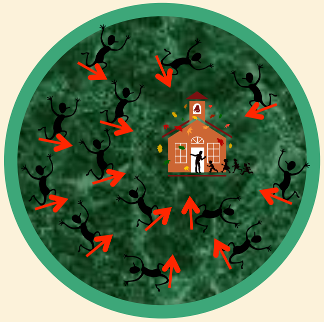


Define Loop Invariants

Define Measure of Progress



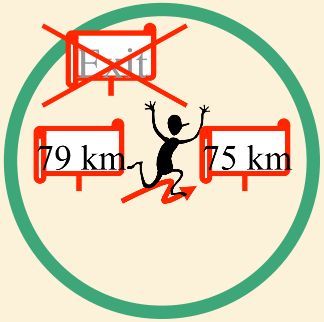
Define Step



Define Exit Condition

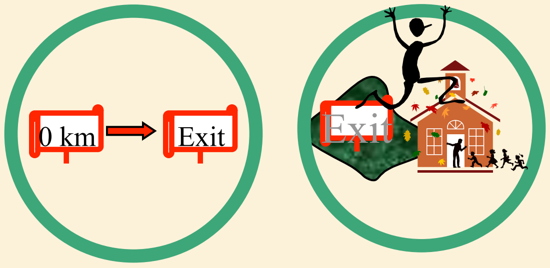
Maintain Loop Inv

Make Progress



Initial Conditions

Ending



Problem Specification

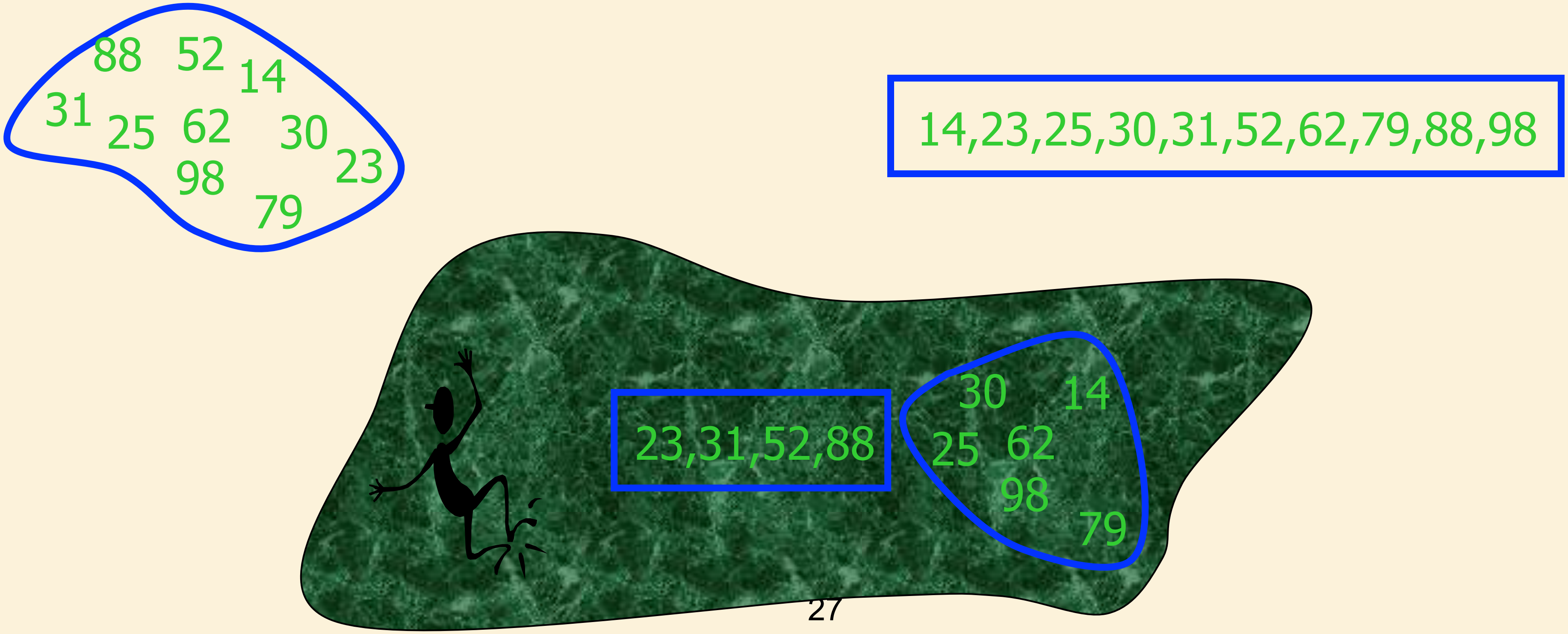
- Precondition: The input is a list of n values with the same value possibly repeated.
- Post condition: The output is a list consisting of the same n values in non-decreasing order.

88 52 14
31 25 98 30
62 23
79

14,23,25,30,31,52,62,79,88,98

Define Loop Invariant

- Some subset of the elements are sorted
- The remaining elements are off to the side.



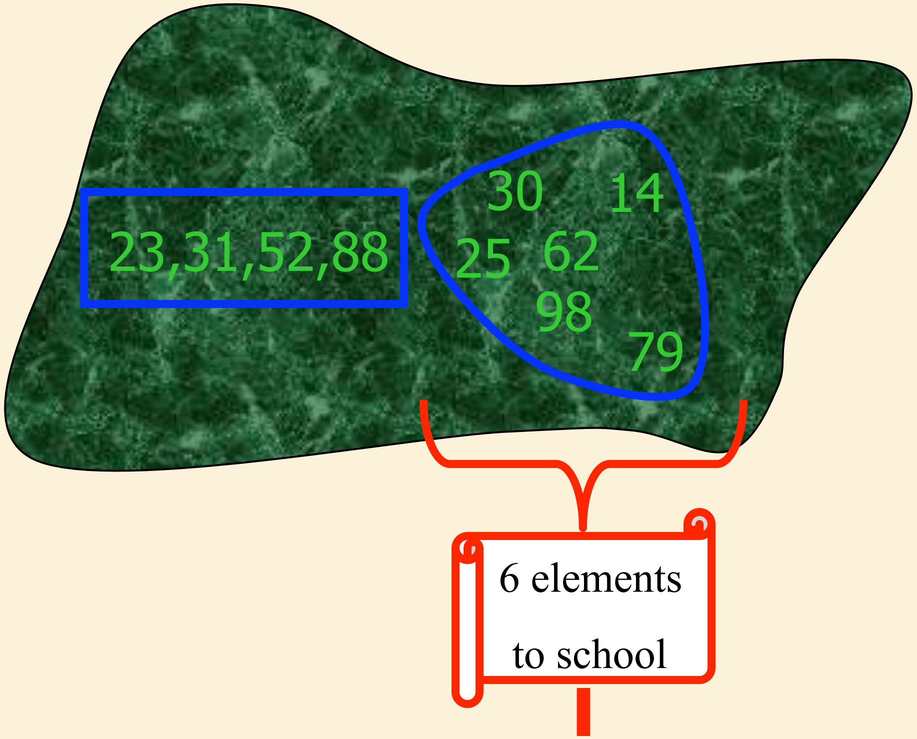
88 52 14
31 25 62 30
98 23
79

14,23,25,30,31,52,62,79,88,98

23,31,52,88

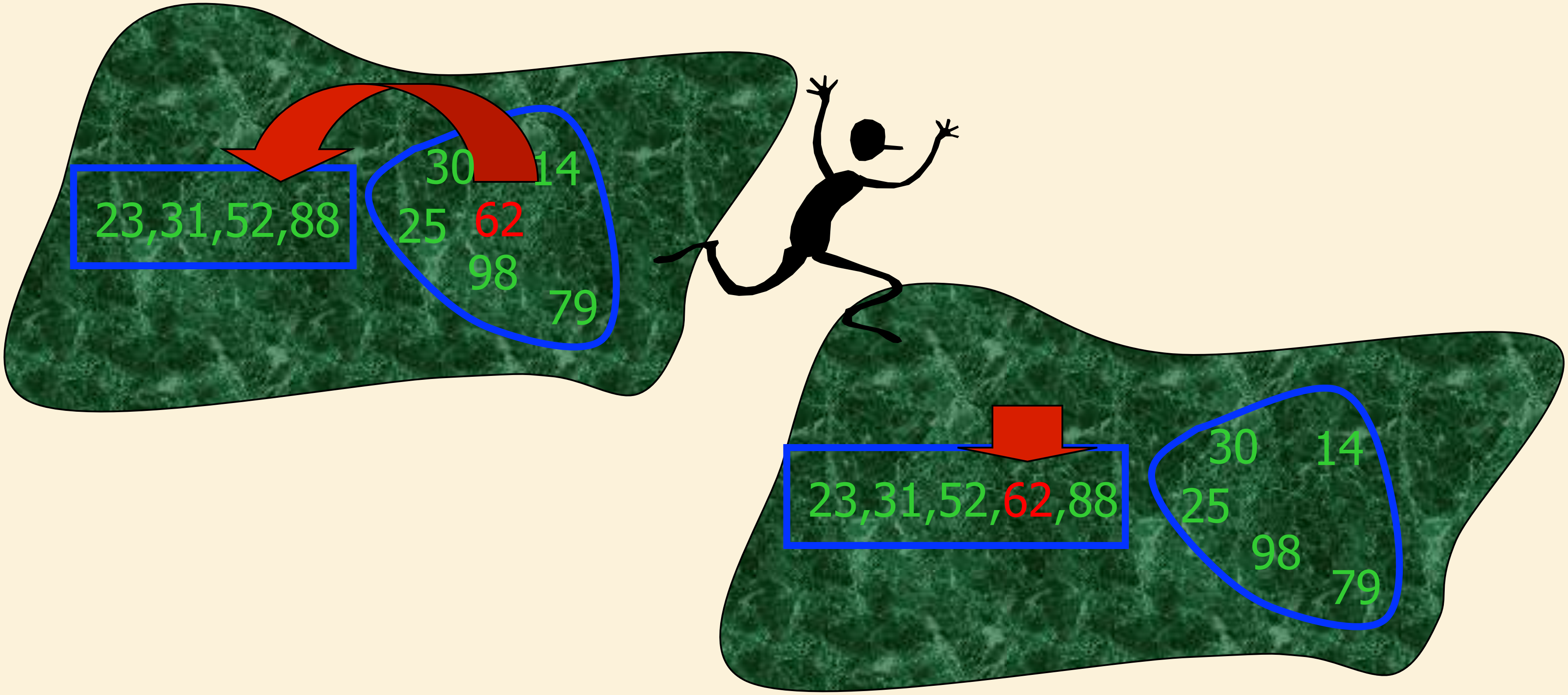
30 14
25 62
98
79

Defining Measure of Progress



Define Step

- Select arbitrary element from side.
- Insert it where it belongs.



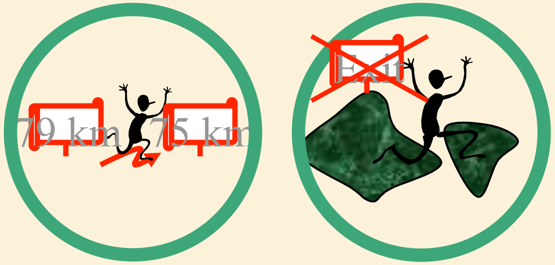
23,31,52,88

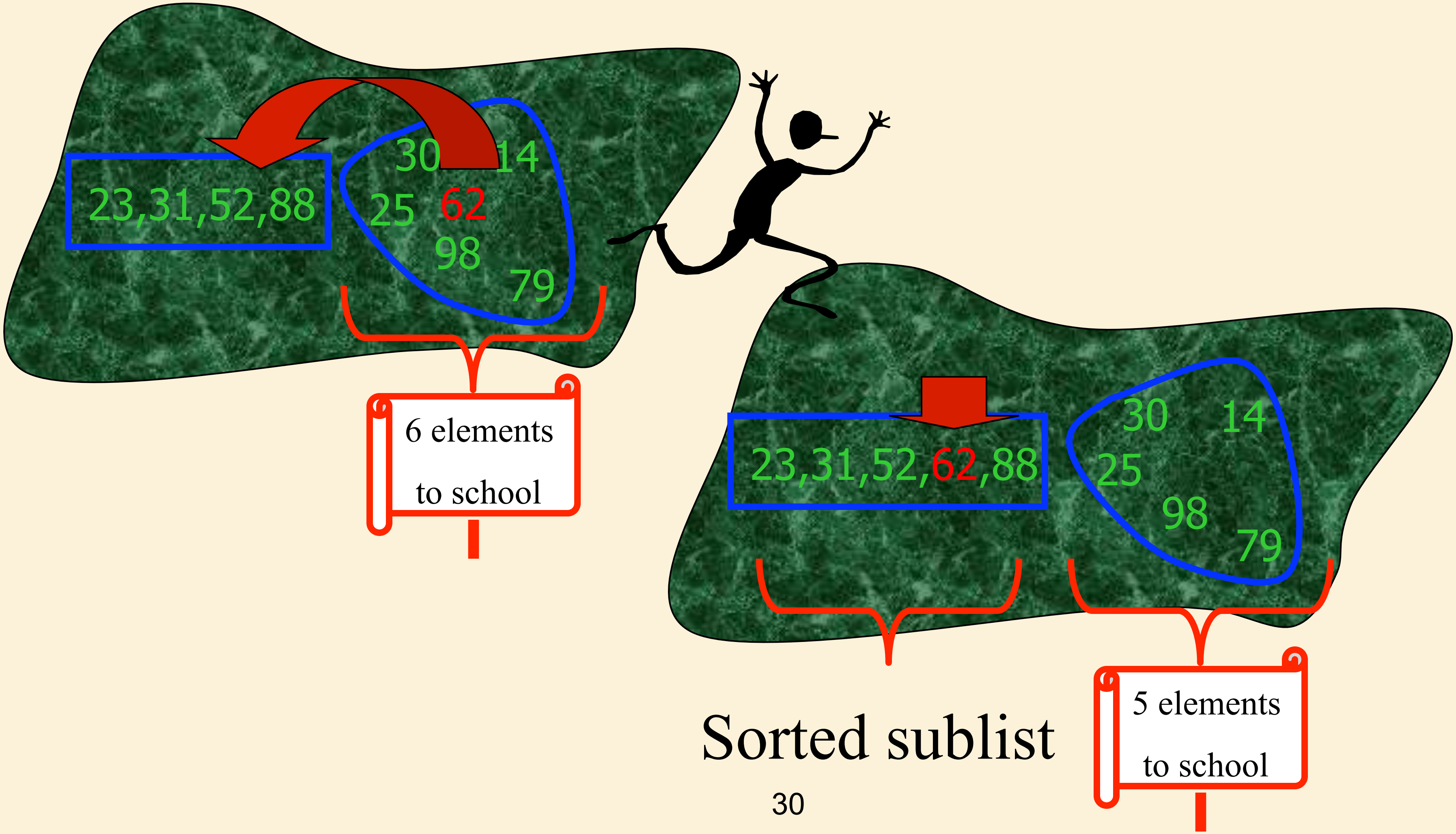
30 14
25 62
98
79

23,31,52,62,88

30 14
25
98
79

Making progress while Maintaining the loop invariant





23,31,52,88

30 14
25 62
98
79

6 elements
to school

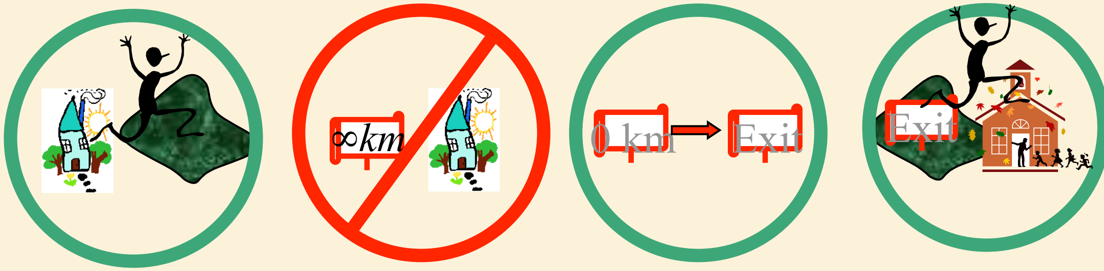
23,31,52,62,88

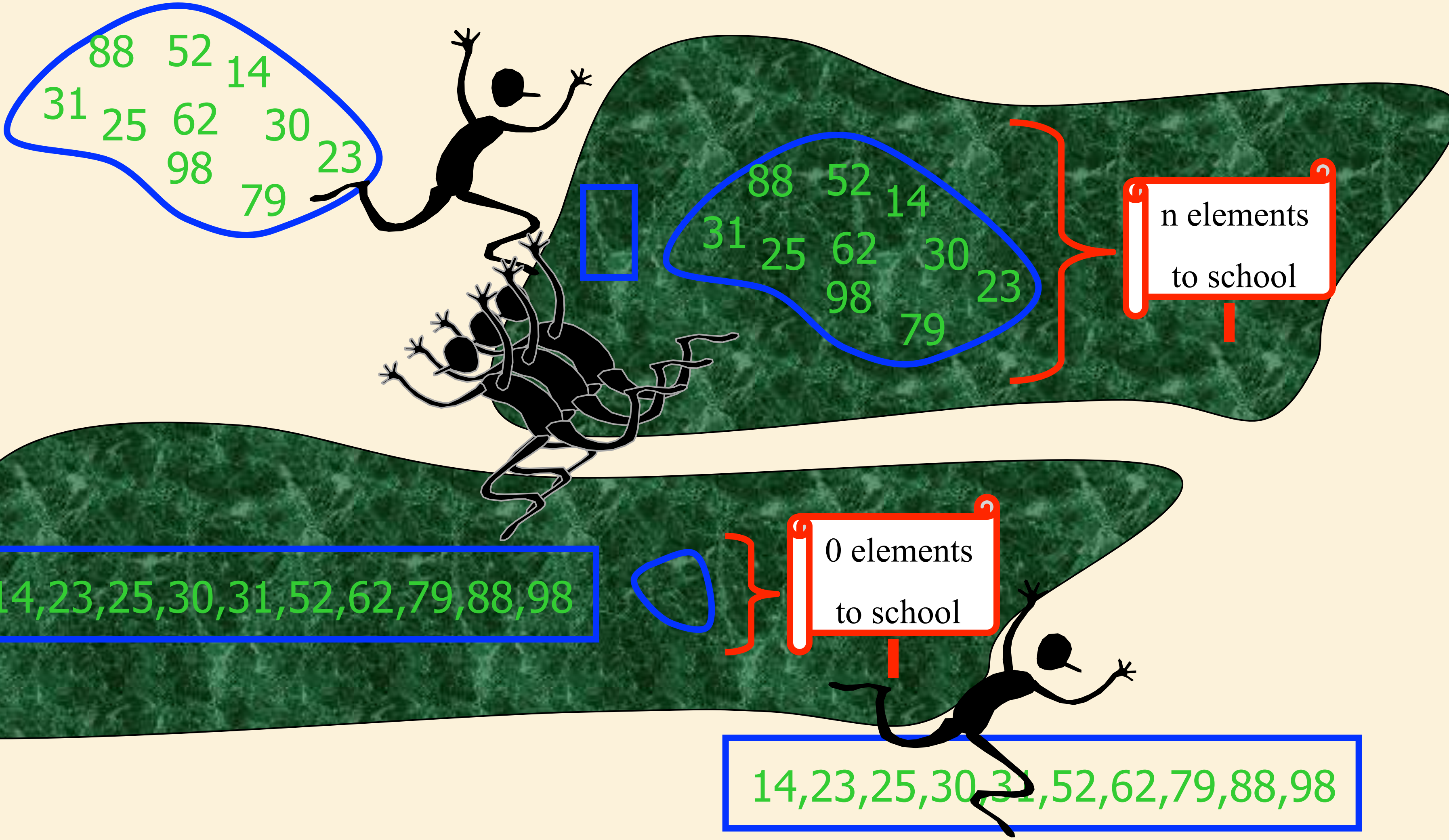
30 14
25
98
79

Sorted sublist

5 elements
to school

Beginning & Ending



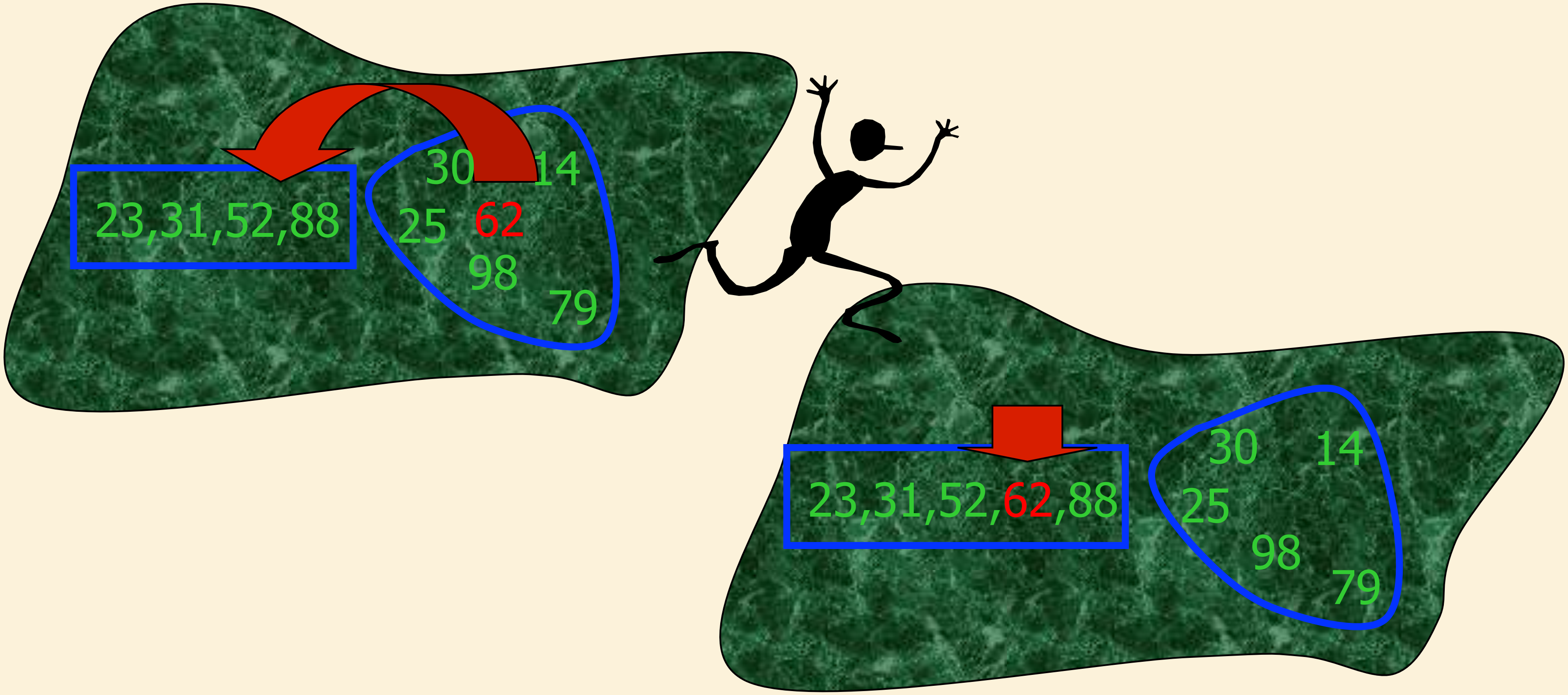


Running Time

Inserting an element into a list of size i

takes $\theta(i)$ time.

Total = $1+2+3+\ldots+n = \theta(n^2)$



Ok
I know you knew Insertion Sort

But hopefully you are beginning to appreciate

Loop Invariants

for describing algorithms

Assertions

in Algorithms

Purpose of Assertions

Useful for

- thinking about algorithms
- developing
- describing
- proving correctness

Definition of Assertions

An assertion is a statement about the current state of the data structure that is either true or false.

eg. the amount in your bank account is not negative.

Definition of Assertions

It is made at some particular point during the execution of an algorithm.

If it is false, then something has gone wrong in the logic of the algorithm.

Definition of Assertions

An assertion is not a task for the algorithm to perform.

It is only a comment that is added for the benefit of the reader.

Specifying a Computational Problem

Example 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..

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.

An Example:
A Sequence of Assertions

```
<assertion$_0$>
if( <condition$_1$> ) then
    code$_{<1,true>}$
else
    code$_{<1,false>}$
end if
<assertion$_1$>
        ⋮
<assertion$_{r-1}$>
if( <condition$_r$> ) then
    code$_{<r,true>}$
else
    code$_{<r,false>}$
end if
<assertion$_r$>
```

Definition of Correctness

<assertion$_0$>
any <conditions> ➡ <assertion$_r$>
code

How is this proved?

Must check 2^r different
- settings of <conditions>
- paths through the code.

Is there a faster way?

An Example:
A Sequence of Assertions

```
<assertion$_0$>
if( <condition$_1$> ) then
    code$_{<1,true>}$
else
    code$_{<1,false>}$
end if
<assertion$_1$>
    :
<assertion$_{r-1}$>
if( <condition$_r$> ) then
    code$_{<r,true>}$
else
    code$_{<r,false>}$
end if
<assertion$_r$>
```

Step 1

$\langle\text{assertion}_0\rangle$
$\langle\text{condition}_1\rangle$
$\text{code}_{\langle 1,\text{true}\rangle}$ $\Rightarrow$ $\langle\text{assertion}_1\rangle$

$\langle\text{assertion}_0\rangle$
$\neg\langle\text{condition}_1\rangle$
$\text{code}_{\langle 1,\text{false}\rangle}$ $\Rightarrow$ $\langle\text{assertion}_1\rangle$

An Example:
A Sequence of Assertions

```
$\langle \text{assertion}_0 \rangle$
if( $\langle \text{condition}_1 \rangle$ ) then
    $\text{code}_{\langle 1,\text{true} \rangle}$
else
    $\text{code}_{\langle 1,\text{false} \rangle}$
end if
$\langle \text{assertion}_1 \rangle$
        ⋮
$\langle \text{assertion}_{r-1} \rangle$
if( $\langle \text{condition}_r \rangle$ ) then
    $\text{code}_{\langle r,\text{true} \rangle}$
else
    $\text{code}_{\langle r,\text{false} \rangle}$
end if
$\langle \text{assertion}_r \rangle$
```

Step 2

$$\begin{array}{l} \langle \text{assertion}_1 \rangle \\ \langle \text{condition}_2 \rangle \\ \text{code}_{\langle 2,\text{true} \rangle} \end{array} \Rightarrow \langle \text{assertion}_2 \rangle$$

$$\begin{array}{l} \langle \text{assertion}_1 \rangle \\ \neg\langle \text{condition}_2 \rangle \\ \text{code}_{\langle 2,\text{false} \rangle} \end{array} \Rightarrow \langle \text{assertion}_2 \rangle$$

An Example:
A Sequence of Assertions

```
$\langle assertion_0 \rangle$
if( $\langle condition_1 \rangle$ ) then
    $code_{\langle 1,true \rangle}$
else
    $code_{\langle 1,false \rangle}$
end if
$\langle assertion_1 \rangle$
    ⋮
$\langle assertion_{r-1} \rangle$
if( $\langle condition_r \rangle$ ) then
    $code_{\langle r,true \rangle}$
else
    $code_{\langle r,false \rangle}$
end if
$\langle assertion_r \rangle$
```

Step r

$$\begin{array}{l} \langle assertion_{r-1} \rangle \\ \langle condition_r \rangle \\ code_{\langle r,true \rangle} \end{array} \Rightarrow \langle assertion_r \rangle$$

$$\begin{array}{l} \langle assertion_{r-1} \rangle \\ \neg\langle condition_r \rangle \\ code_{\langle r,false \rangle} \end{array} \Rightarrow \langle assertion_r \rangle$$

A Sequence of Assertions

```
$\langle \text{assertion}_0 \rangle$
if( $\langle \text{condition}_1 \rangle$ ) then
    $\text{code}_{\langle 1,\text{true} \rangle}$
else
    $\text{code}_{\langle 1,\text{false} \rangle}$
end if
$\langle \text{assertion}_1 \rangle$
    ⋮
$\langle \text{assertion}_{r-1} \rangle$
if( $\langle \text{condition}_r \rangle$ ) then
    $\text{code}_{\langle r,\text{true} \rangle}$
else
    $\text{code}_{\langle r,\text{false} \rangle}$
end if
$\langle \text{assertion}_r \rangle$
```

Step r

$\langle \text{assertion}_{r-1} \rangle$
$\langle \text{condition}_r \rangle$ → $\langle \text{assertion}_r \rangle$
$\text{code}_{\langle r,\text{true} \rangle}$

$\langle \text{assertion}_{r-1} \rangle$
$\neg\langle \text{condition}_r \rangle$ → $\langle \text{assertion}_r \rangle$
$\text{code}_{\langle r,\text{false} \rangle}$

Another Example: A Loop

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Type of Algorithm:

- Iterative

Type of Assertion:

- Loop Invariants

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Definition of Correctness?

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Definition of Correctness

<preCond>
any <conditions> ⇒ <postCond>
code

How is this proved?

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Definition of Correctness

<preCond>
any <conditions>
code
⇒ <postCond>

The computation may go around the loop an arbitrary number of times.

Is there a faster way?

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Step 0

<preCond>
codeA

⇒

<loop-invariant

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Step 1

<loop-invariant>
¬<exit Cond>
codeB

⇒ <loop-invariant

Iterative Algorithms
Loop Invariants

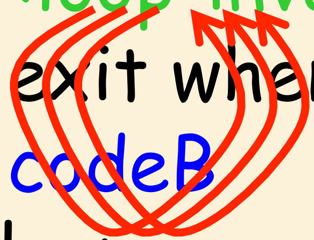
```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Step 2

<loop-invariant>
¬<exit Cond>
codeB

→ <loop-invariant

Iterative Algorithms
Loop Invariants



```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Step 3

<loop-invariant>
¬<exit Cond>
codeB

→ <loop-invariant

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Step i

<loop-invariant>
¬<exit Cond>
codeB ⇒ <loop-invariant

All these steps are the same
and therefore only need be done once!

Iterative Algorithms
Loop Invariants

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

Last Step

<loop-invariant>
<exit Cond>
codeC
⇒ <postCond>

Partial Correctness

Establishing Loop Invariant

<preCond>
codeA
⇒ <loop-invariant>

Maintaining Loop Invariant

<loop-invariant>
¬<exit Cond>
codeB
⇒ <loop-invariant>

Clean up loose ends

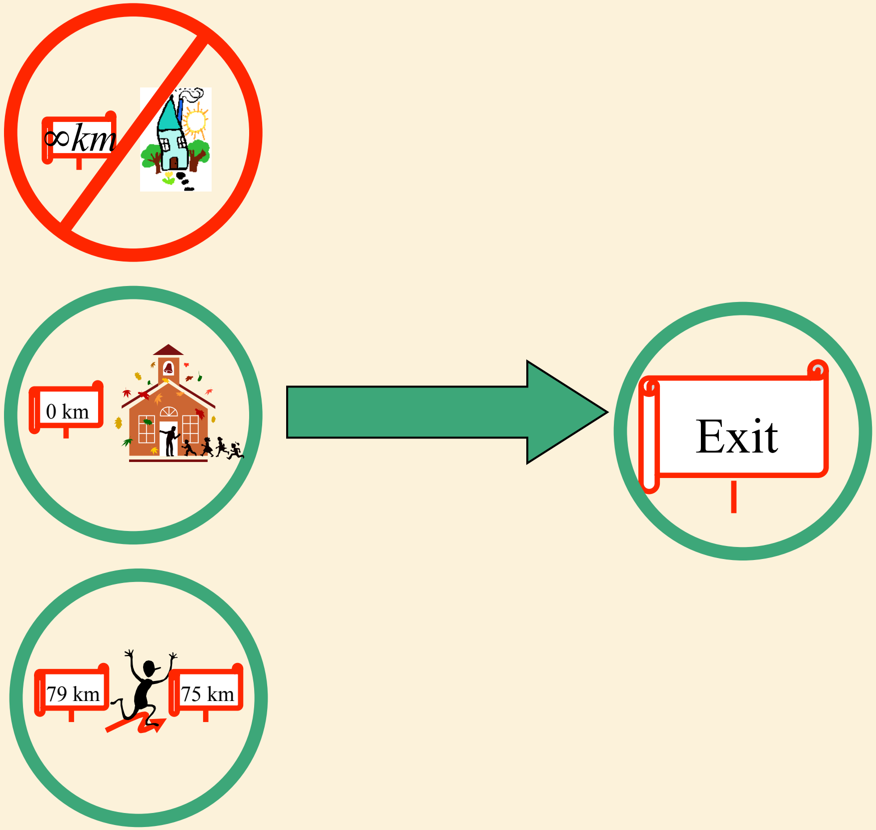
<loop-invariant>
<exit Cond>
codeC
⇒ <postCond>

Proves that IF the program terminates then it works

<PreCond> & <code> ⇒ <PostCond>

Algorithm Termination

Measure of progress



Algorithm Correctness

Partial Correctness
+ Termination → Correctness

Designing Loop Invariants

Coming up with the loop invariant is the hardest part of designing an algorithm.

It requires practice, perseverance, and insight.

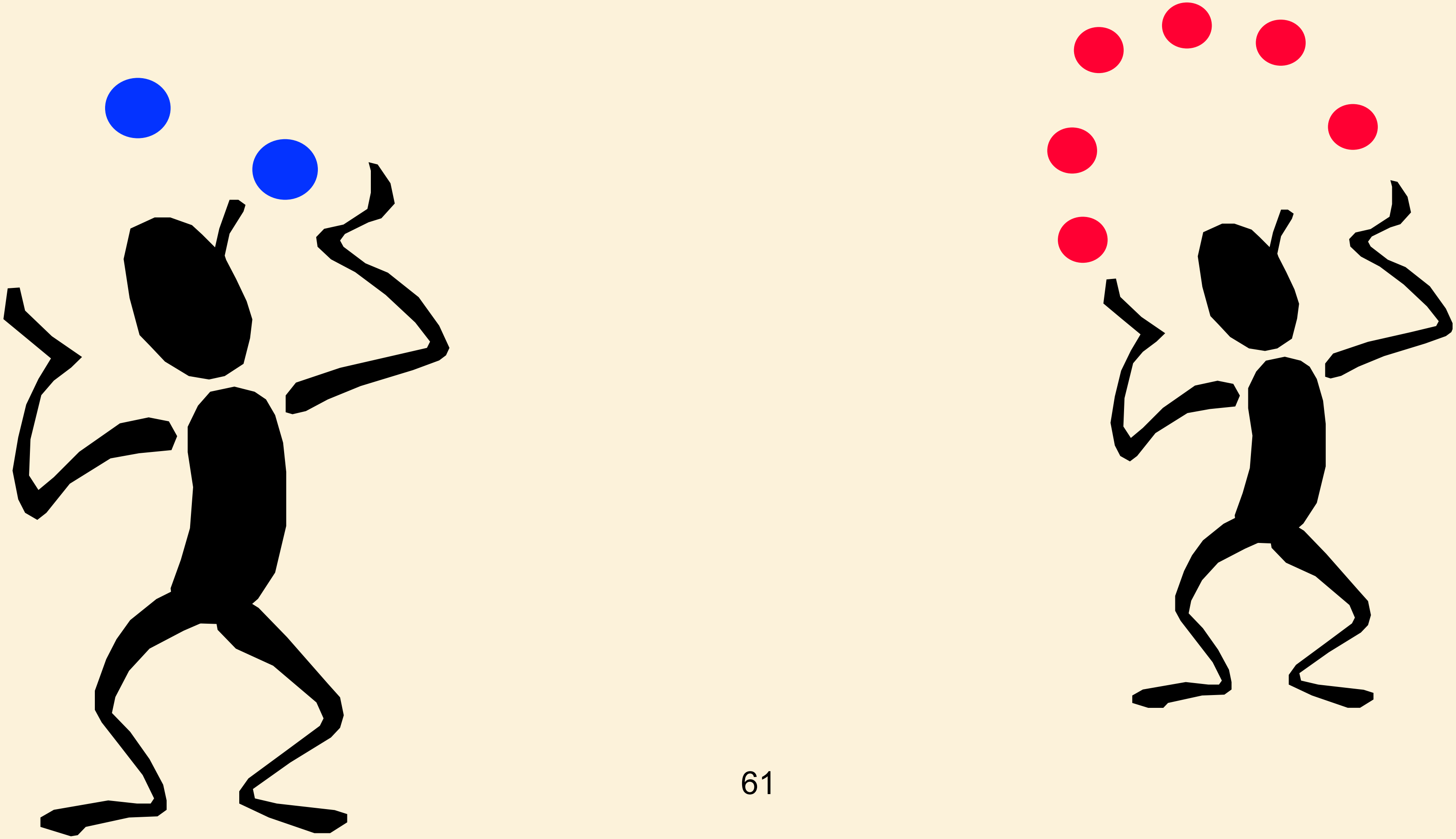
Yet from it
the rest of the algorithm
follows easily

Don’t start coding

You must design a working algorithm first.

Exemplification:

Try solving the problem on small input examples.



Start with Small Steps

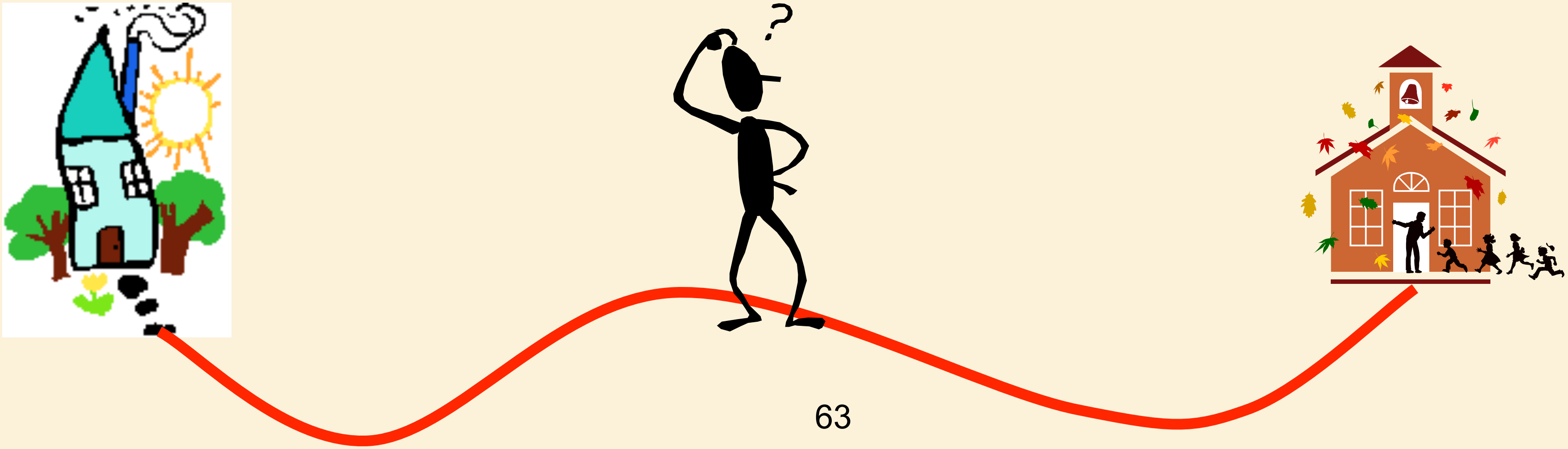
What basic steps might you follow to make some kind of progress towards the answer?

Describe or draw a picture of what the data structure might look like after a number of these steps.

Picture from the Middle

Leap into the middle of the algorithm.

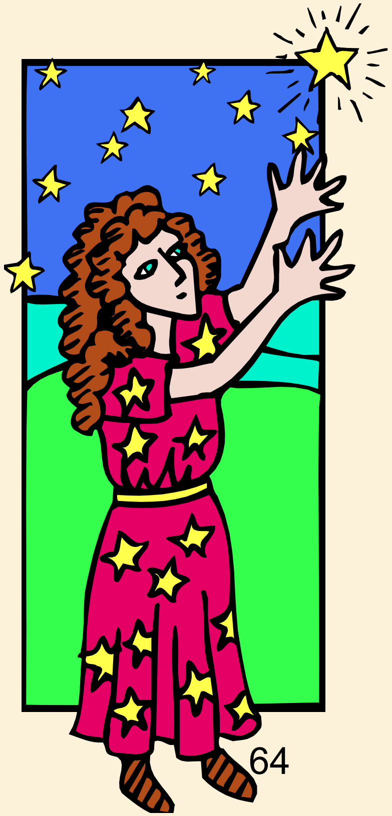
What would you like your data structure to look like when you are half done?



Ask for 100%

Pretend that a genie has granted your wish.

- You are now in the middle of your computation and your dream loop invariant is true.



Ask for 100%

Maintain the Loop Invariant:

- From here, are you able to take some computational steps that will make progress while maintaining the loop invariant?

Ask for 100%

- If you can maintain the loop invariant, great.
- If not,
  - Too Weak: If your loop invariant is too weak, then the genie has not provided you with everything you need to move on.
  - Too Strong: If your loop invariant is too strong, then you will not be able to establish it initially or maintain it.

Differentiating between Iterations

$x=x+2$

- Meaningful as code
- False as a mathematical statement

$x' = x_i$ = value at the beginning of the iteration

$x'' = x_{i+1}$ = new value after going around the loop one more time.

$x'' = x'+2$

- Meaningful as a mathematical statement

Loop Invariants
for
Iterative Algorithms

Three

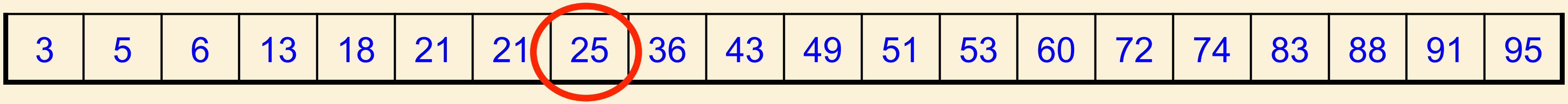
Search Examples

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).

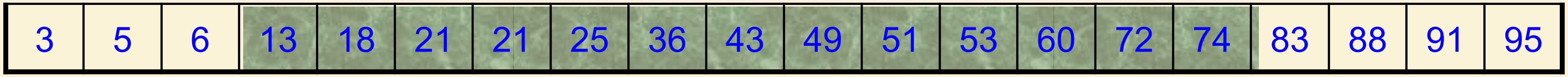


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

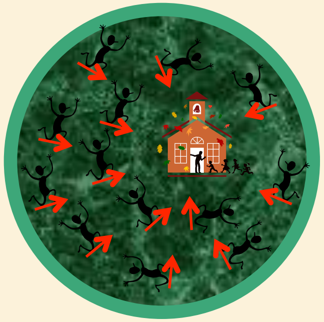
Define Loop Invariant

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

key 25



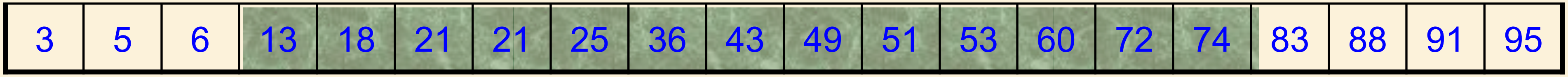
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95



Define Step

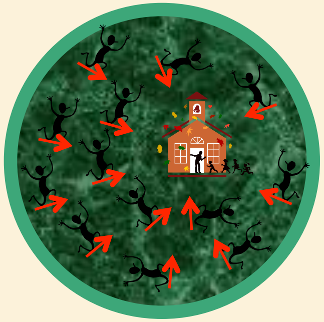
- Make Progress
- Maintain Loop Invariant

key 25

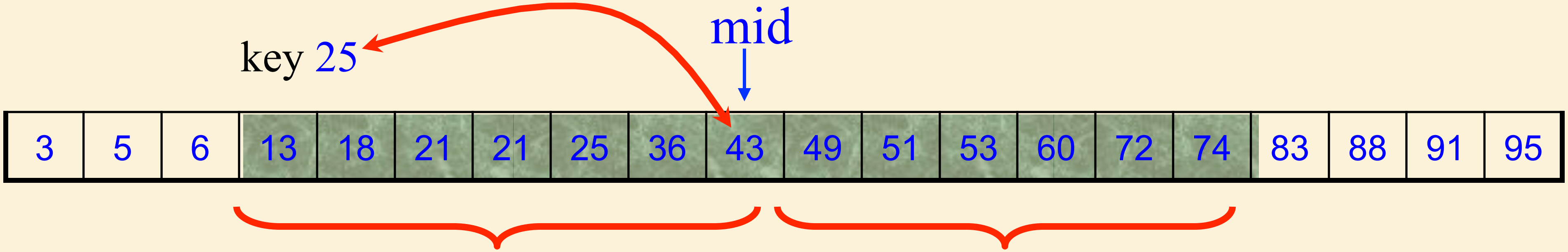


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Step



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

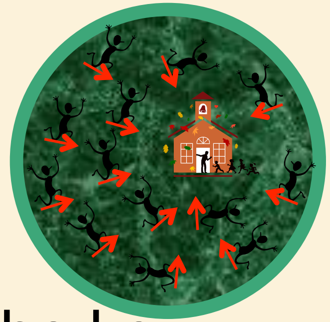


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

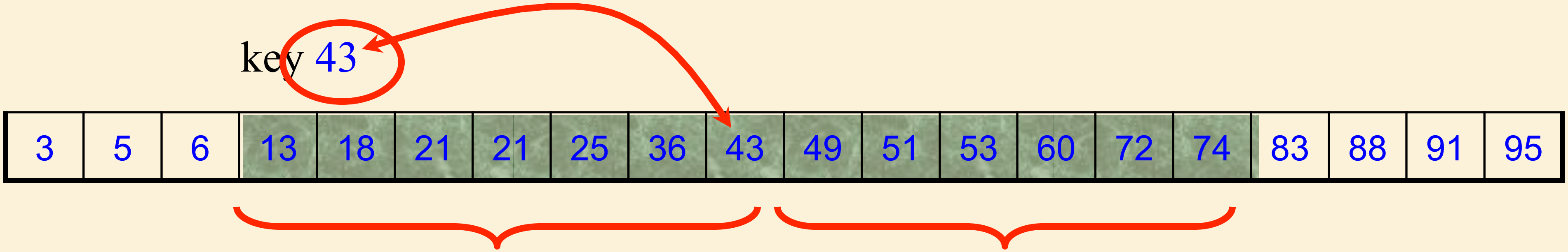
If key ≤ mid,
then key is in
left half.

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

Define Step



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



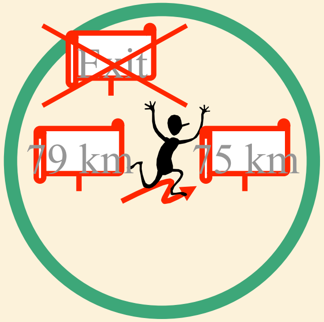
key 43

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

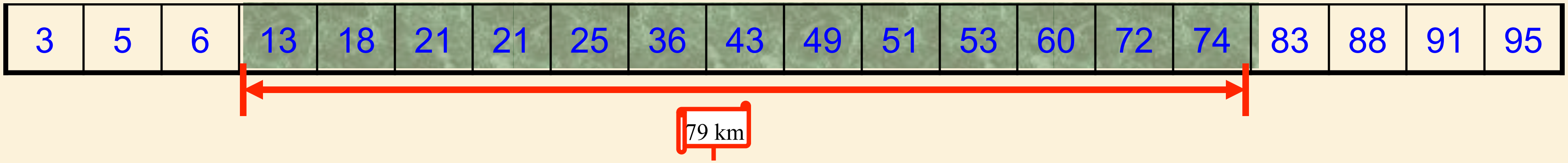
If key $\leq$ mid,
then key is in
left half.

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

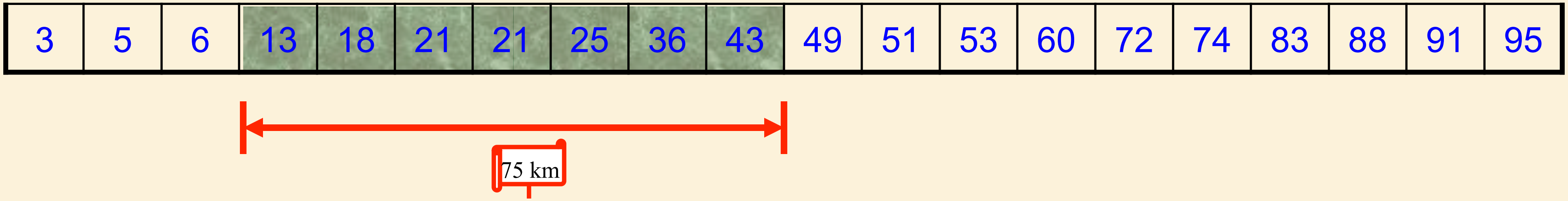
Make Progress



- The size of the list becomes smaller.

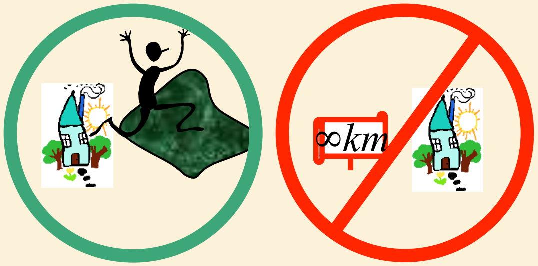


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95



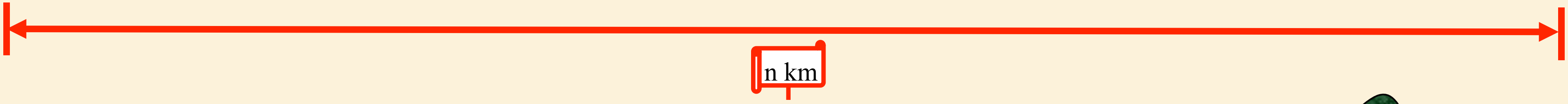
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Initial Conditions



key 25

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



- The sublist is the entire original list.

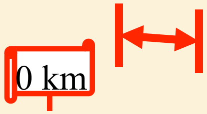
- If the key is contained in the original list,

then the key is contained in the sublist.

Ending Algorithm

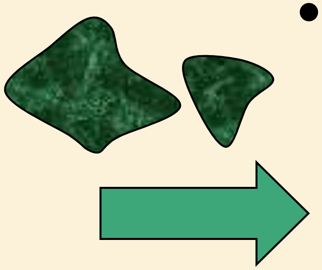
key 25

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 the key is contained in the original list,

  then the key is at this location.

If key not in original list

- If the key is contained in the original list,

  then the key is contained in the sublist.

- Loop invariant true, even if the key is not in the list.

key 24

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 at this location.

- Conclusion still solves the problem.

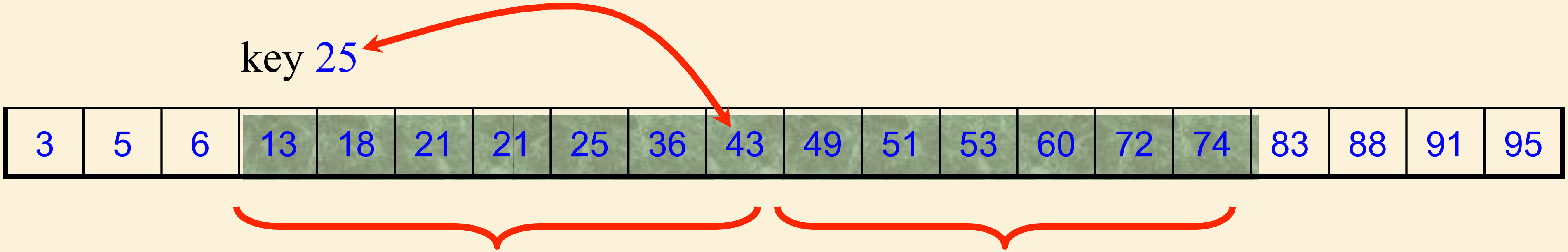
  Simply check this one location for the key.

Running Time

The sublist is of size n, $^{n}/_{2}$, $^{n}/_{4}$, $^{n}/_{8}$,…,1

Each step $\theta(1)$ time.

Total = $\theta(\log n)$

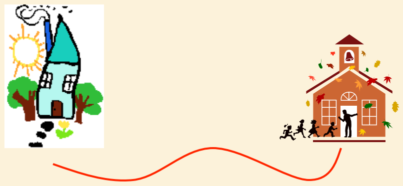
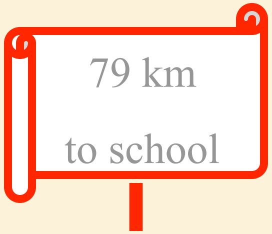
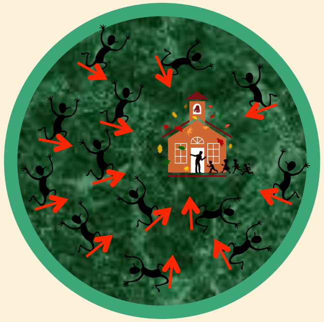
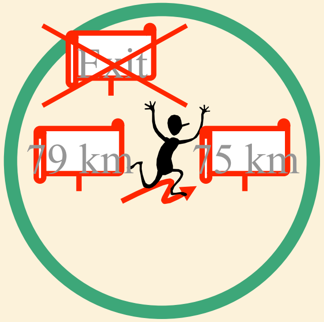
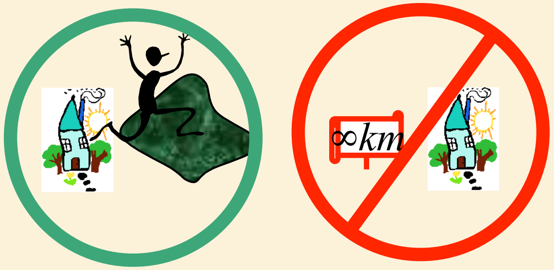
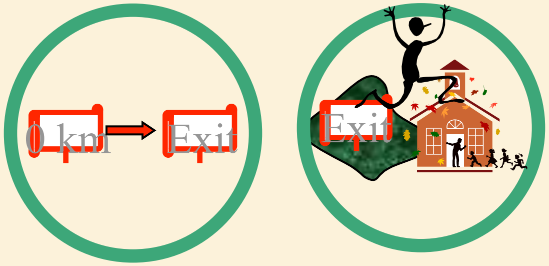


If key ≤ mid,
then key is in
left half.

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

```
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
```

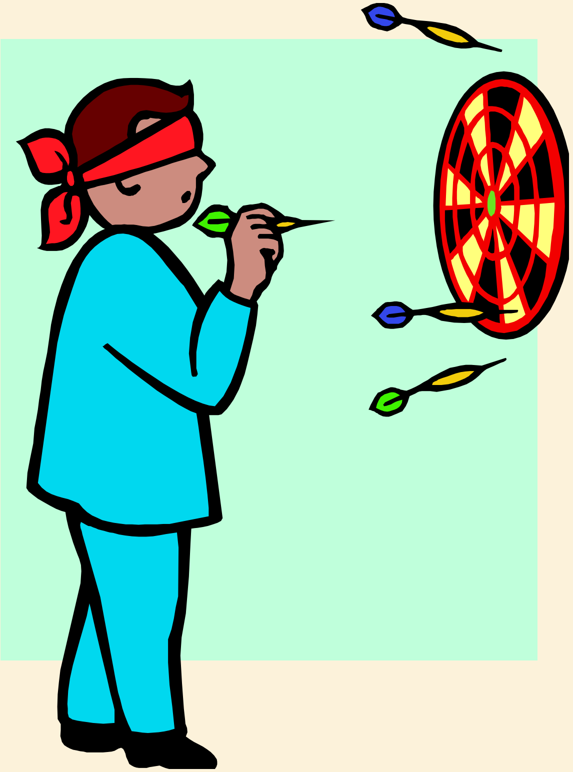
Algorithm Definition Completed

Define Problem 	Define Loop Invariants 	Define Measure of Progress 
Define Step 	Define Exit Condition  	Maintain Loop Inv 
Make Progress  	Initial Conditions  	Ending 

```
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?



The Devil in the Details

- 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*.

The Devil in the Details

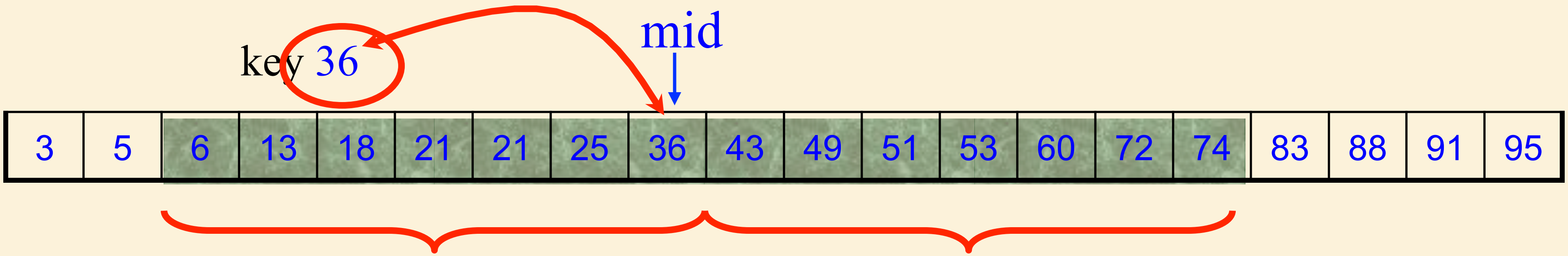
```
if $key \le A[mid]$
    $q = mid$
else
    $p = mid + 1$
end
```

or

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

What condition will break the loop invariant?

The Devil in the Details



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

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

Bug!!

The Devil in the Details

```
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~~

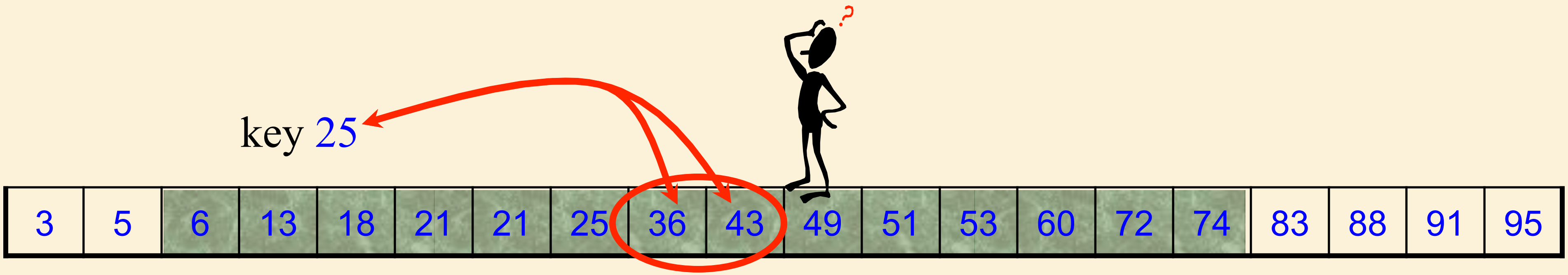
~~$p = mid + 1$~~

~~end~~

Not OK!!

The Devil in the Details

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

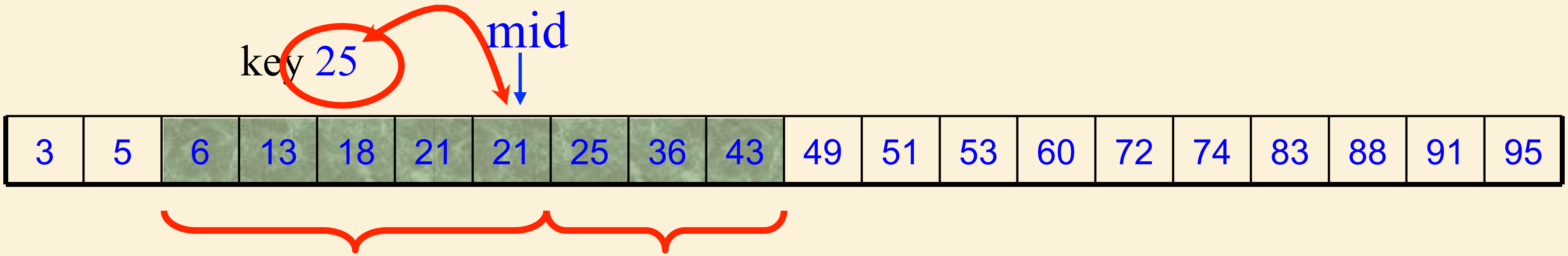


key 25

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

The Devil in the Details

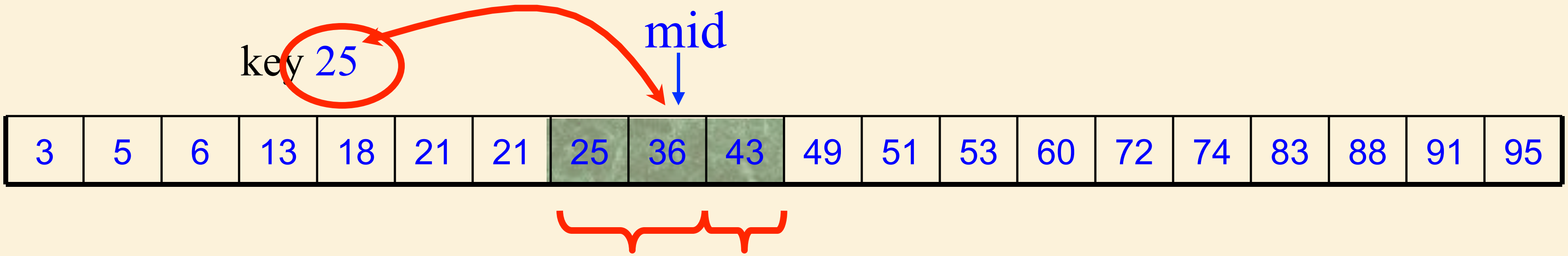


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

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

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

The Devil in the Details

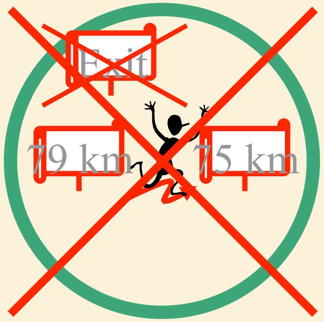


3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

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

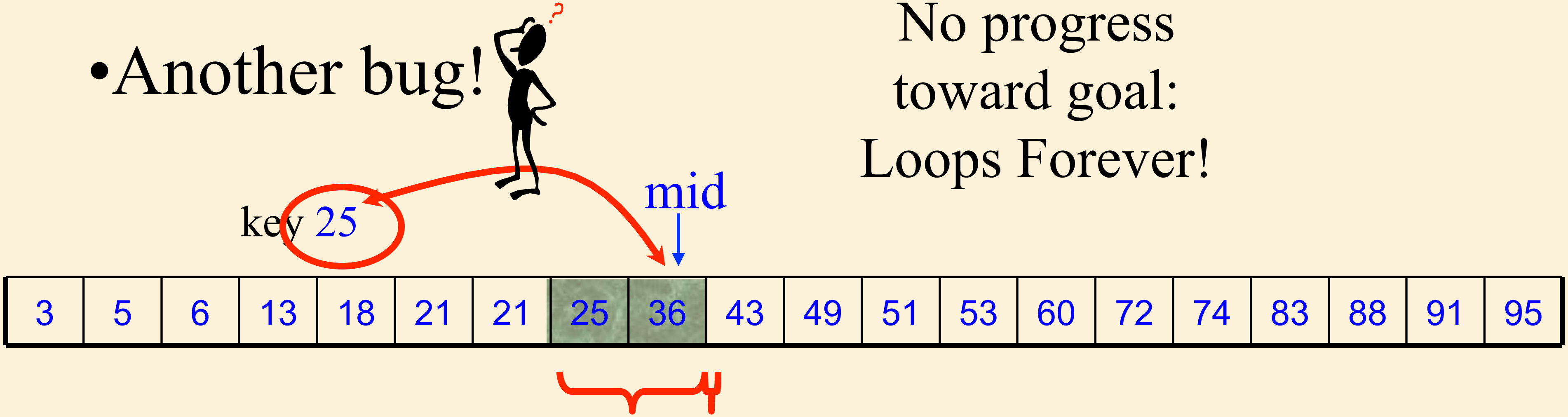
If $key > mid$, then key is in right half.

The Devil in the Details



- Another bug!

No progress
toward goal:
Loops Forever!



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.

The Devil in the Details

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

OK

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

OK

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

Not OK!!

How Many Possible Algorithms?

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

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

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]
    $mid = \left\lfloor \frac{p+q}{2} \right\rfloor$
    if $key = A[mid]$
        return(mid)
    elseif $key < A[mid]$
        $q = mid - 1$
    else
        $p = mid + 1$
    end
end
if $key = A[p]$
    return($p$)
else
    return("Key not in list")
end
```

Moral

- Use the loop invariant method to think about algorithms.
- Be careful with your definitions.
- Be sure that the loop invariant is always maintained.
- Be sure progress is always made.
- Having checked the ‘typical’ cases, pay particular attention to boundary conditions and the end game.

Loop Invariants
for
Iterative Algorithms

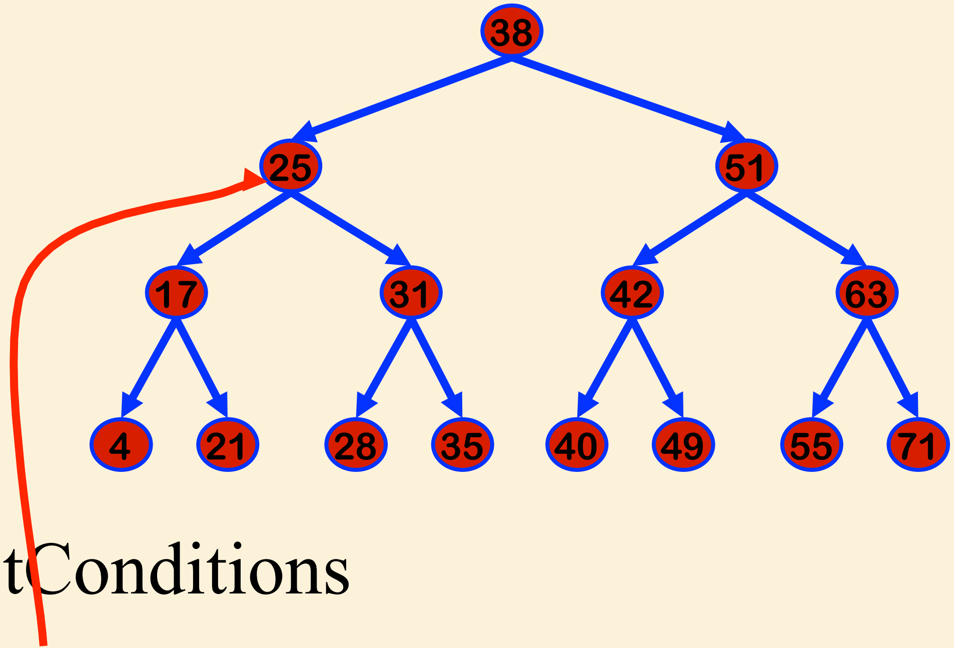
A Second

Search Example:

The Binary Search Tree

Define Problem: Binary Search Tree

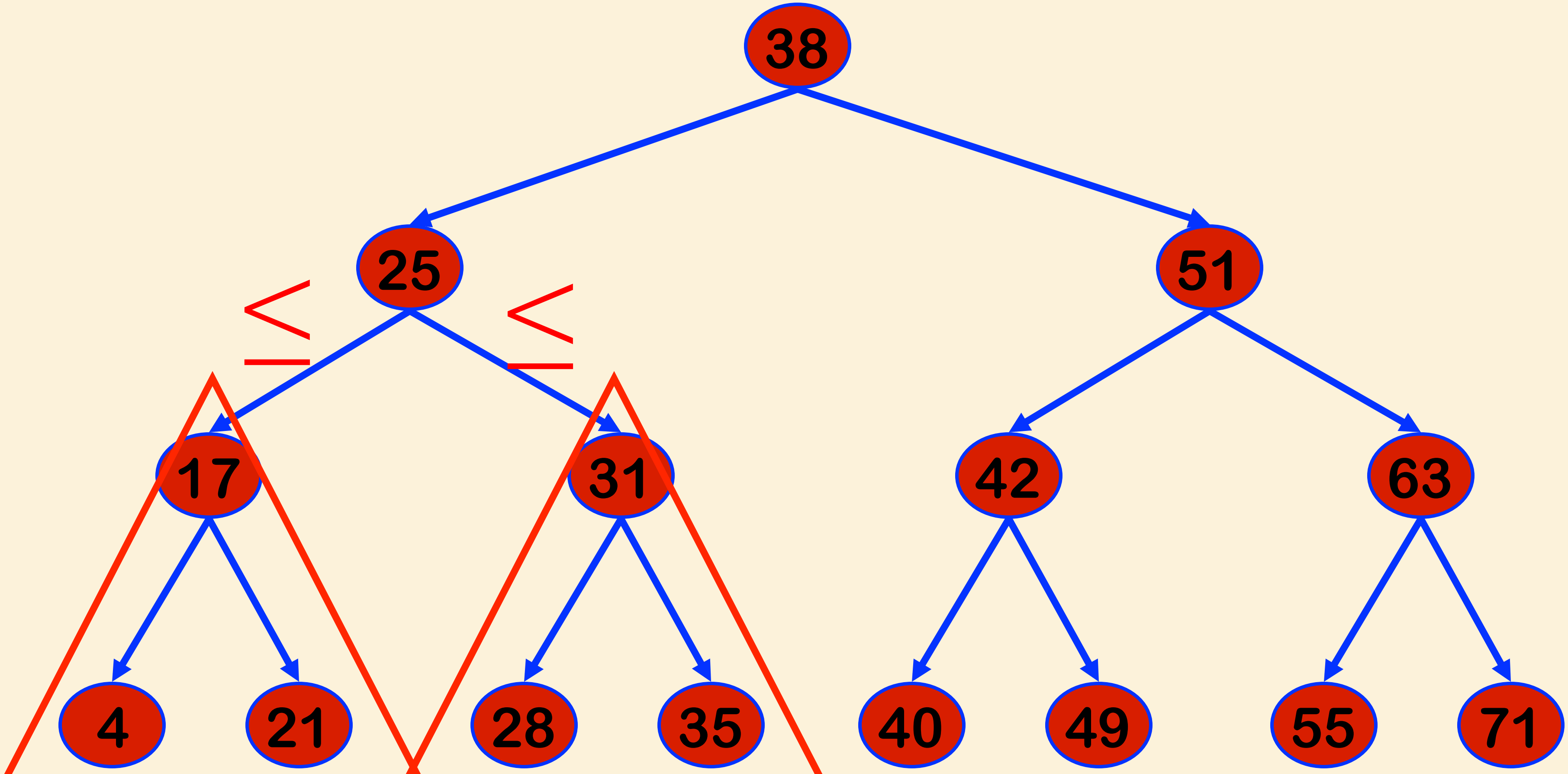
- PreConditions
  - Key 25
  - A binary search tree.



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

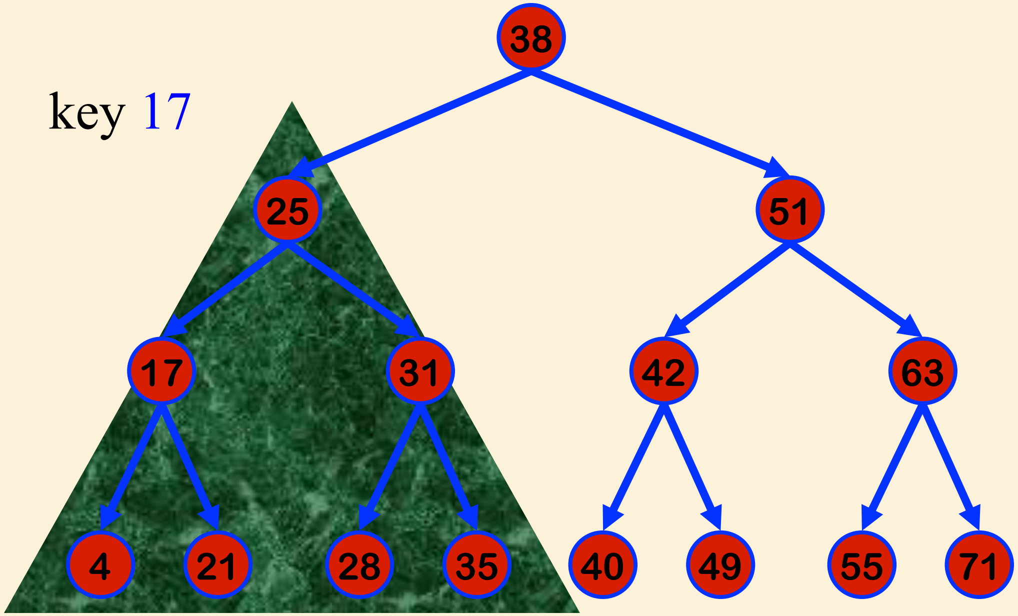
Binary Search Tree

All nodes in left subtree ≤ Any node ≤ All nodes in right subtree

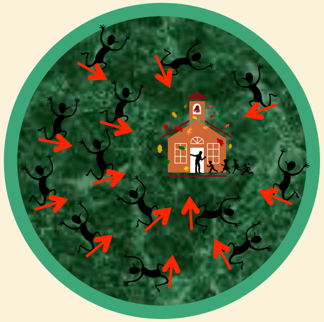


Define Loop Invariant

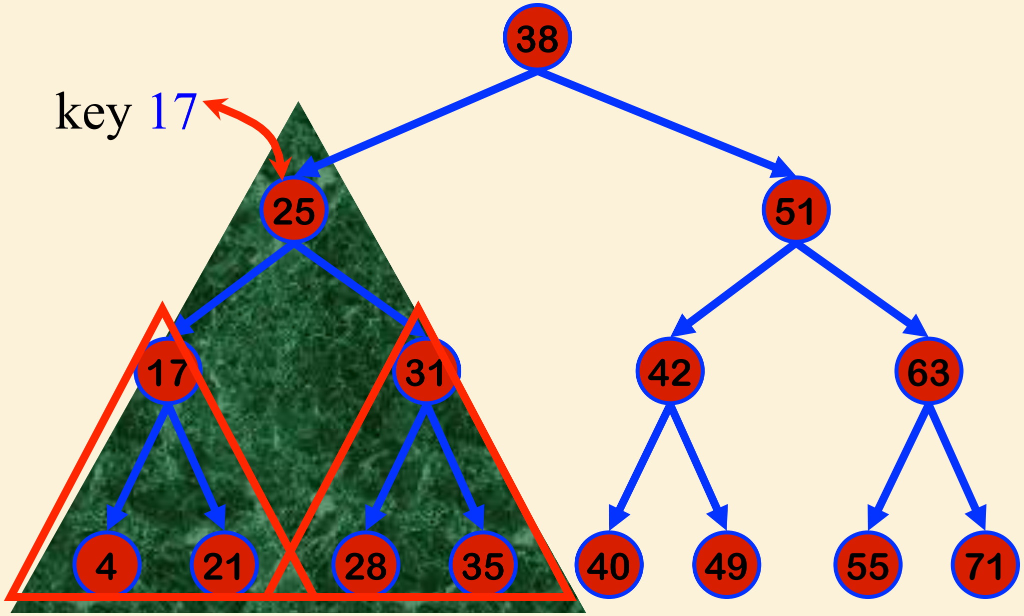
- Maintain a sub-tree.
- If the key is contained in the original tree, then the key is contained in the sub-tree.



Define Step



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



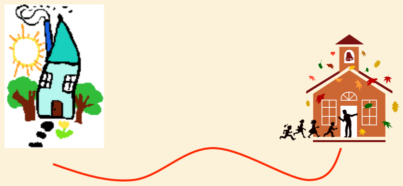
If key < root,
then key is
in left half.

If key = root,
then key is
found

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

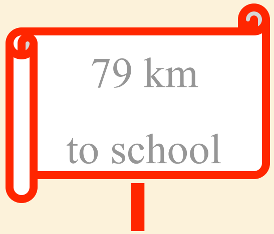
Algorithm Definition Completed

Define Problem

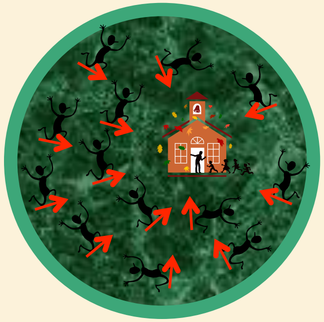


Define Loop Invariants

Define Measure of Progress



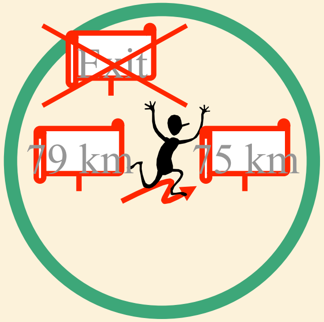
Define Step



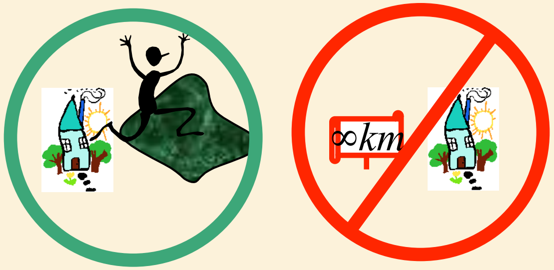
Define Exit Condition

Maintain Loop Inv

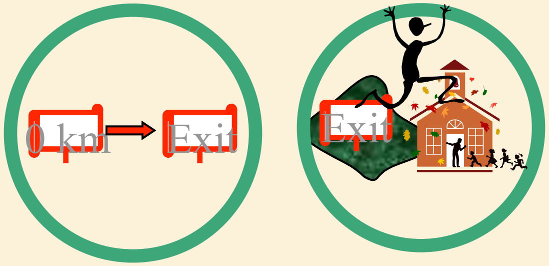
Make Progress



Initial Conditions



Ending



Card Trick

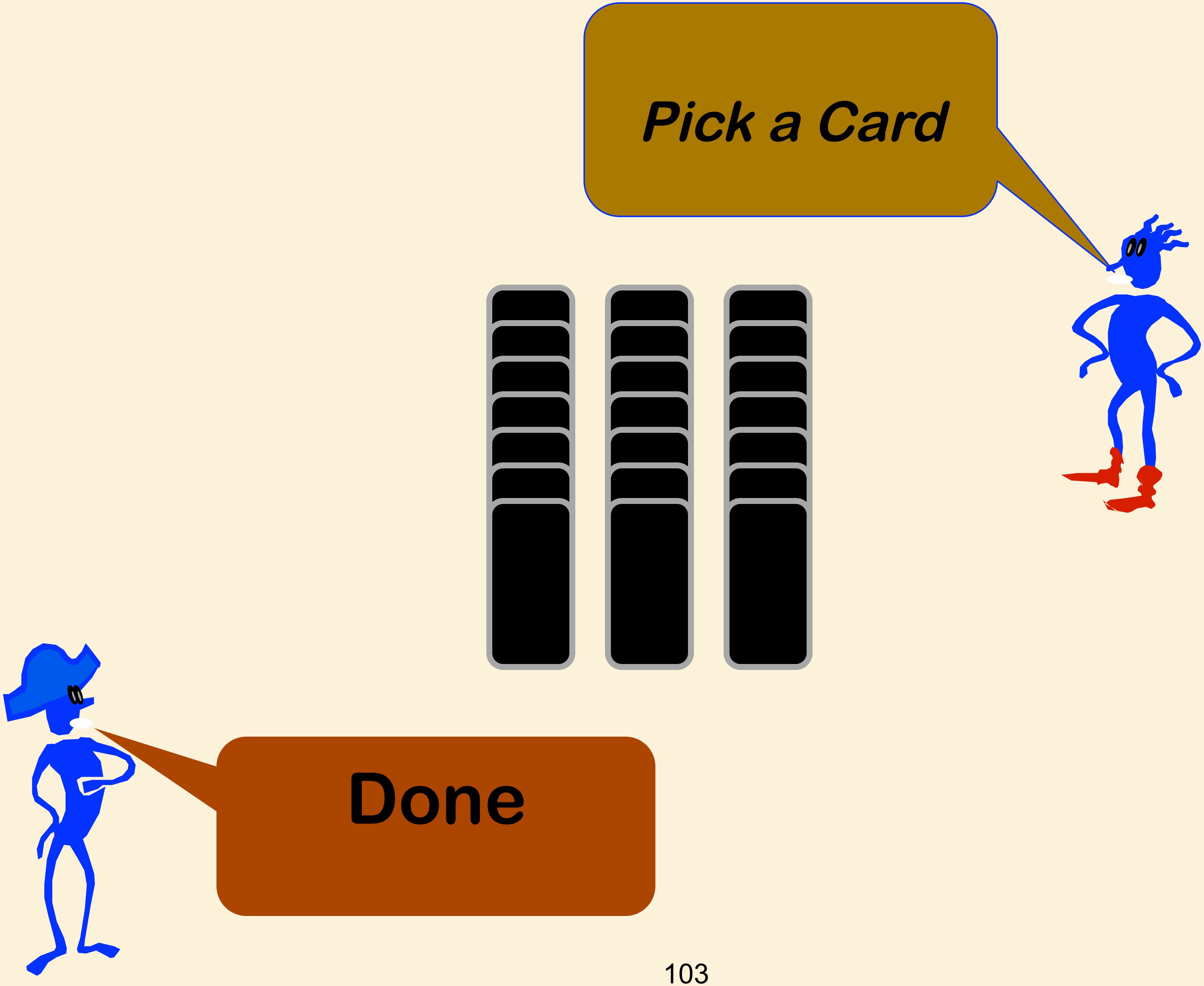
- A volunteer, please.

Loop Invariants
for
Iterative Algorithms

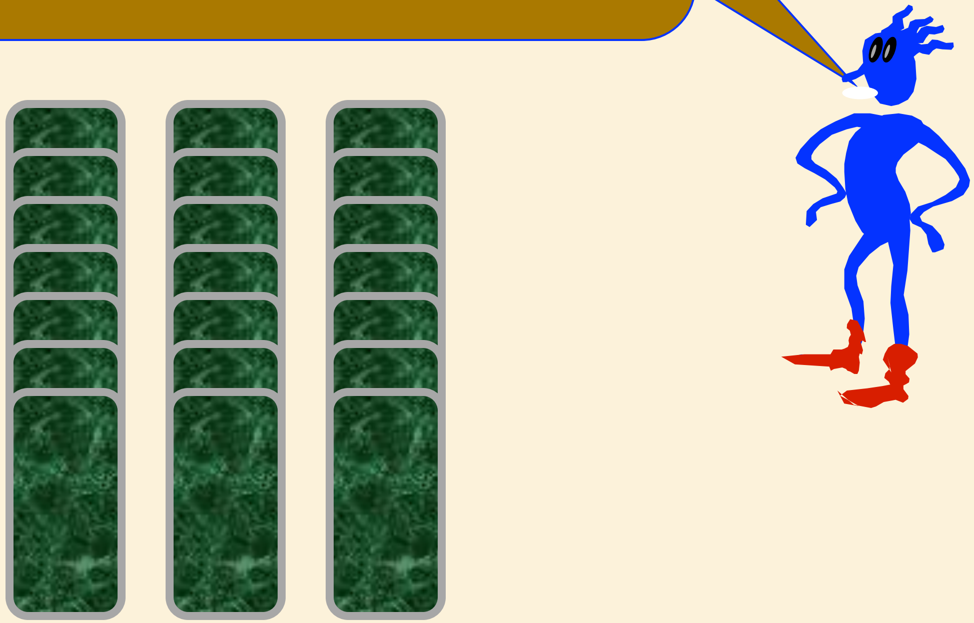
A Third

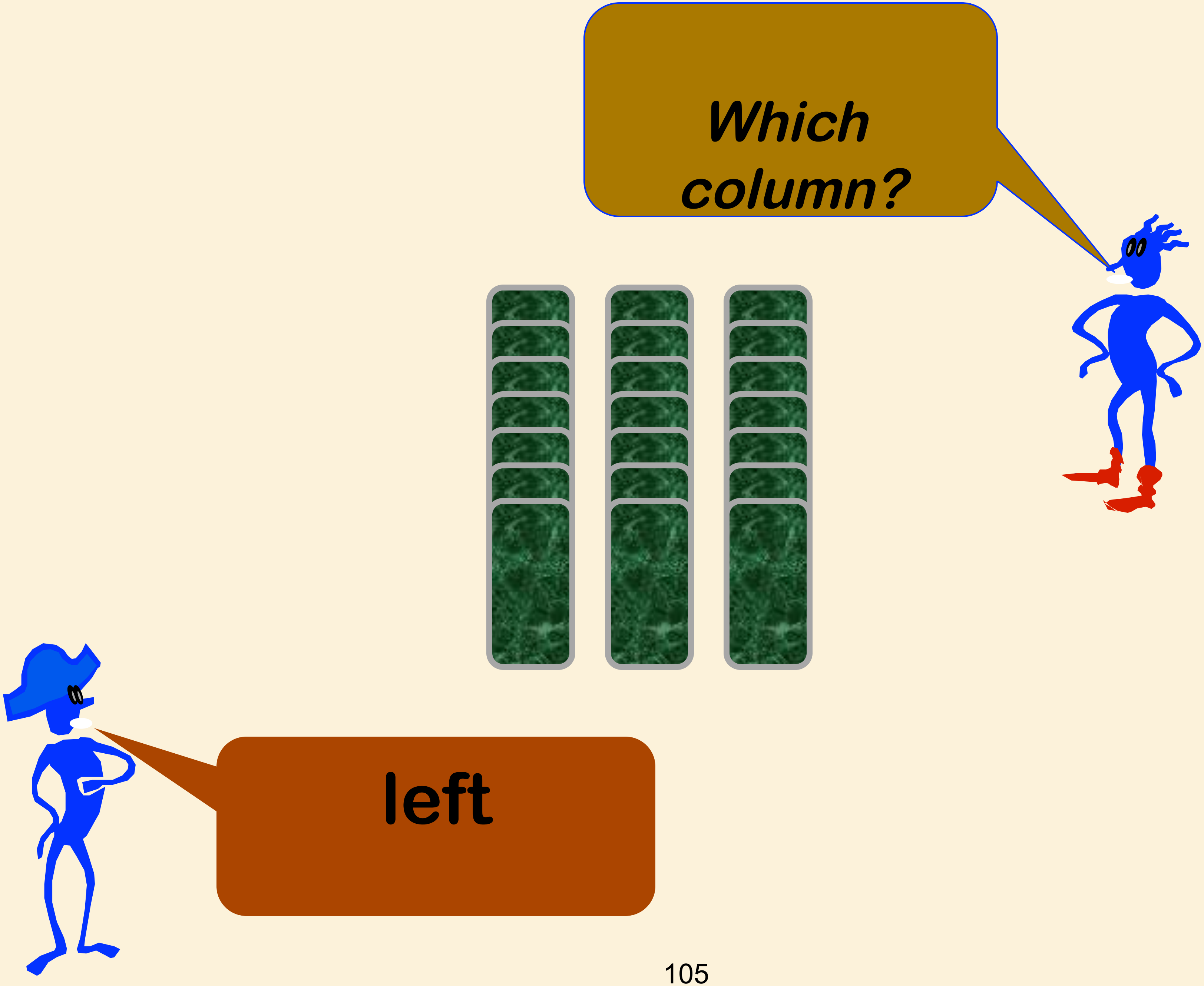
Search Example:

A Card Trick

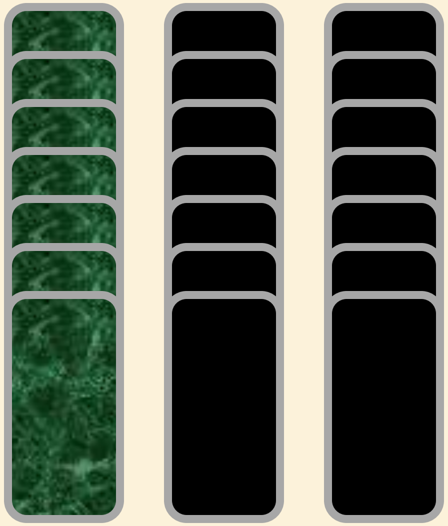

Pick a Card
Done

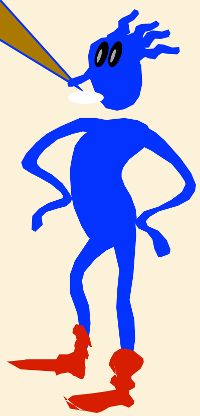
Loop Invariant:
The selected card is one of these.



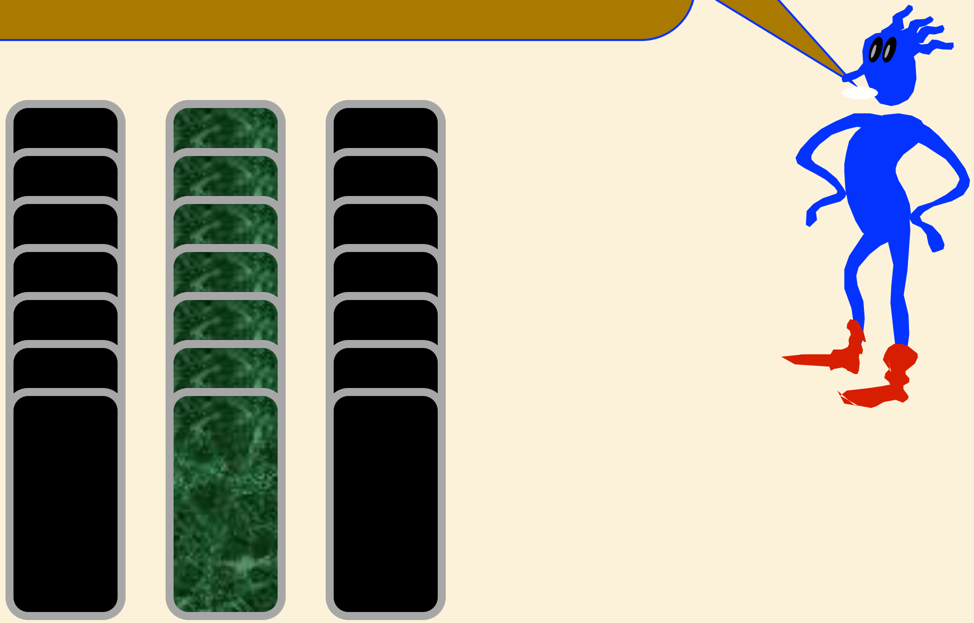

Which column?
left

Loop Invariant:
The selected card is one of these.

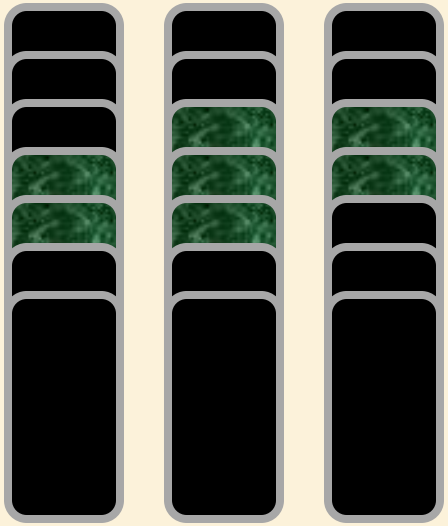


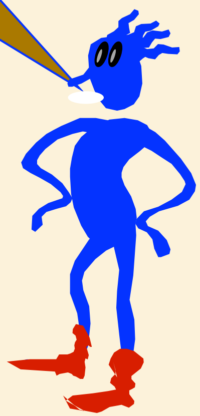


Selected column is placed in the middle

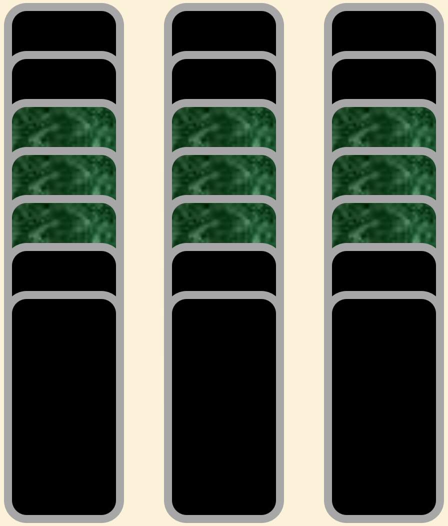


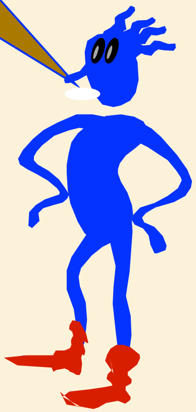
I will rearrange the cards

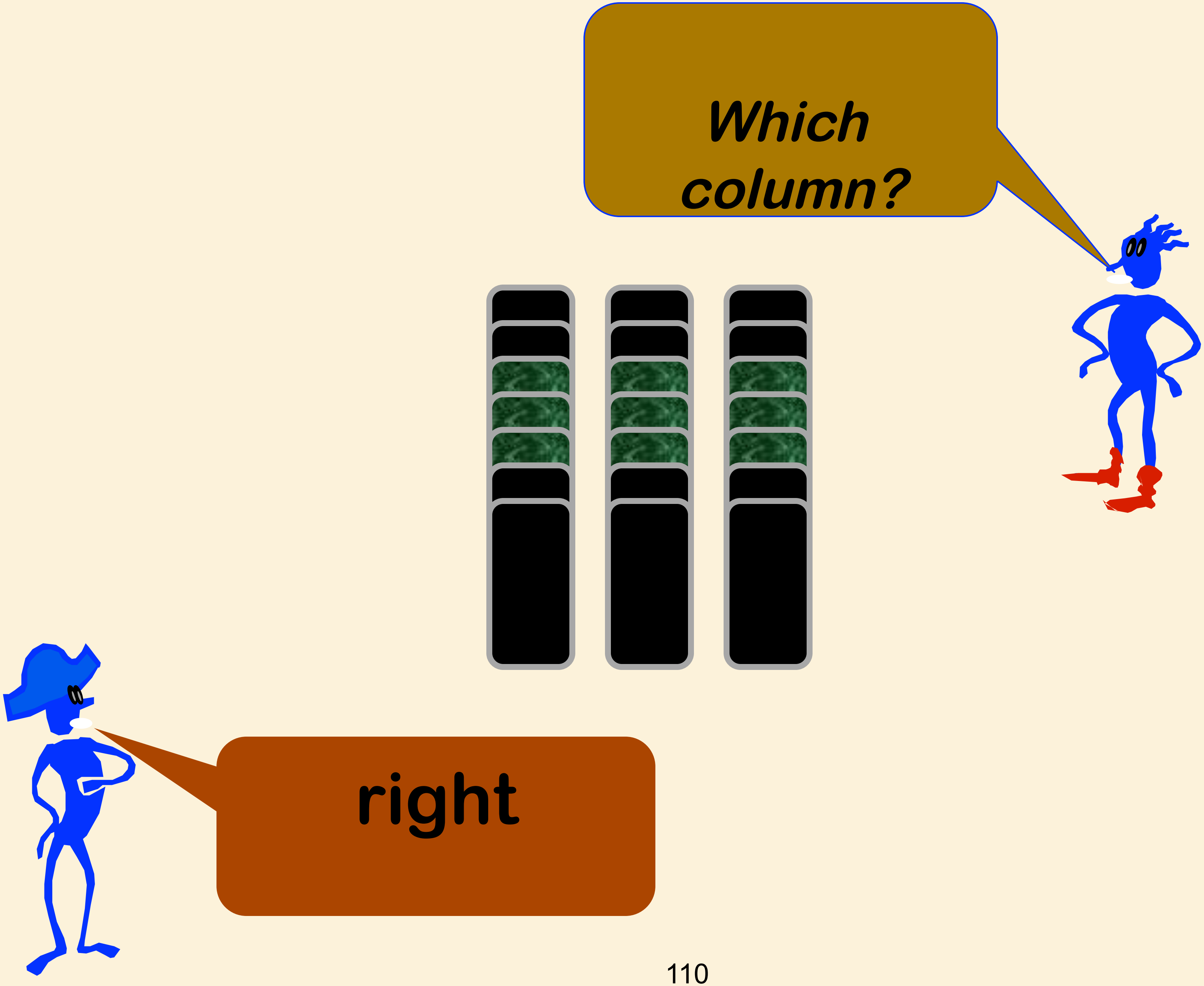




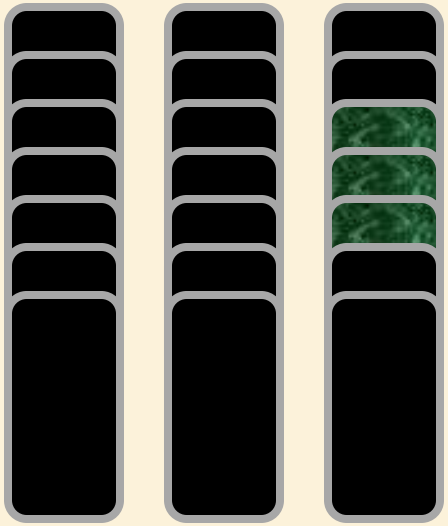
Relax Loop Invariant:
I will remember the same about each column.

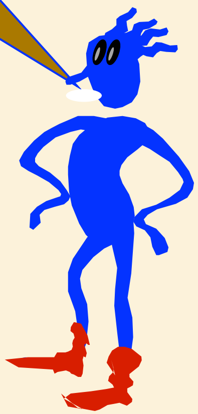




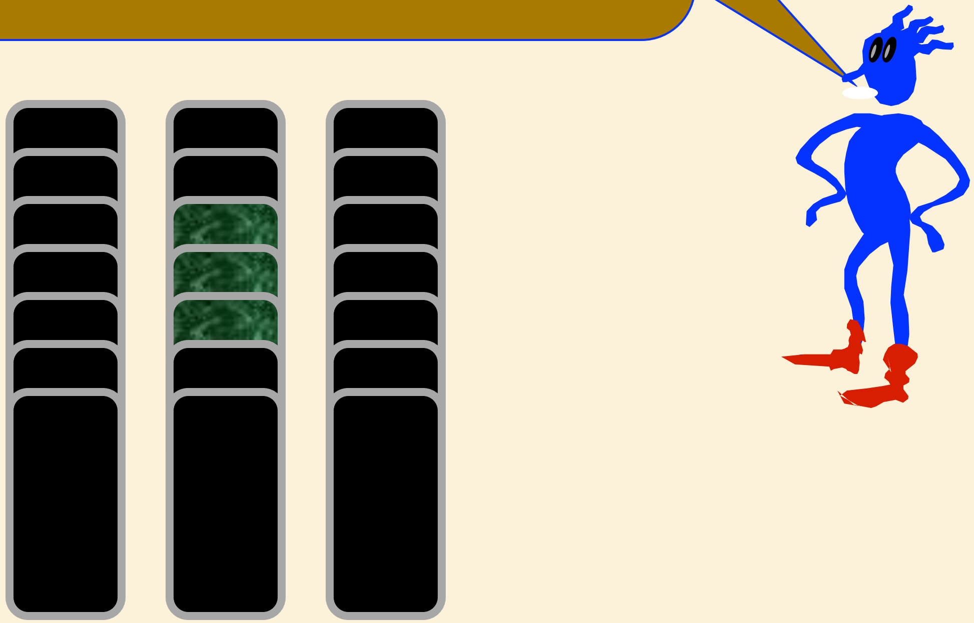

Which column?
right

Loop Invariant:
The selected card is one of these.

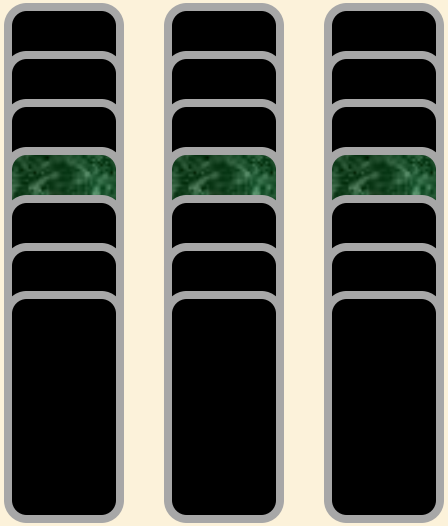


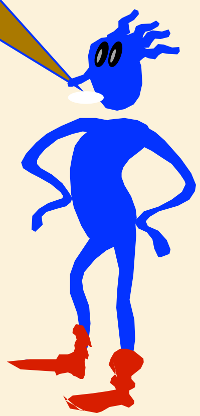


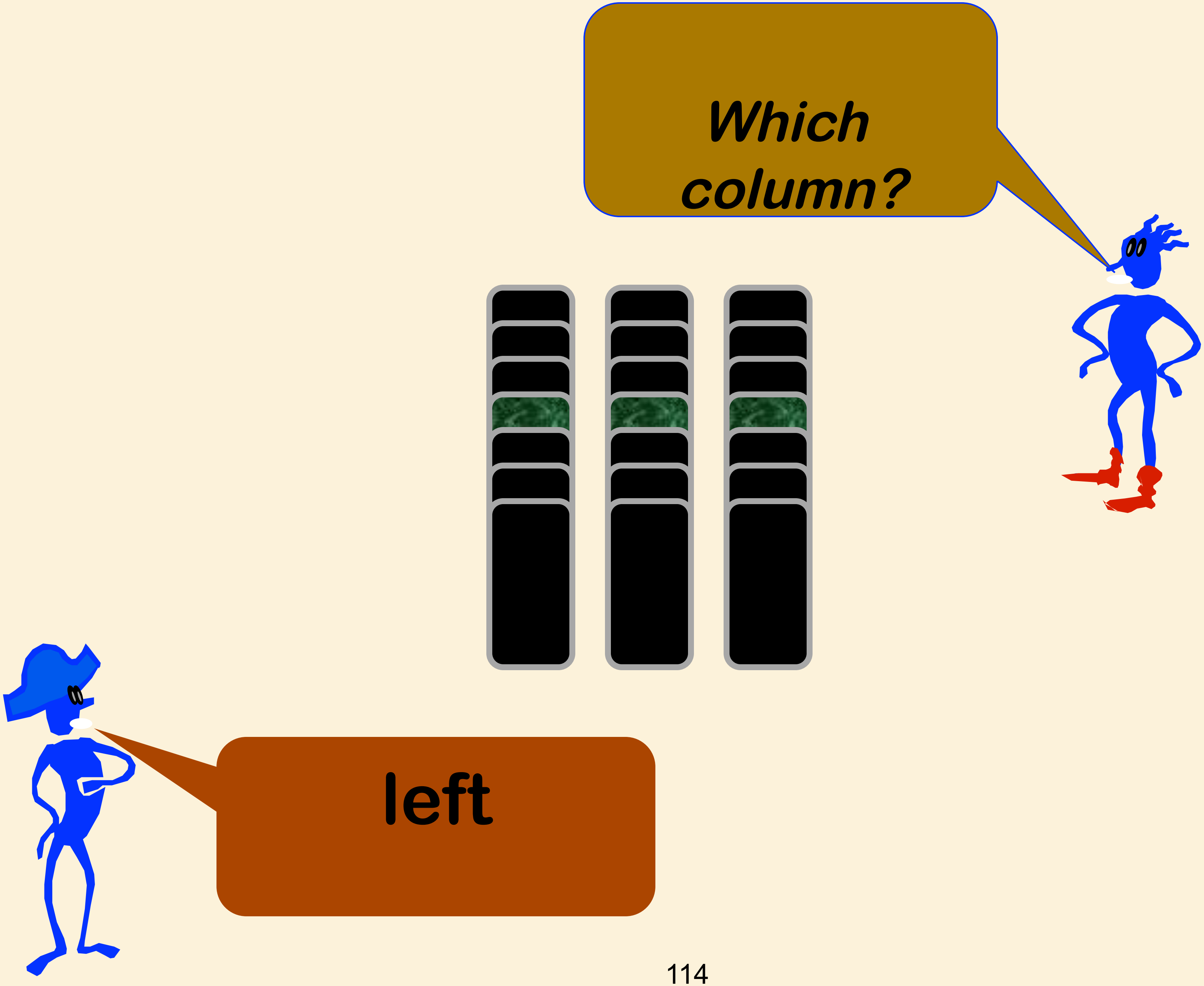
Selected column is placed in the middle



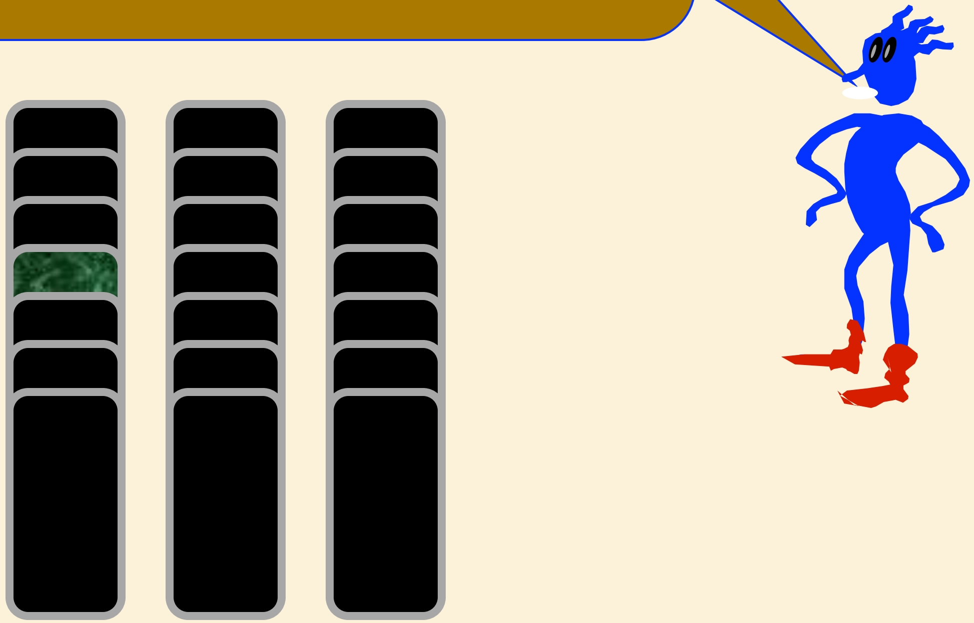
I will rearrange the cards



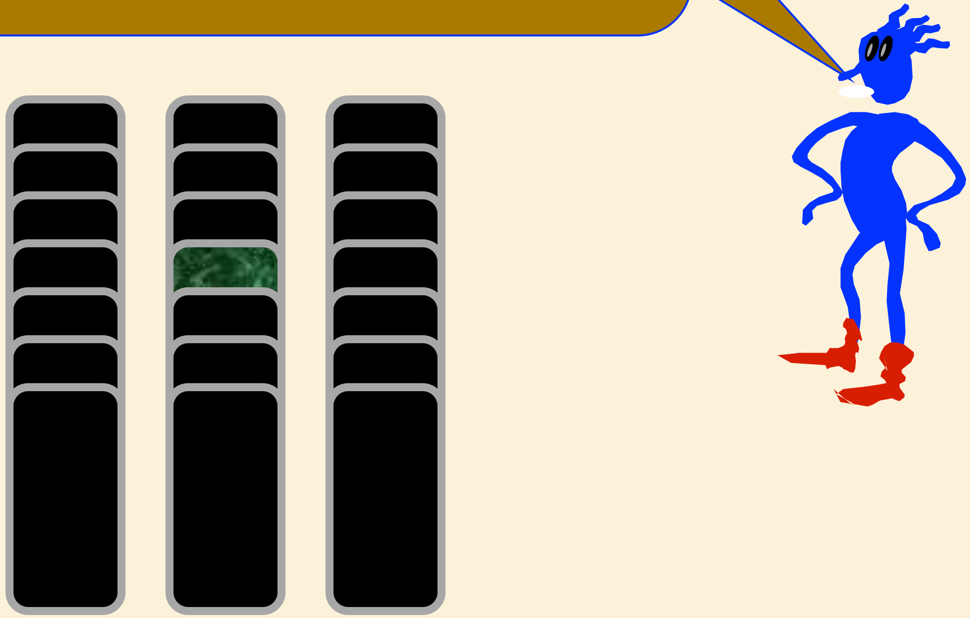


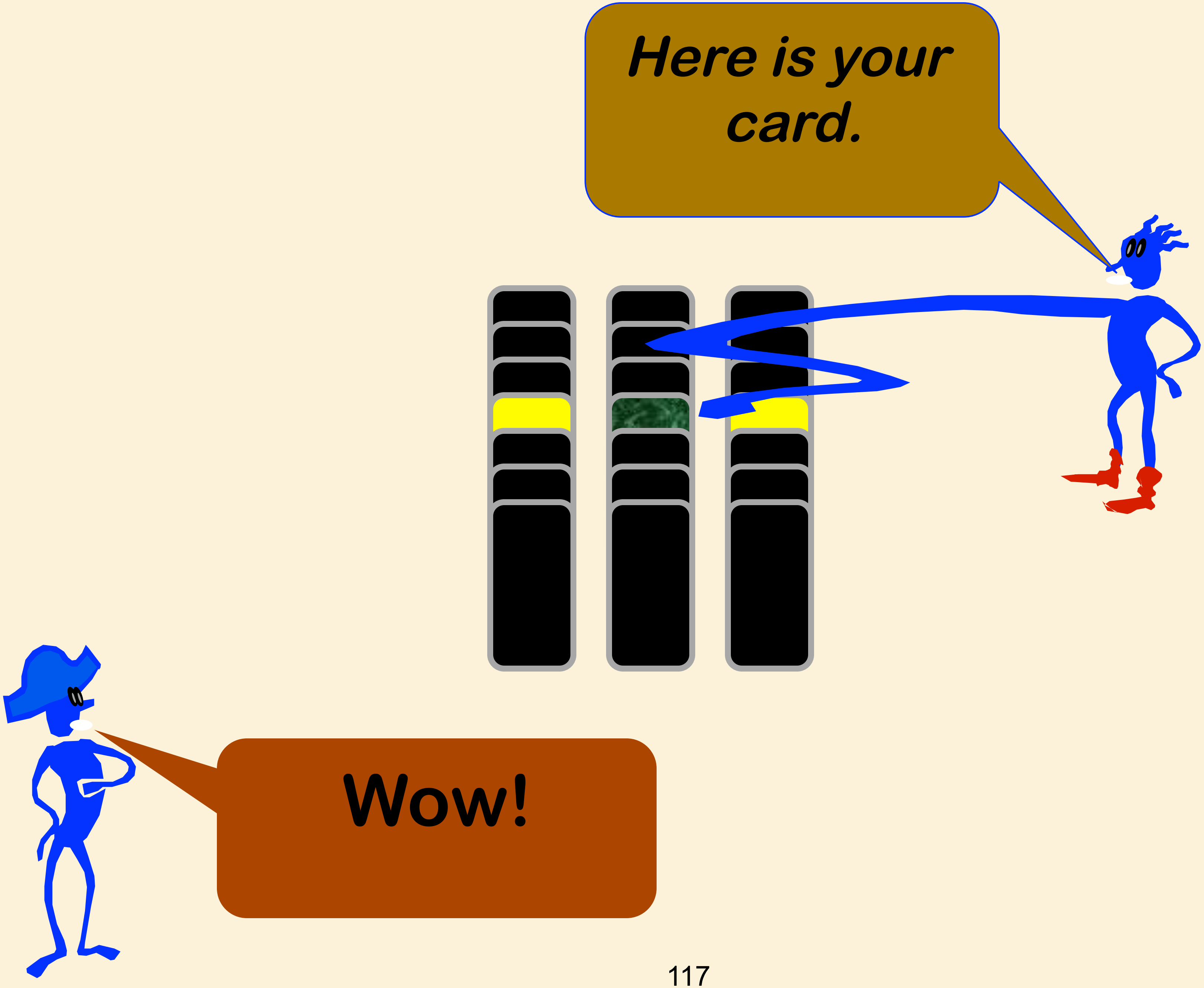

Which column?
left

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?

Loop Invariants
for
Iterative Algorithms

A Fourth Example:

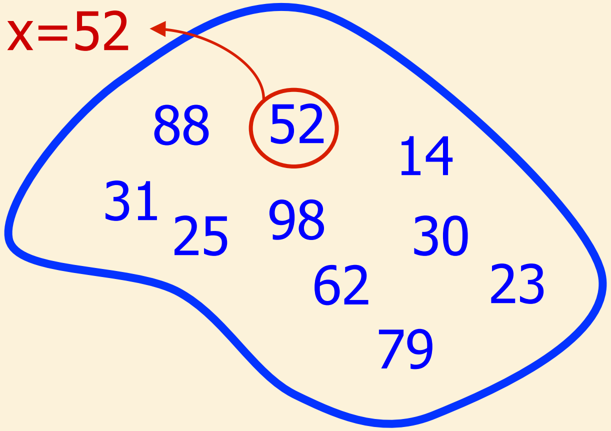
Partitioning

(Not a search problem:

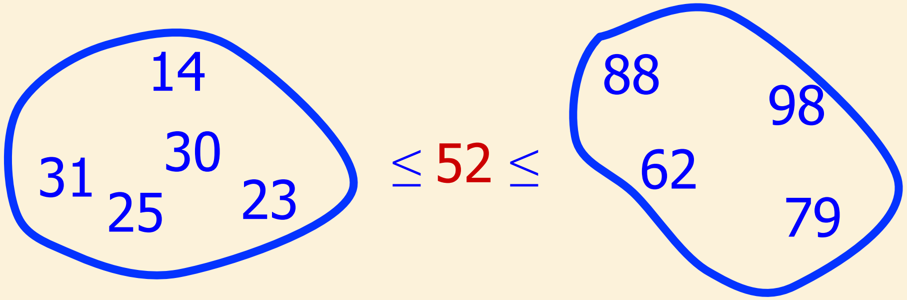
can be used for sorting, e.g., Quicksort)

The “Partitioning” Problem

Input:



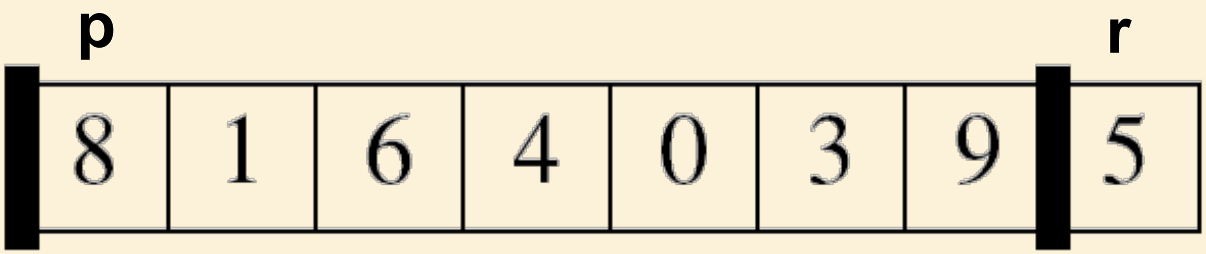
Output:



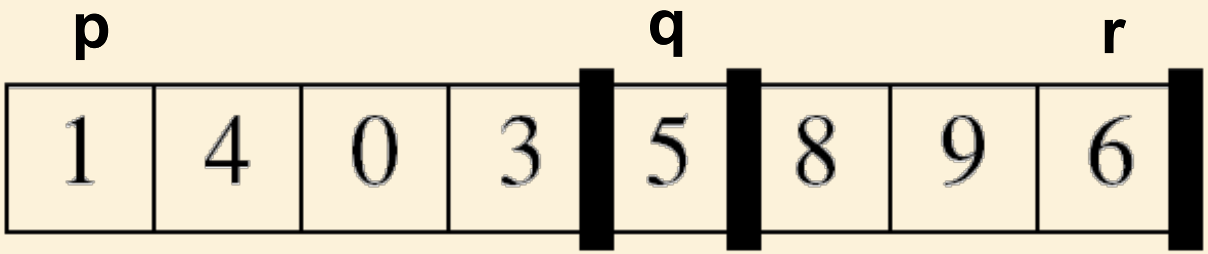
Problem: Partition a list into a set of small values and a set of large values.

Precise Specification

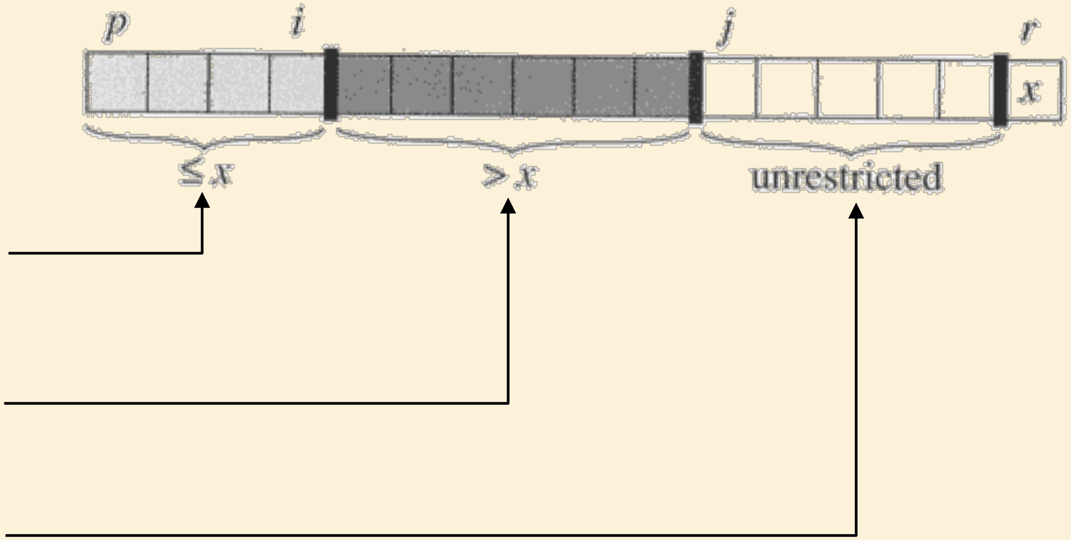
Precondition: $A[p...r]$ is an arbitrary list of values. $x = A[r]$ is the pivot.



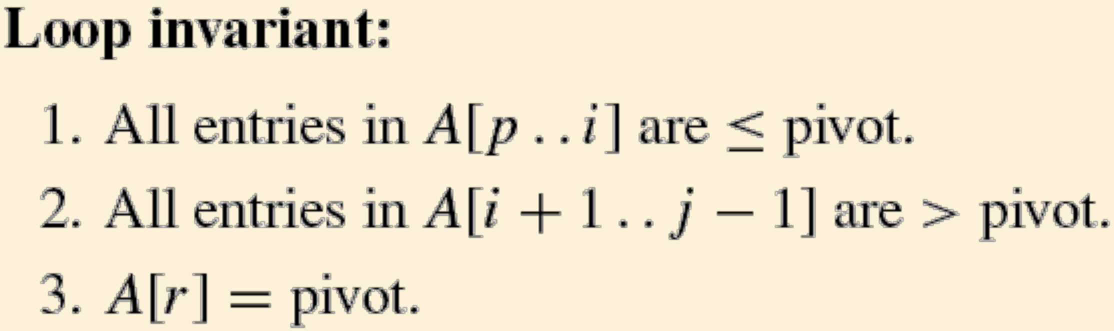
Postcondition: A is rearranged such that $A[p...q-1] \le A[q] = x \le A[q+1...r]$ for some q.



Loop Invariant



- 3 subsets are maintained
  - One containing values less than or equal to the pivot
  - One containing values greater than the pivot
  - One containing values yet to be processed

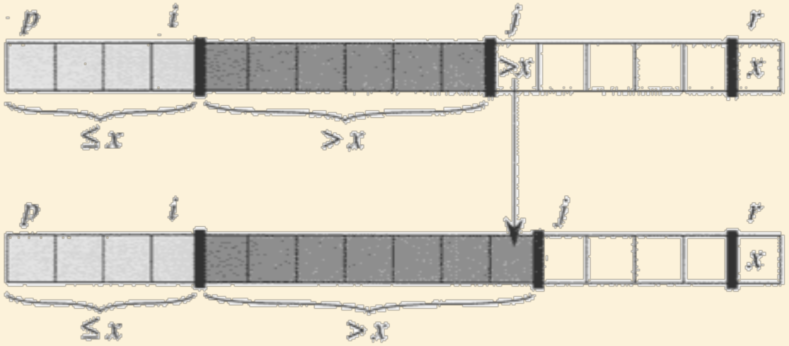


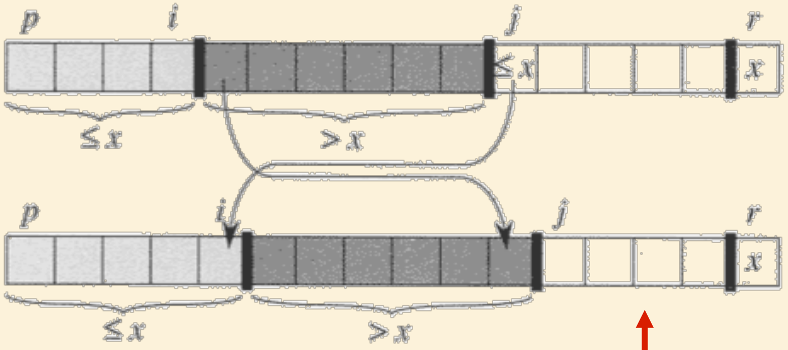
Loop invariant:

1. All entries in $A[p \mathinner{..} i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \mathinner{..} j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Maintaining Loop Invariant

- Consider element at location j
  - If greater than pivot, incorporate into '> set' by incrementing j.

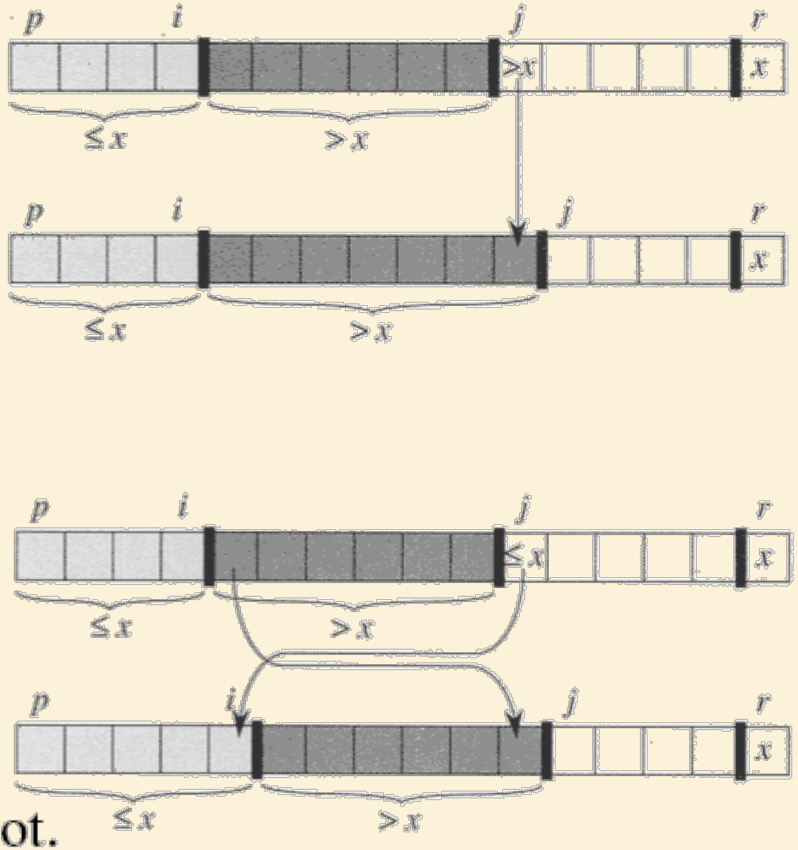




  - If less than or equal to pivot, incorporate into '≤ set' by swapping with element at location i+1 and incrementing both i and j.
  - Measure of progress: size of unprocessed set.

Maintaining Loop Invariant

```
PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $A[j] \leq x$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$
```



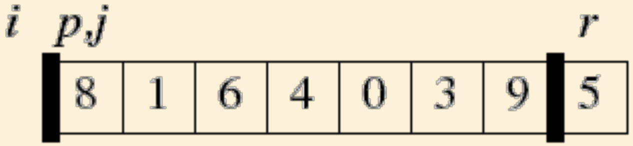
Loop invariant:

1. All entries in $A[p \mathinner{..} i]$ are $\leq$ pivot.
2. All entries in $A[i + 1 \mathinner{..} j - 1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Establishing Loop Invariant

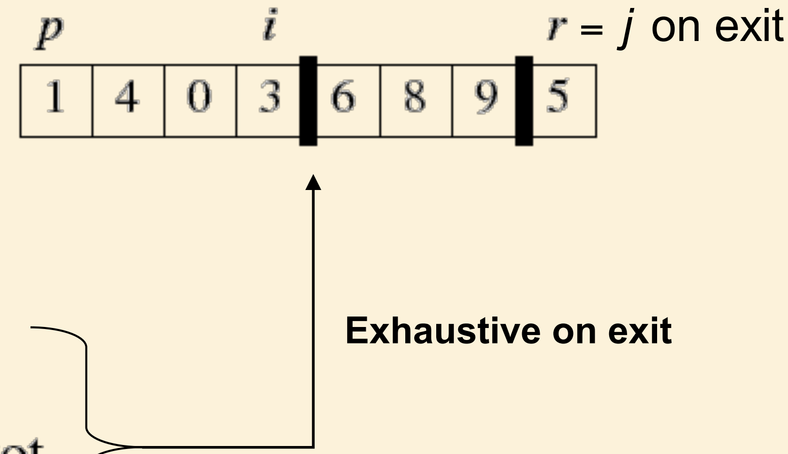
Loop invariant:

1. All entries in $A[p \, .. \, i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \, .. \, j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.



Establishing Postcondition

```
PARTITION($A, p, r$)
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j \leftarrow p$ to $r - 1$
        do if $A[j] \leq x$
            then $i \leftarrow i + 1$
                exchange $A[i] \leftrightarrow A[j]$
    exchange $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$
```

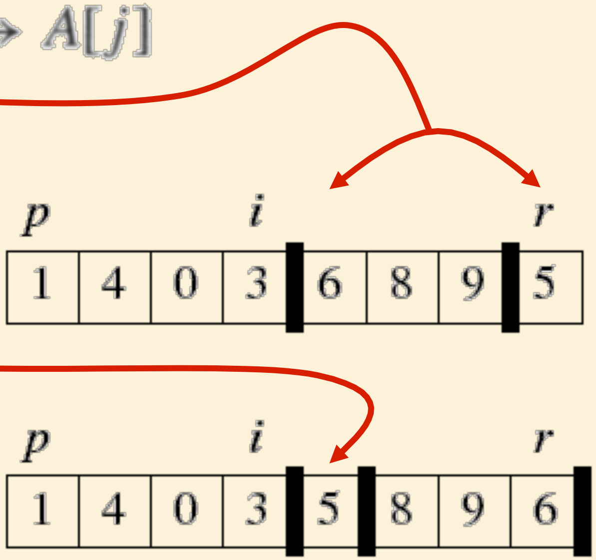


Loop invariant:

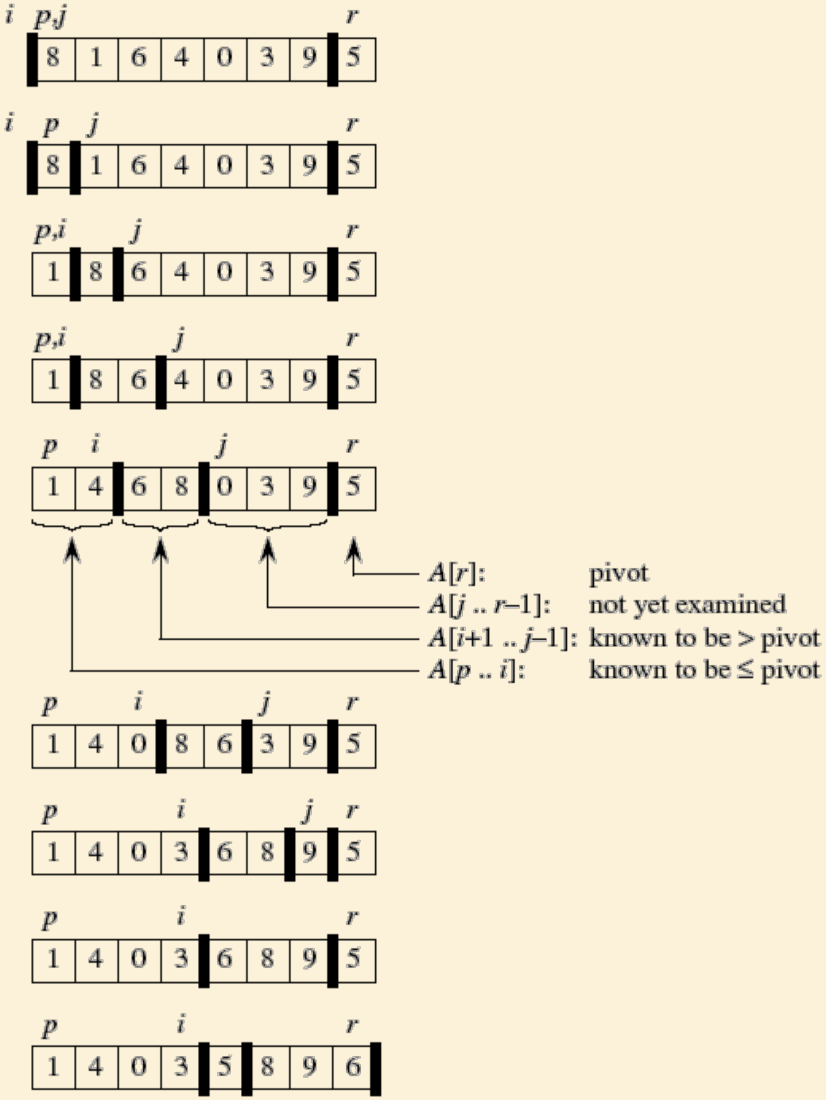
1. All entries in $A[p \mathinner{..} i]$ are $\leq$ pivot.
2. All entries in $A[i + 1 \mathinner{..} j - 1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Establishing Postcondition

```
PARTITION($A, p, r$)
  $x \leftarrow A[r]$
  $i \leftarrow p - 1$
  for $j \leftarrow p$ to $r - 1$
      do if $A[j] \leq x$
          then $i \leftarrow i + 1$
              exchange $A[i] \leftrightarrow A[j]$
  exchange $A[i + 1] \leftrightarrow A[r]$
  return $i + 1$
```

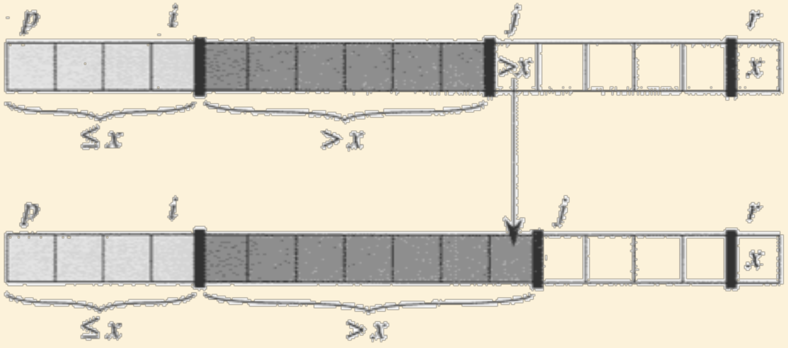


An Example

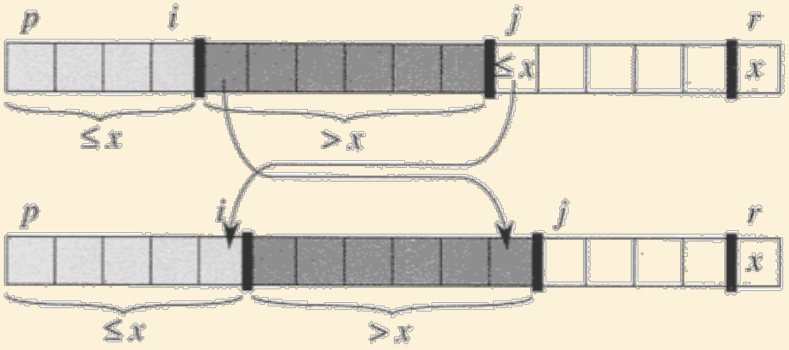

i p,j r
8 1 6 4 0 3 9 5
i p j r
8 1 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p i j r
1 4 6 8 0 3 9 5
A[r]: pivot
A[j .. r–1]: not yet examined
A[i+1 .. j–1]: known to be > pivot
A[p .. i]: known to be ≤ pivot
p i j r
1 4 0 8 6 3 9 5
p i j r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 5 8 9 6

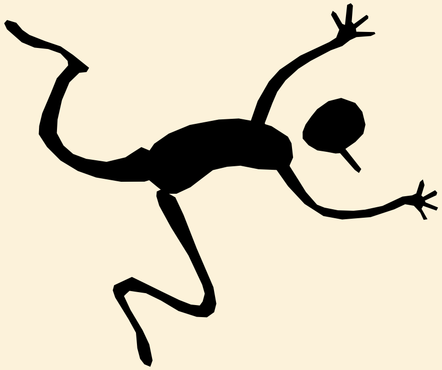
Running Time

Each iteration takes θ(1) time →Total = θ(n)



or

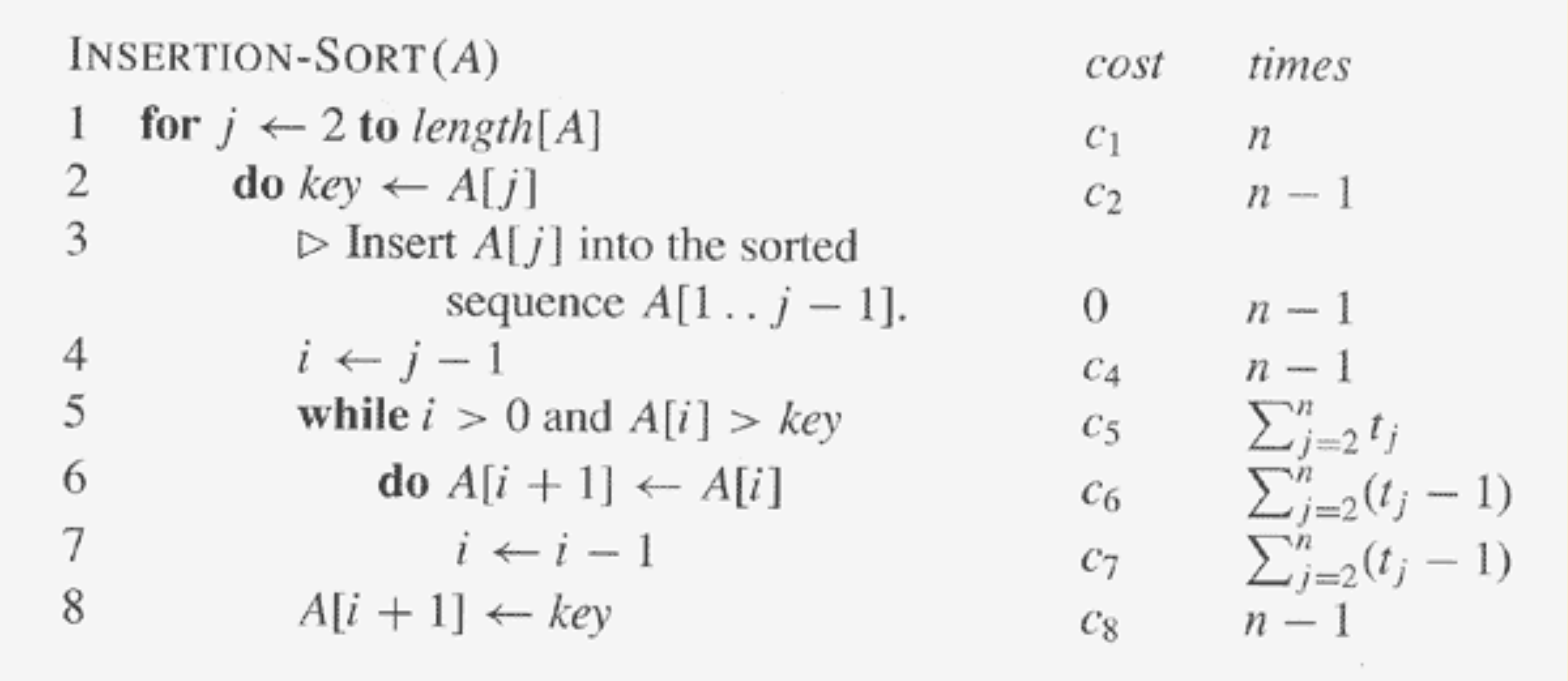




More Examples of Iterative Algorithms

Using Constraints on Input to Achieve Linear-Time Sorting

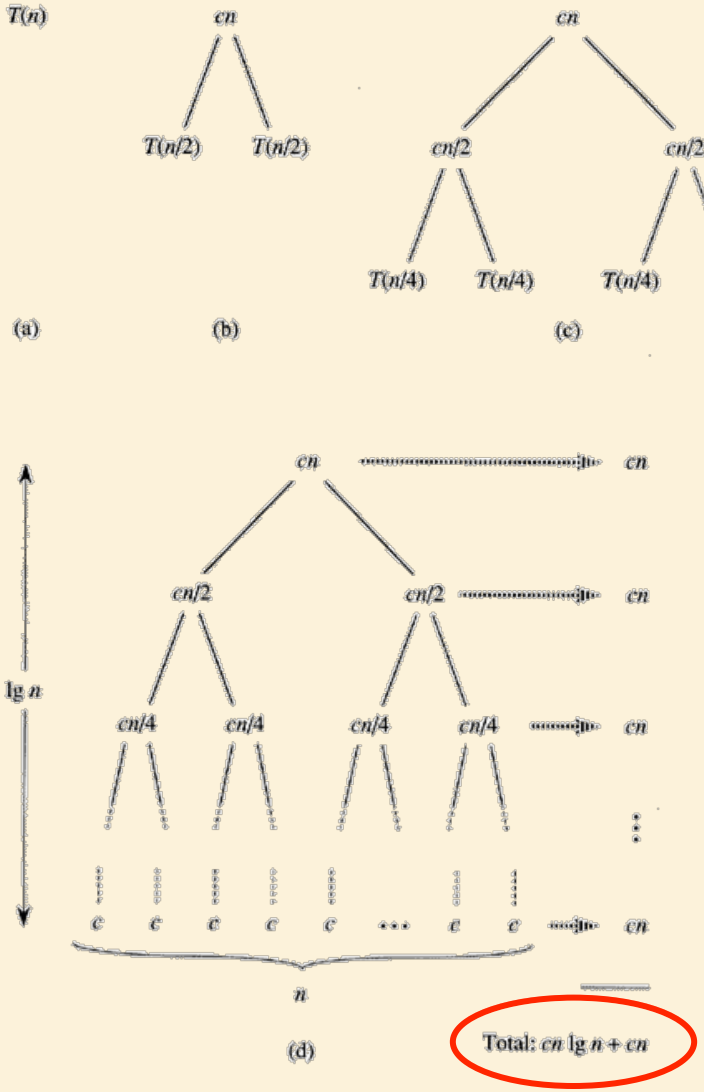
Recall: InsertionSort



```
INSERTION-SORT(A)                                        cost    times
1  for $j \leftarrow 2$ to $length[A]$                   $c_1$   $n$
2       do $key \leftarrow A[j]$                         $c_2$   $n-1$
3          ▷ Insert $A[j]$ into the sorted
                  sequence $A[1 \,..\, j-1]$.            0       $n-1$
4          $i \leftarrow j-1$                            $c_4$   $n-1$
5          while $i > 0$ and $A[i] > key$                $c_5$   $\sum_{j=2}^{n} t_j$
6              do $A[i+1] \leftarrow A[i]$               $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
7                 $i \leftarrow i-1$                     $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
8          $A[i+1] \leftarrow key$                       $c_8$   $n-1$
```

Worst case (reverse order): $t_j = j$: $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \rightarrow T(n) \in \theta(n^2)$

Recall: MergeSort



$T(n) \in \theta(n \log n)$

Comparison Sorts

- InsertionSort and MergeSort are examples of (stable) Comparison Sort algorithms.
- QuickSort is another example we will study shortly.
- Comparison Sort algorithms sort the input by successive comparison of pairs of input elements.
- Comparison Sort algorithms are very general: they make no assumptions about the values of the input elements.

Comparison Sorts

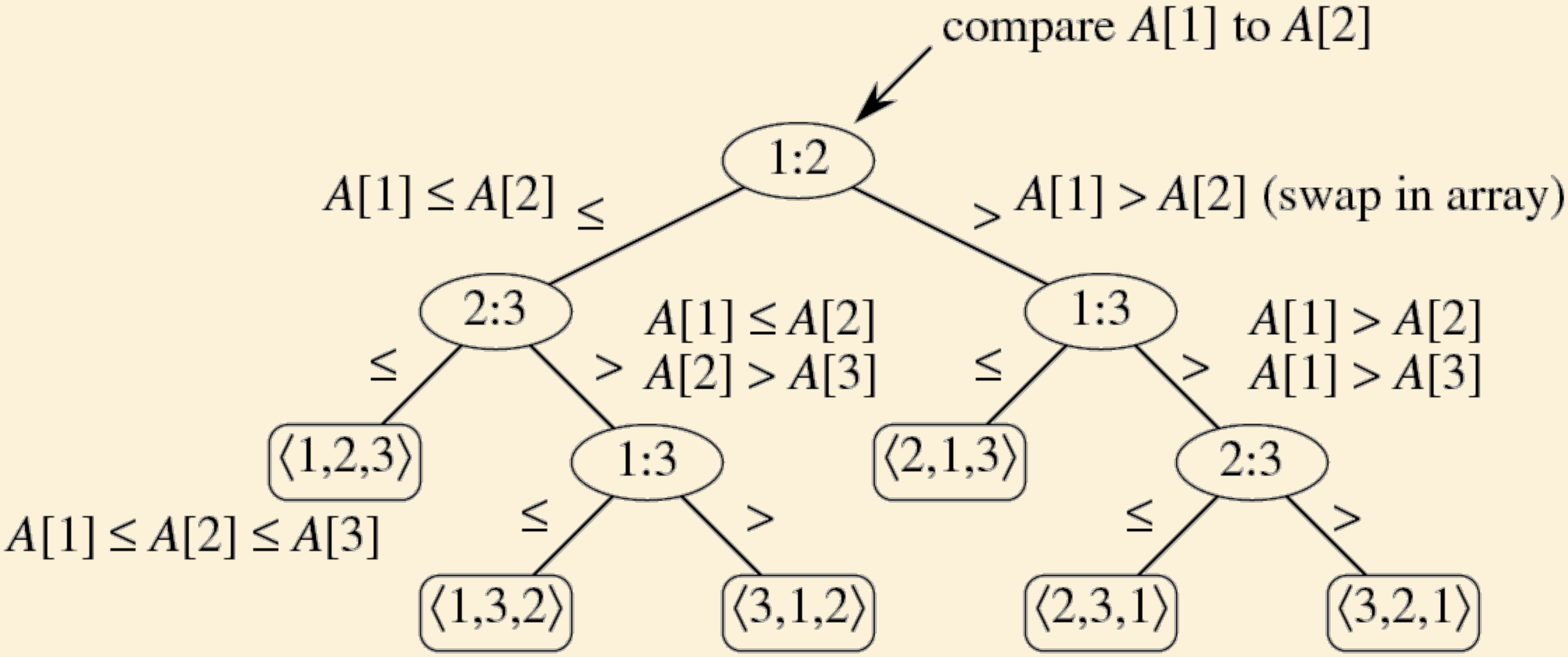
InsertionSort is $\theta(n^2)$.

MergeSort is $\theta(n \log n)$.

Can we do better?

Comparison Sort: Decision Trees

- Example: Sorting a 3-element array A[1..3]



Comparison Sort

- Worst-case time is equal to the height of the binary decision tree.
- The height of the tree is the log of the number of leaves.
- The leaves of the tree represent all possible permutations of the input. How many are there?

$$\log(n!) \in \Omega(n \log n)$$

Thus MergeSort is asymptotically optimal.

Linear Sorts?

Comparison sorts are very general, but are $\Omega(n \log n)$

Faster sorting may be possible if we can constrain the nature of the input.

Example 1. Counting Sort

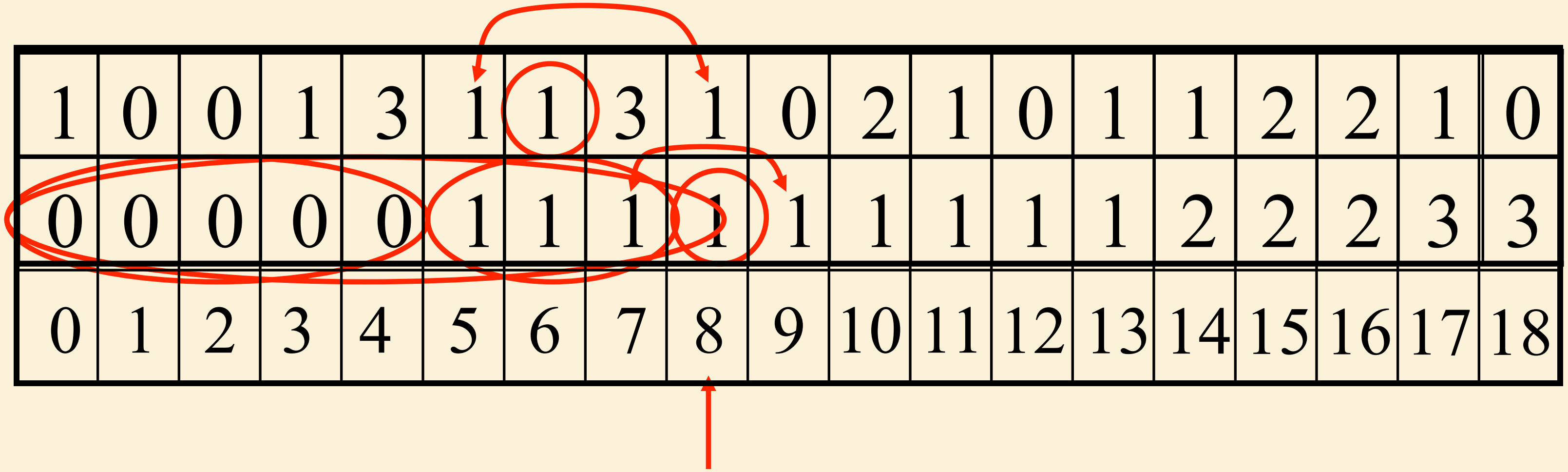
- Counting Sort applies when the elements to be sorted come from a finite (and preferably small) set.
- For example, the elements to be sorted are integers in the range [0…k-1], for some fixed integer k.
- We can then create an array V[0…k-1] and use it to count the number of elements with each value [0…k-1].
- Then each input element can be placed in exactly the right place in the output array in constant time.

Counting Sort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	3	3	3

- Input: N records with integer keys between [0…k-1].
- Output: Stable sorted keys.
- Algorithm:
  - Count frequency of each key value to determine transition locations
  - Go through the records in order putting them where they go.

CountingSort



Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Stable sort: If two keys are the same, their order does not change.

Thus the 4th record in input with digit 1 must be
the 4th record in output with digit 1.

It belongs at output index 8, because 8 records go before it

ie, 5 records with a smaller digit & 3 records with the same digit

Count These!

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:																			
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit v:	5	9	3	2

N records. Time to count? $\Theta(N)$

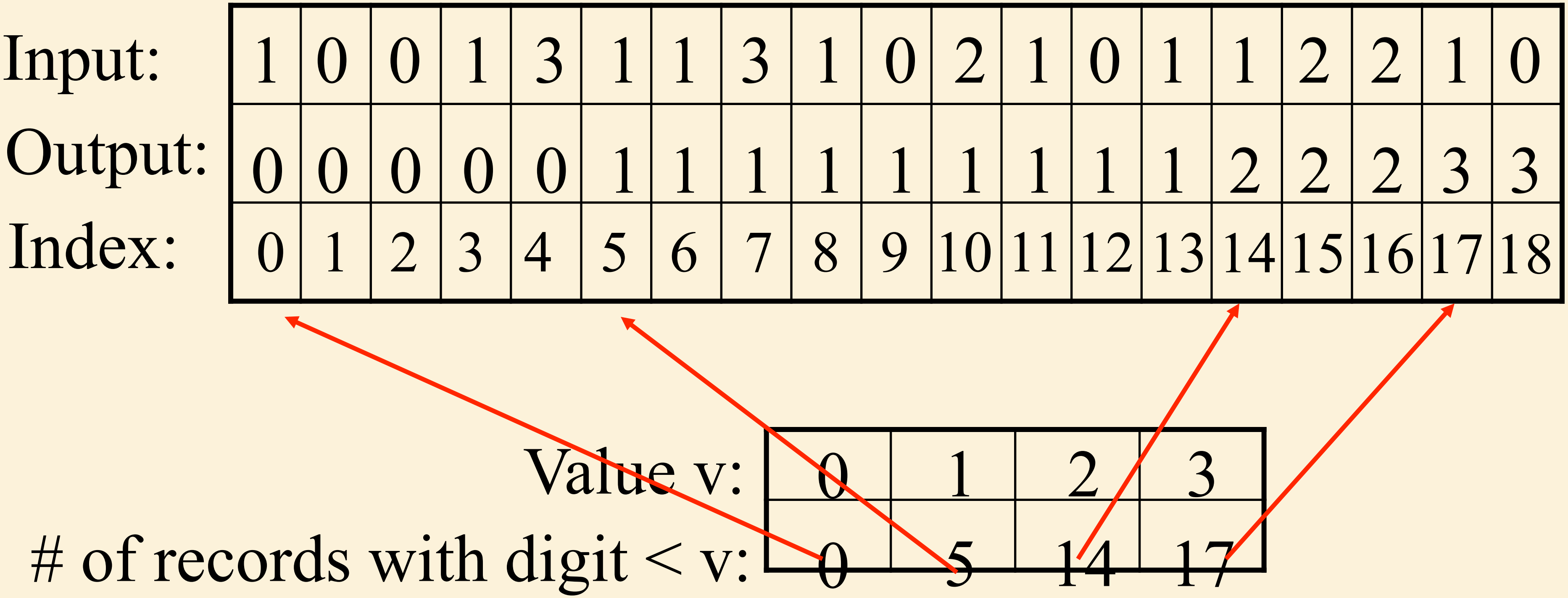
CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:																			
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit v:	5	9	3	3
# of records with digit < v:	0	5	14	17

N records, k different values. Time to count? $\Theta(k)$

CountingSort

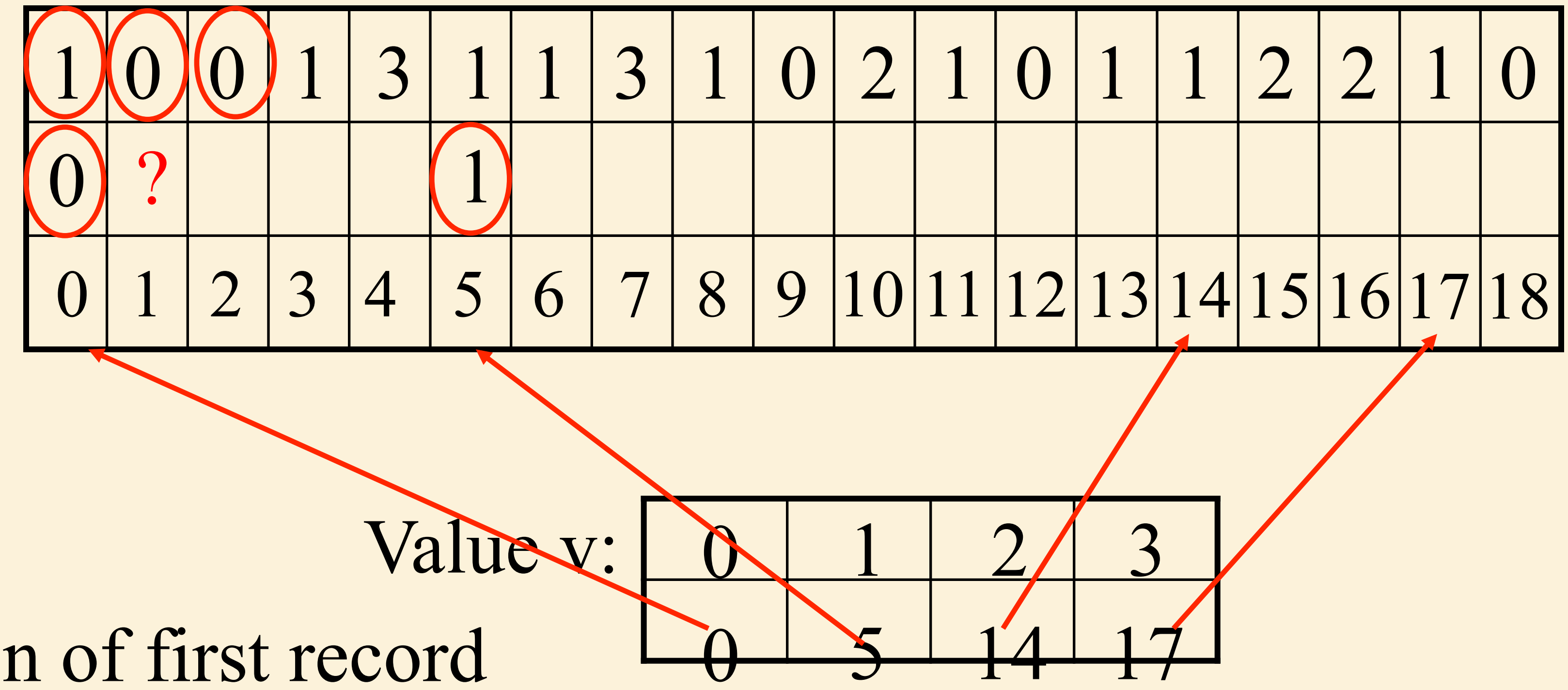


Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit < v:	0	5	14	17

= location of first record with digit v.

CountingSort



Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	?				1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of first record with digit v.	0	5	14	17

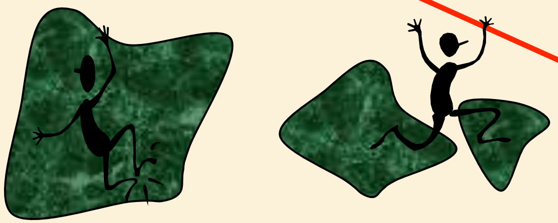
Algorithm: Go through the records in order putting them where they go.

Loop Invariant

- The first $i-1$ keys have been placed in the correct locations in the output array
- The auxiliary data structure v indicates the location at which to place the i^{th} key for each possible key value from $[1..k-1]$.

CountingSort

Input:	~~1~~	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:						1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



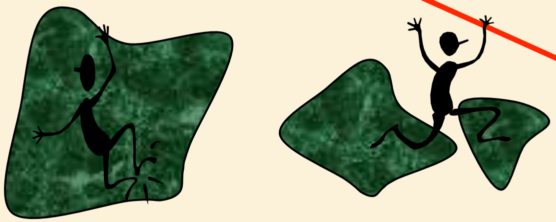
Value v:	0	1	2	3
	0	5	14	17

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0					1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

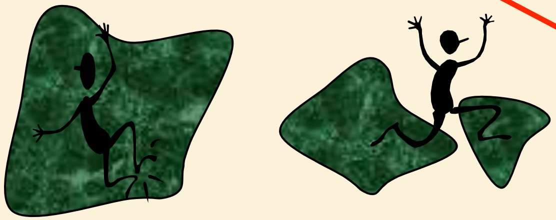


Value v:	0	1	2	3
Location of next record with digit v.	0	6	14	17

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



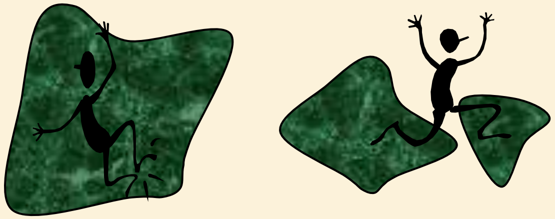
Value v:	0	1	2	3
	1	6	14	17

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1												
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



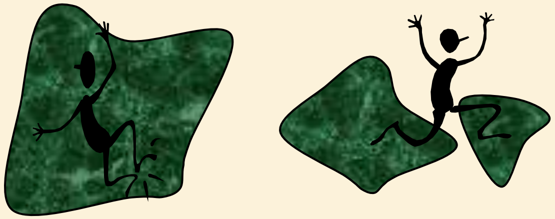
Value v:	0	1	2	3
	2	6	14	17

Location of next record with digit v.

Algorithm: Go through the records in order
putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1											3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



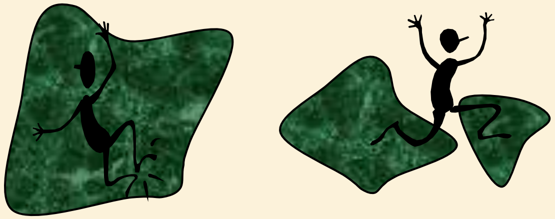
Value v:	0	1	2	3
	2	7	14	17

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1										3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

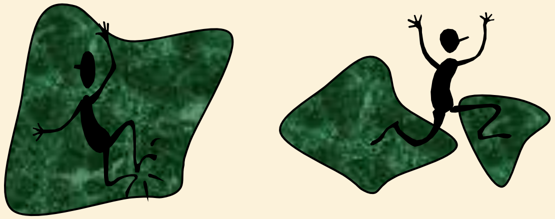


Value v:	0	1	2	3
Location of next record with digit v.	2	7	14	18

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1									3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



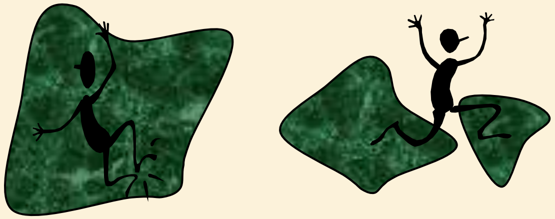
Value v:	0	1	2	3
	2	8	14	18

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1									3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



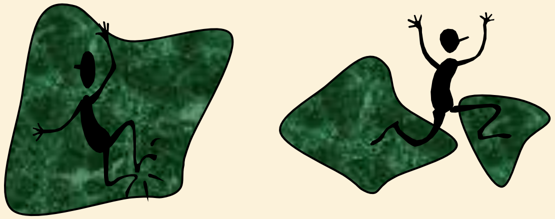
Value v:	0	1	2	3
	2	9	14	18

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1	1								3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



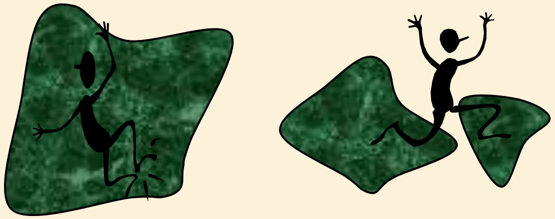
Value v:	0	1	2	3
	2	9	14	19

Location of next record
with digit v.

Algorithm: Go through the records in order
putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	2	1	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1								3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



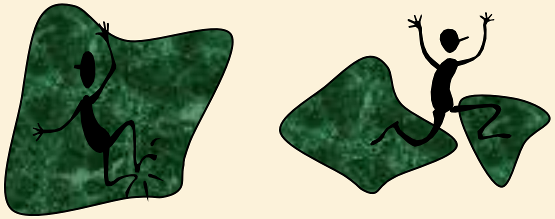
Value v:	0	1	2	3
	2	10	14	19

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	1	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1					2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



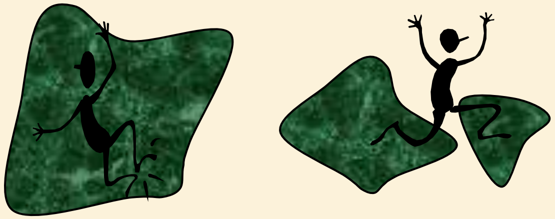
Value v:	0	1	2	3
	3	10	14	19

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1	1				2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



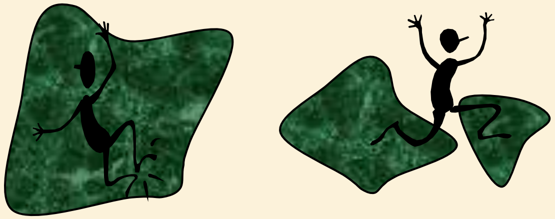
Value v:	0	1	2	3
	3	10	15	19

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1	1				2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



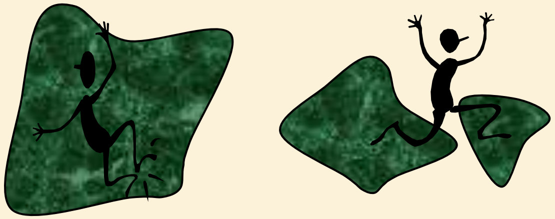
Value v:	0	1	2	3
	3	10	15	19

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~1~~	~~0~~
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



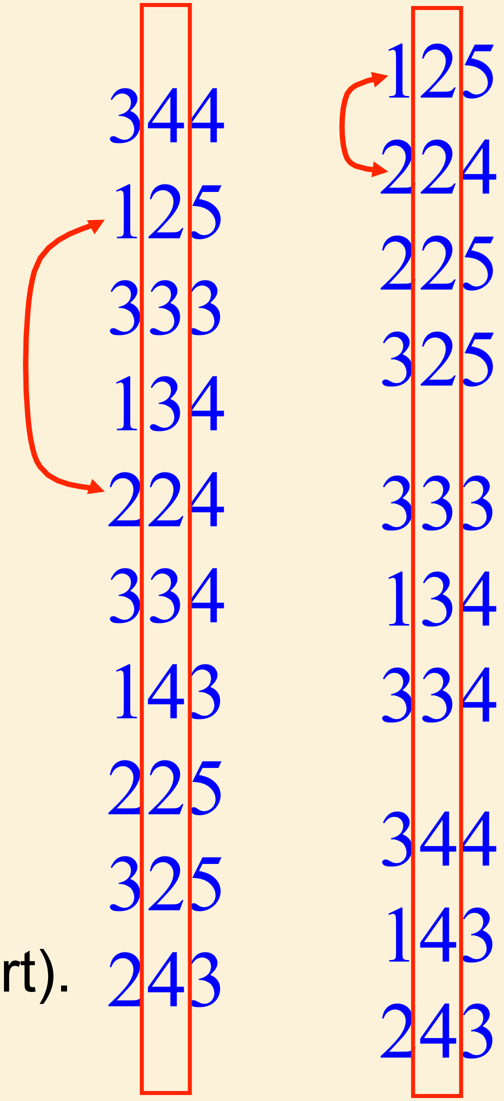
Value v:	0	1	2	3
	5	14	17	19

Location of next record with digit v.

Time = Θ(N)

Total = Θ(N+k)

Example 2. RadixSort



344
125
333
134
224
334
143
225
325
243

125
224
225
325
333
134
334
344
143
243

Input:

- A of stack of *N* punch cards.
- Each card contains *d* digits.
- Each digit between *[0…k-1]*

Output:

- Sorted cards.

Digit Sort:

- Select one digit
- Separate cards into k piles based on selected digit (e.g., Counting Sort).

Stable sort: If two cards are the same for that digit, their order does not change.

RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

The most significant.

125
134
143
224
225
243
344
333
334
325

Sort wrt which digit Second?

The next most significant.

125
224
225
325
134
333
334
143
243
344

All meaning in first sort lost.

RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

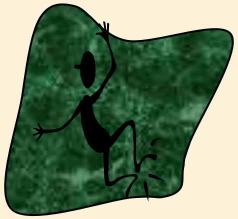
The least significant.

333
143
243
344
134
224
334
125
225
325

Sort wrt which digit Second?

The next least significant.

224
125
225
325
333
134
334
143
243
344



RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

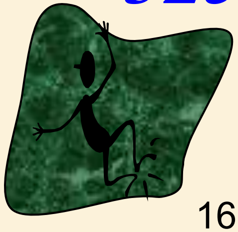
The least significant.

333
143
243
344
134
224
334
125
225
325

Sort wrt which digit Second?

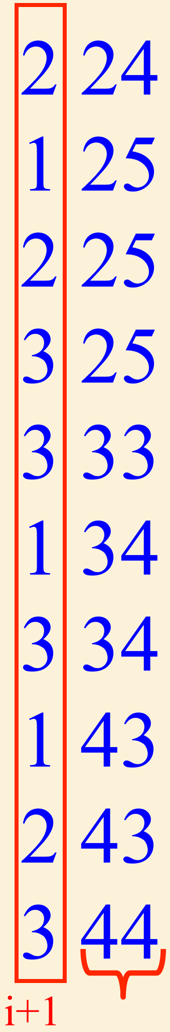
The next least significant.

2 24
1 25
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44

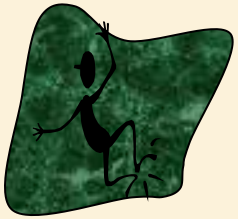


Is sorted wrt least sig. 2 digits.

RadixSort

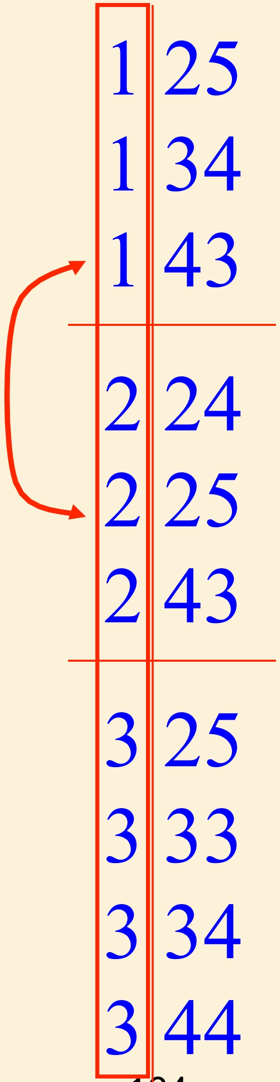


2 24
1 25
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44



Is sorted wrt
first i digits.

Sort wrt i+1st
digit.



1 25
1 34
1 43

2 24
2 25
2 43

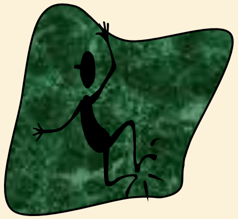
3 25
3 33
3 34
3 44

Is sorted wrt
first i+1 digits.

These are in the
correct order
because sorted
wrt high order digit

RadixSort

2 24
1 25
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44
i+1



Is sorted wrt first i digits.

Sort wrt i+1st digit.

1 25
1 34
1 43

2 24
2 25
2 43

3 25
3 33
3 34
3 44

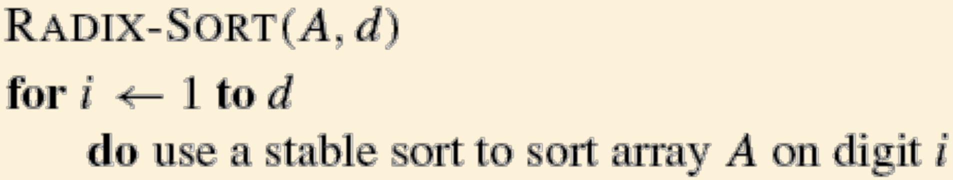
Is sorted wrt first i+1 digits.

These are in the correct order because was sorted & stable sort left sorted

Loop Invariant

- The keys have been correctly stable-sorted with respect to the *i-1* least-significant digits.

Running Time



```
RADIX-SORT($A, d$)
for $i \leftarrow 1$ to $d$
    do use a stable sort to sort array $A$ on digit $i$
```

Running time is $\Theta(d(n+k))$

Where

d = # of digits in each number

n = # of elements to be sorted

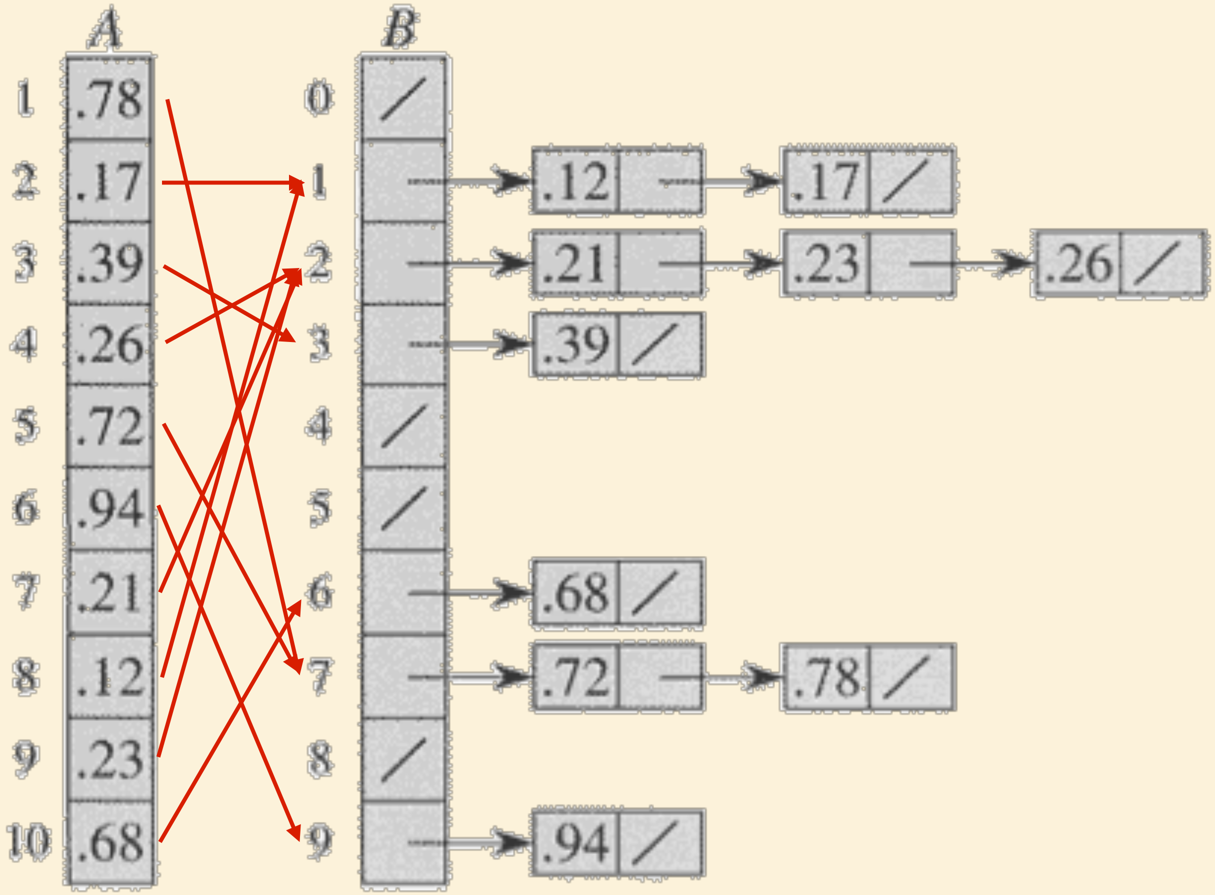
k = # of possible values for each digit

Example 3. Bucket Sort

- Applicable if input is constrained to finite interval, e.g., [0…1).
- If input is random and uniformly distributed, **expected** run time is Θ(n).

Bucket Sort

insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$



Loop Invariants

- Loop 1
  - The first *i-1* keys have been correctly placed into buckets of width *1/n*.
- Loop 2
  - The keys within each of the first *i-1* buckets have been correctly stable-sorted.

PseudoCode

```
BUCKET-SORT$(A, n)$
for $i \leftarrow 1$ to $n$
    do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$    $\longleftarrow \Theta(1)$
for $i \leftarrow 0$ to $n - 1$
    do sort list $B[i]$ with insertion sort    $\longleftarrow \Theta(1) \times n$
concatenate lists $B[0], B[1], \ldots, B[n-1]$    $\longleftarrow \Theta(n)$
return the concatenated lists
```

$$\overline{}$$

$$\Theta(n)$$

Examples of Iterative Algorithms

- Binary Search
- Partitioning
- Insertion Sort
- Counting Sort
- Radix Sort
- Bucket Sort

- Which can be made stable?
- Which sort in place?
- How about MergeSort?

End of Iterative Algorithms