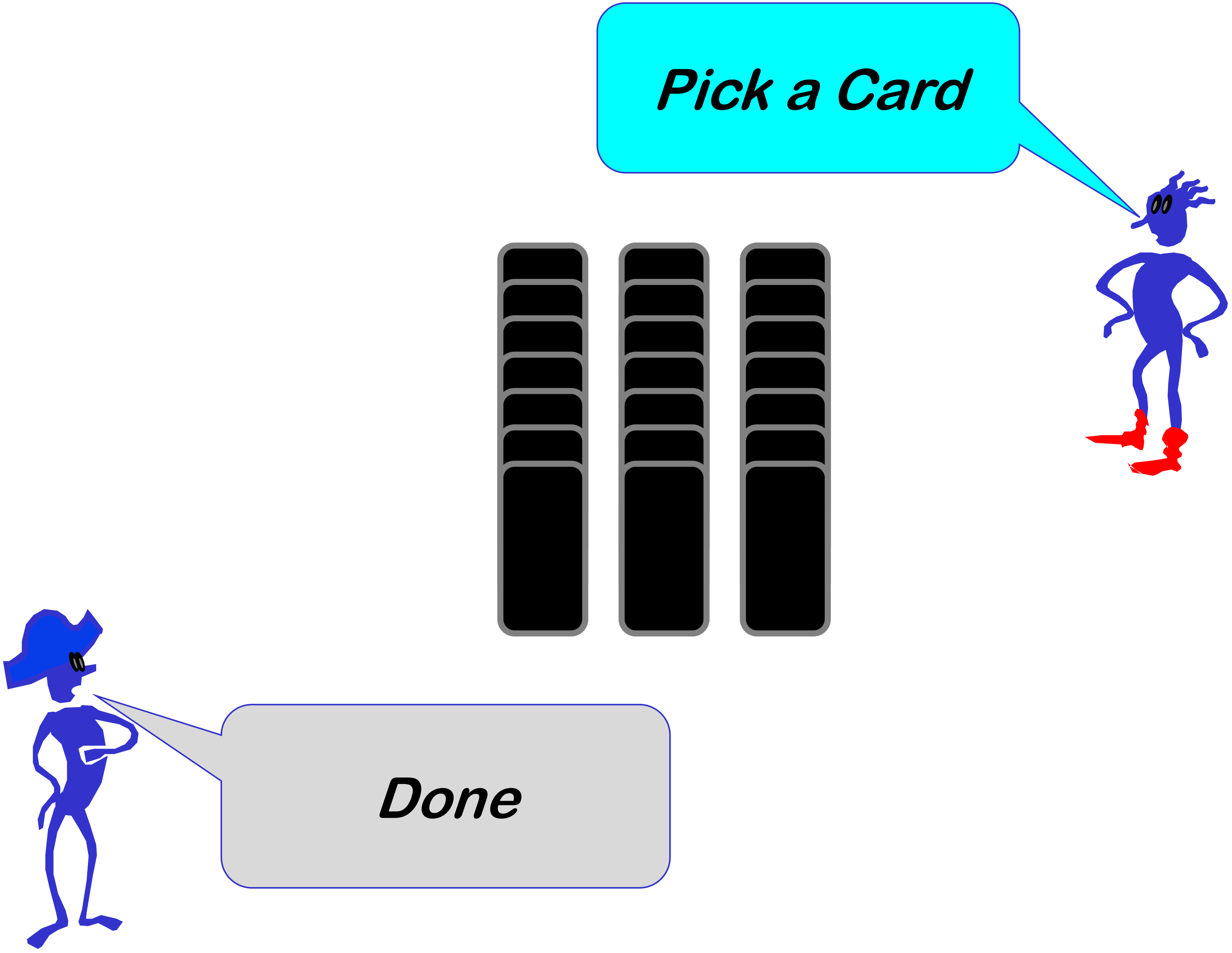
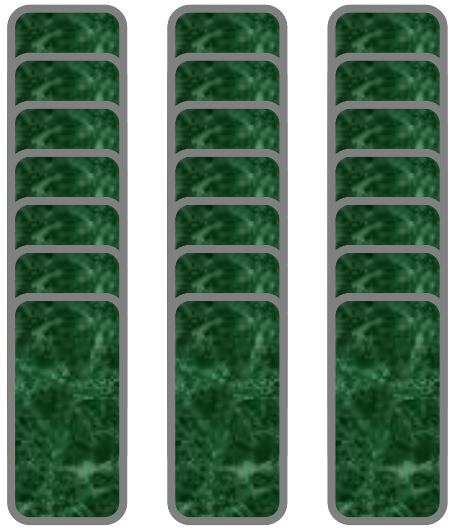
Iterative Algorithms - Part 2

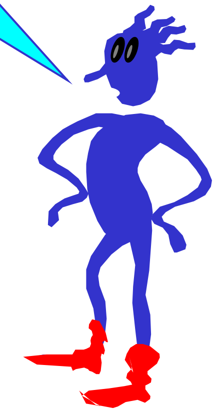
Loop Invariants
for
Iterative Algorithms

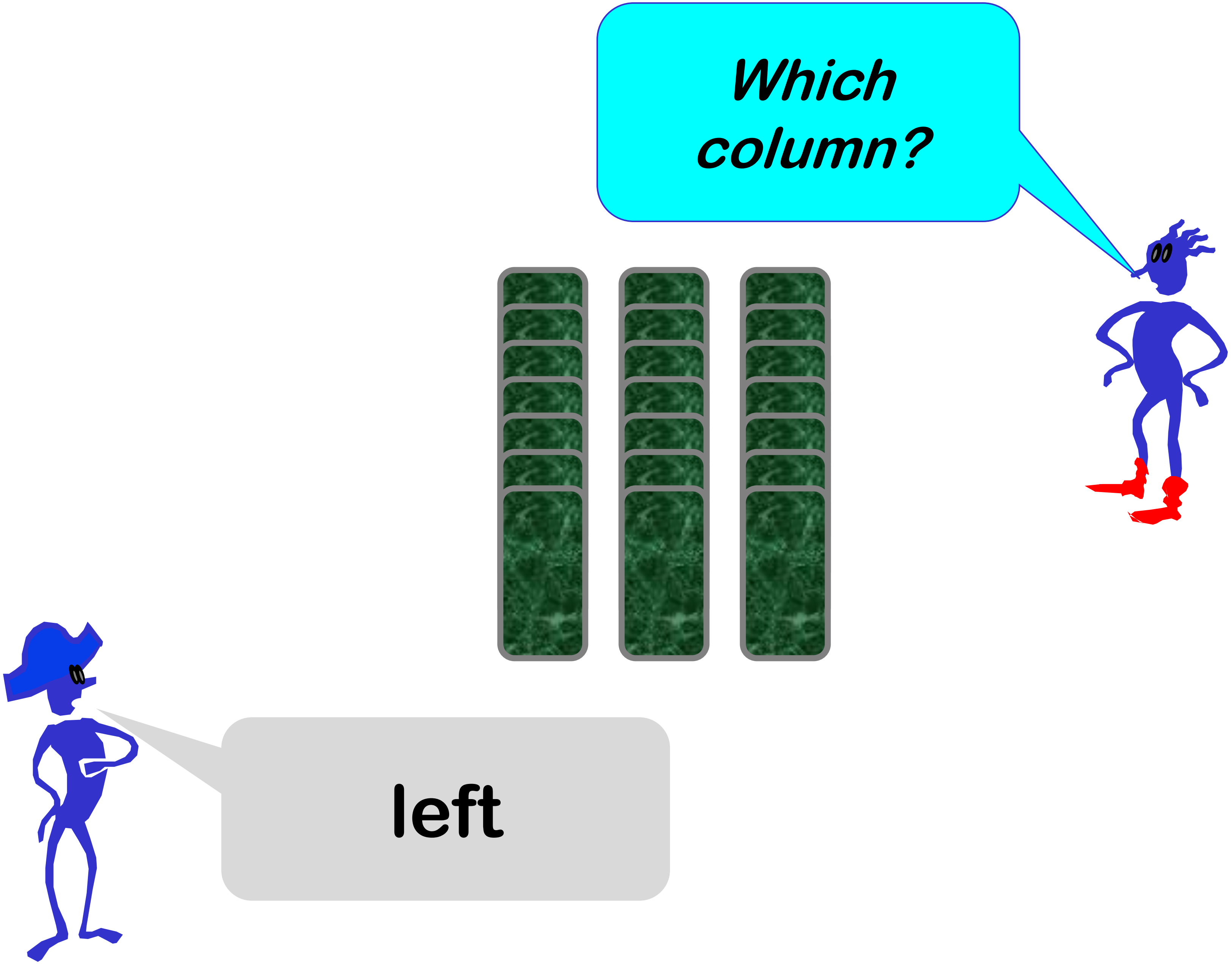
A Third Search Example: A Card Trick


Pick a Card
Done

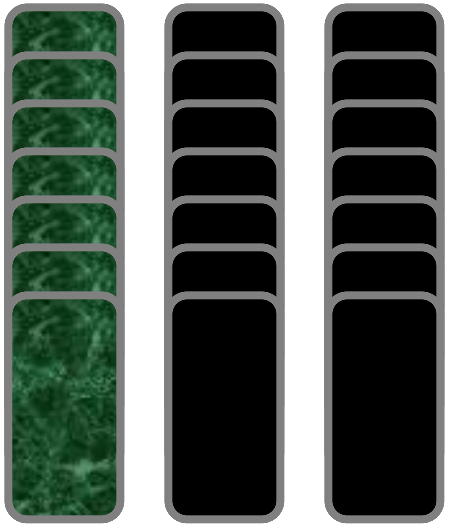
Loop Invariant:
The selected card is one of these.

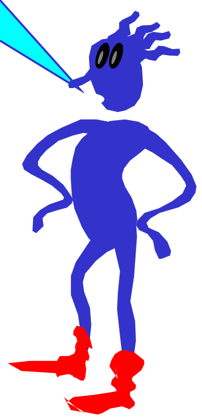


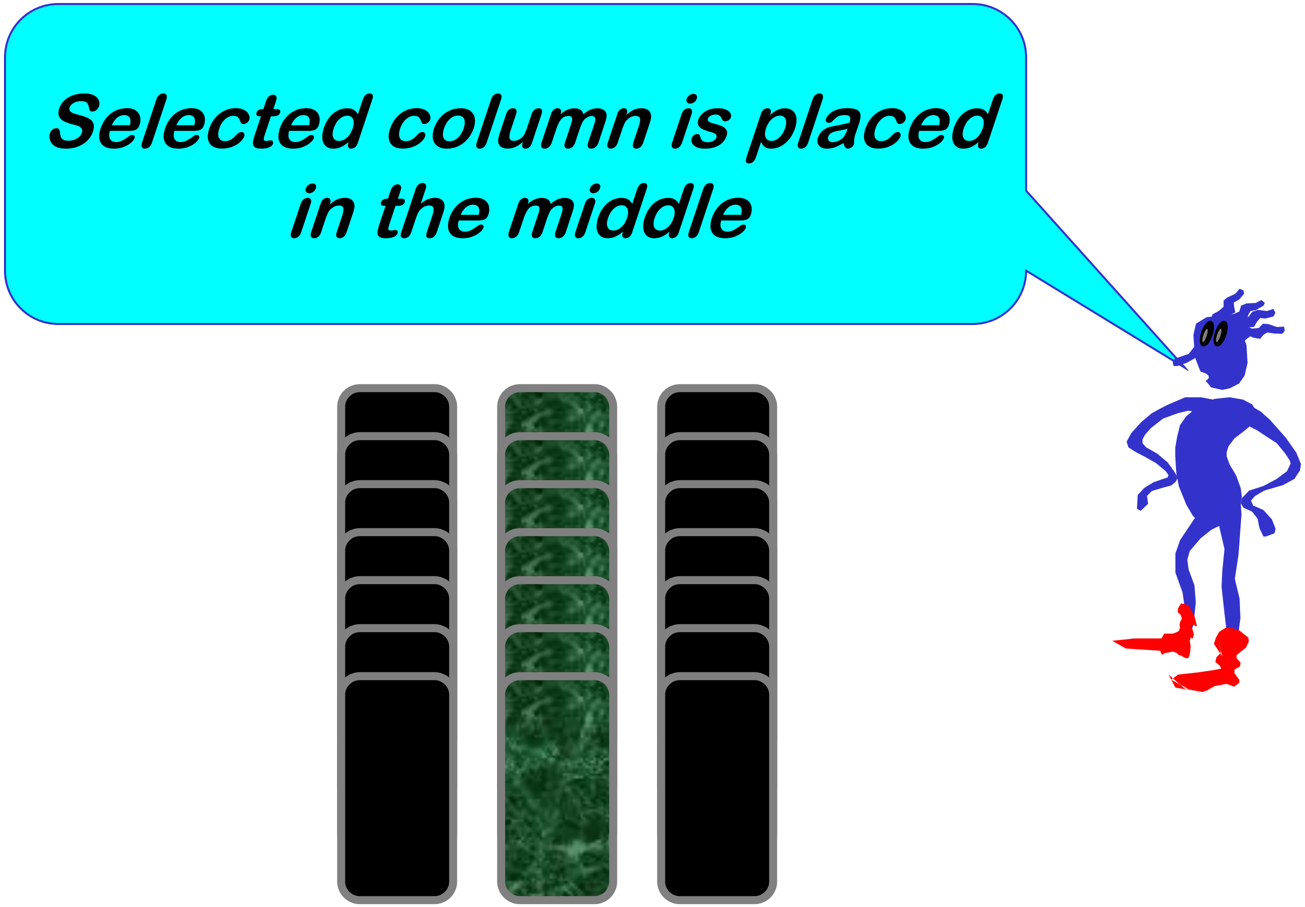


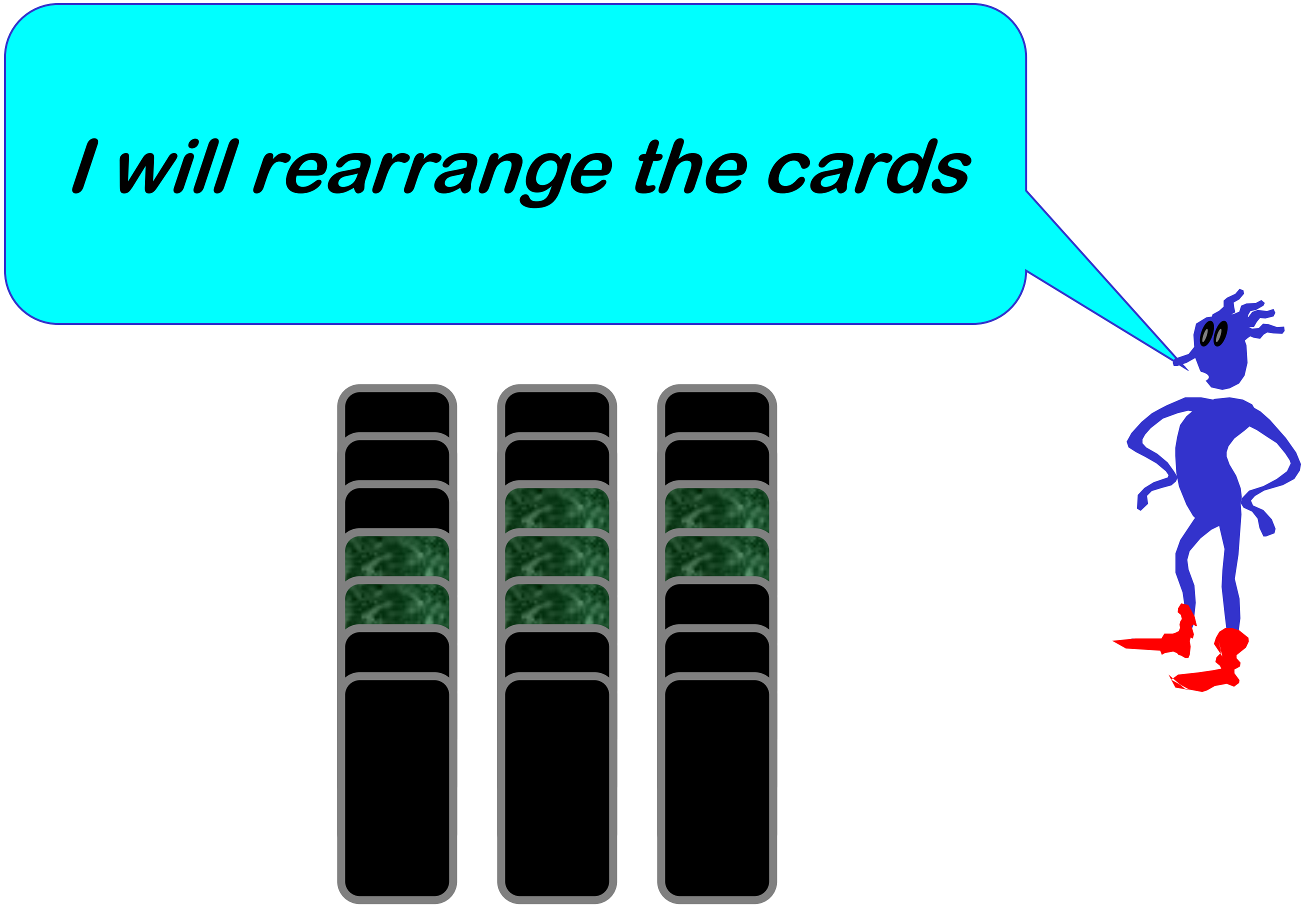

Which column?
left

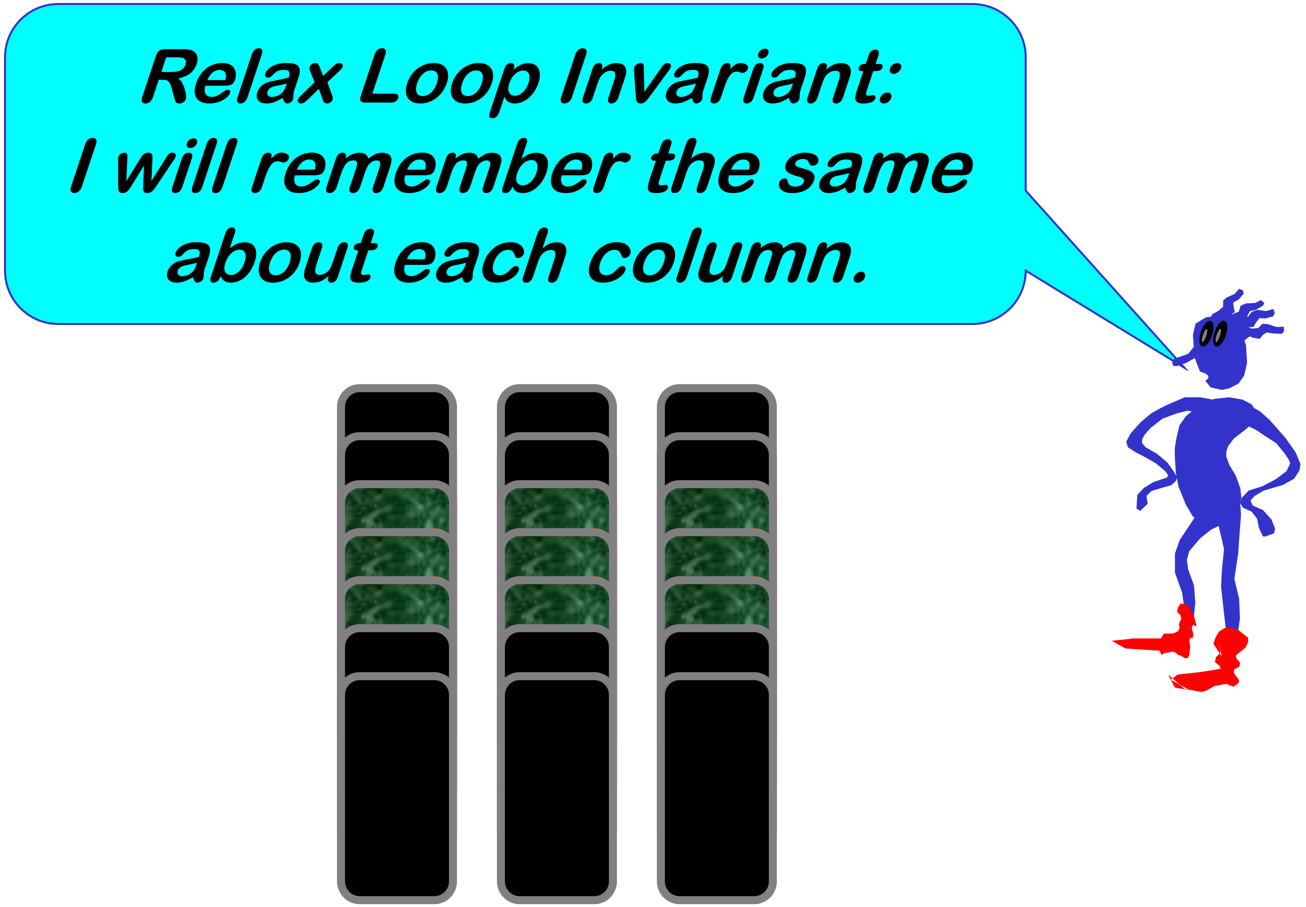
Loop Invariant:
The selected card is one of these.

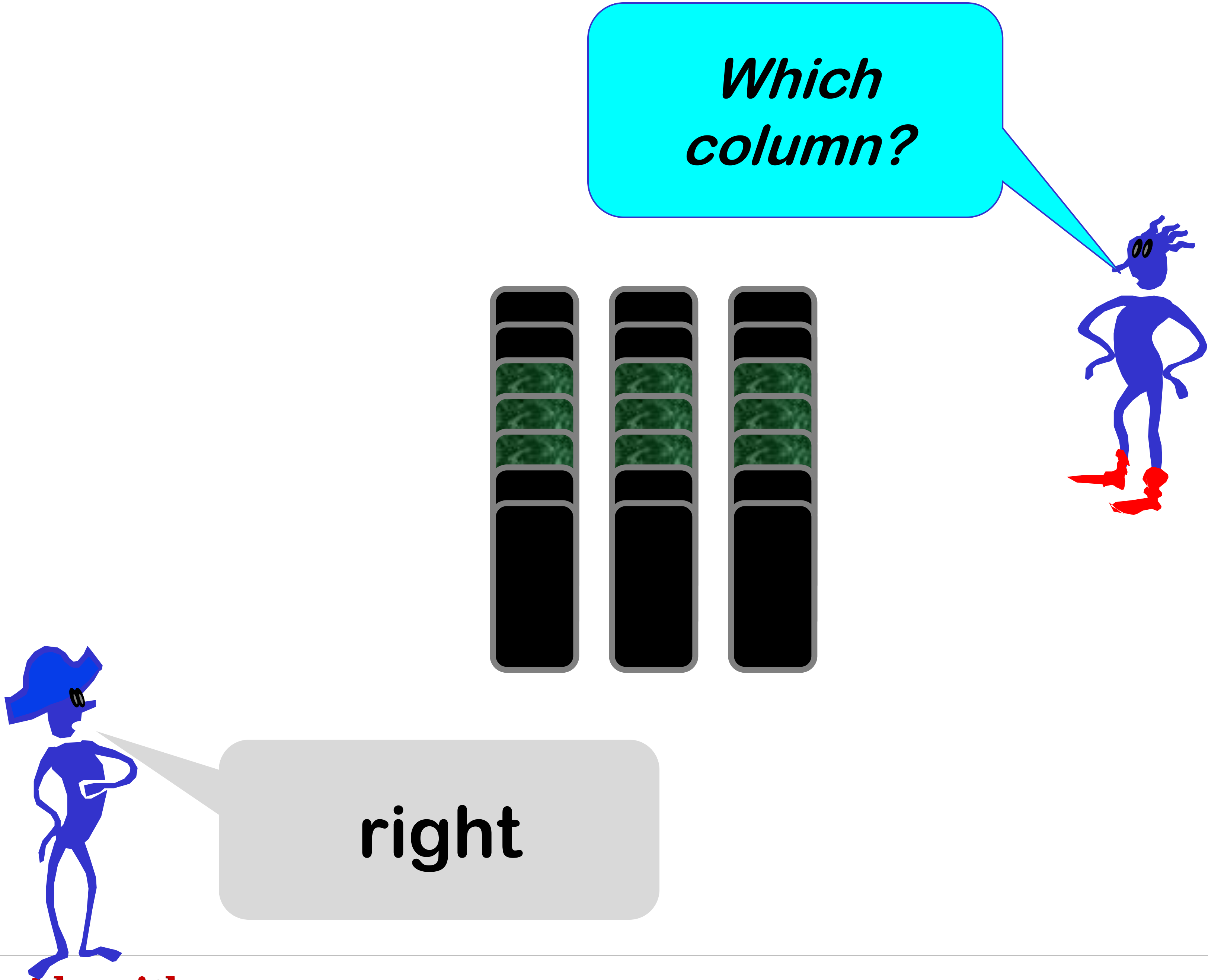




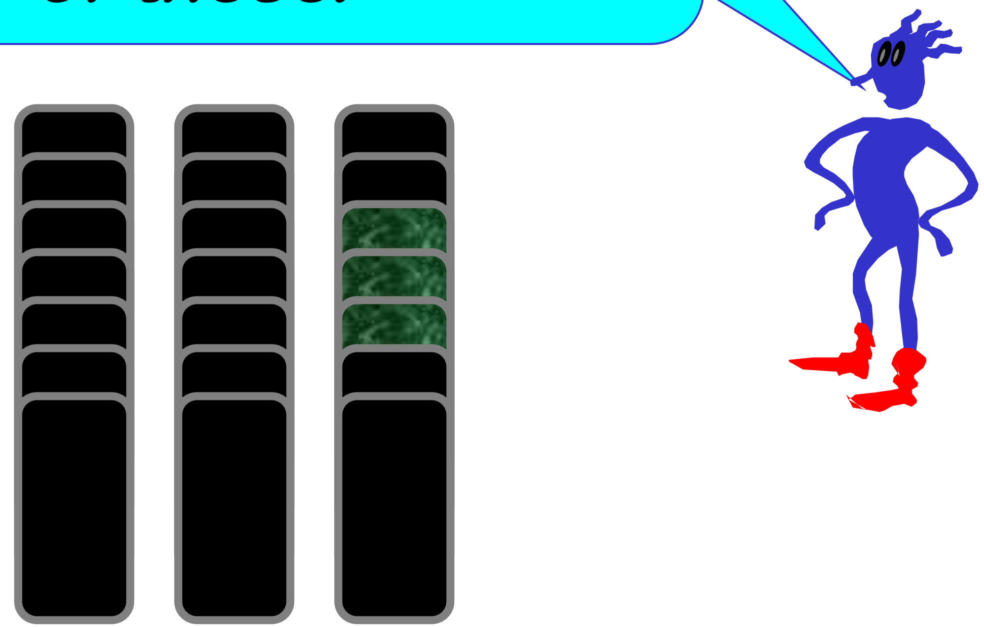

Selected column is placed
in the middle


I will rearrange the cards

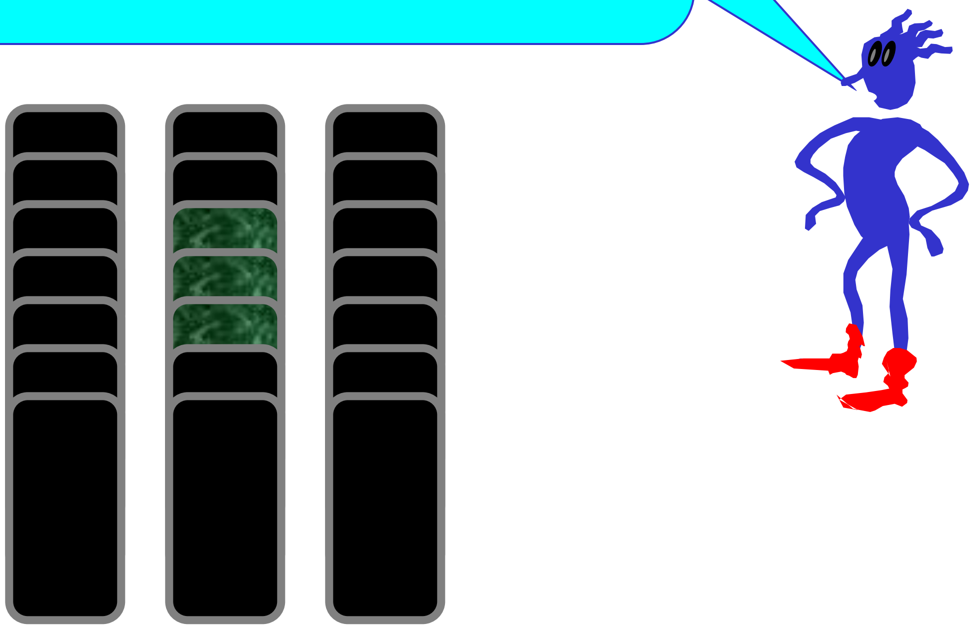

Relax Loop Invariant:
I will remember the same
about each column.

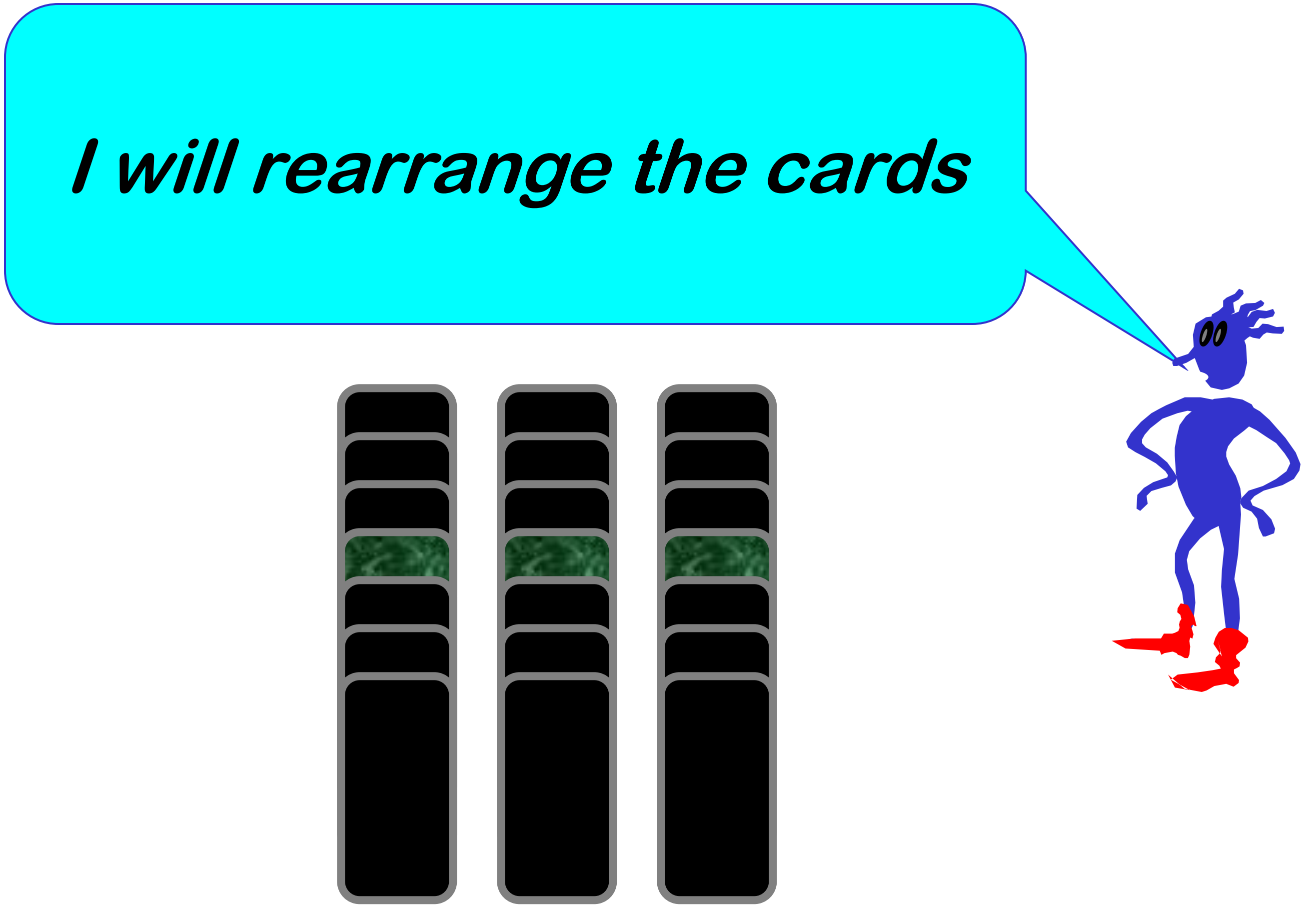

Which column?
right

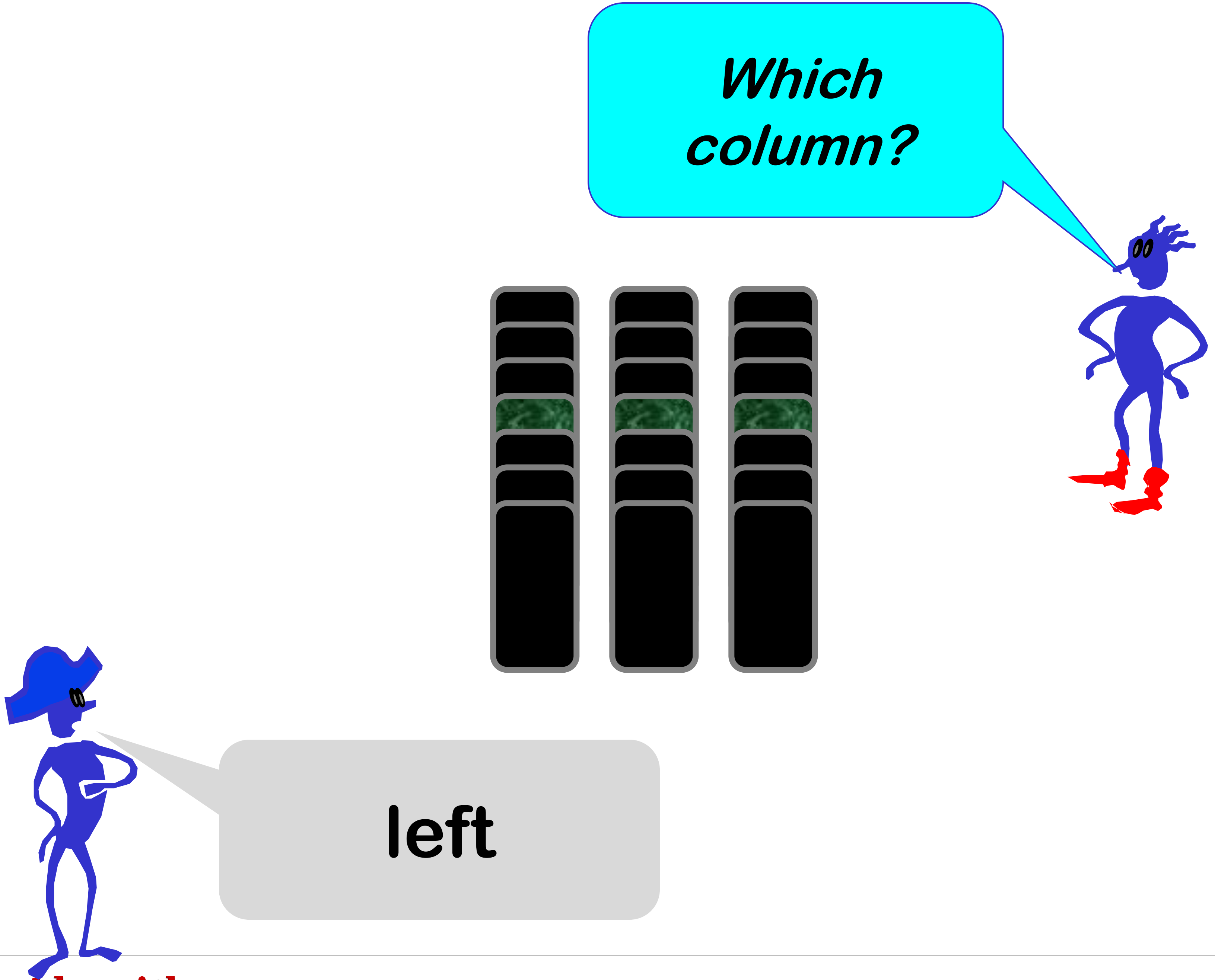
Loop Invariant:
The selected card is one of these.

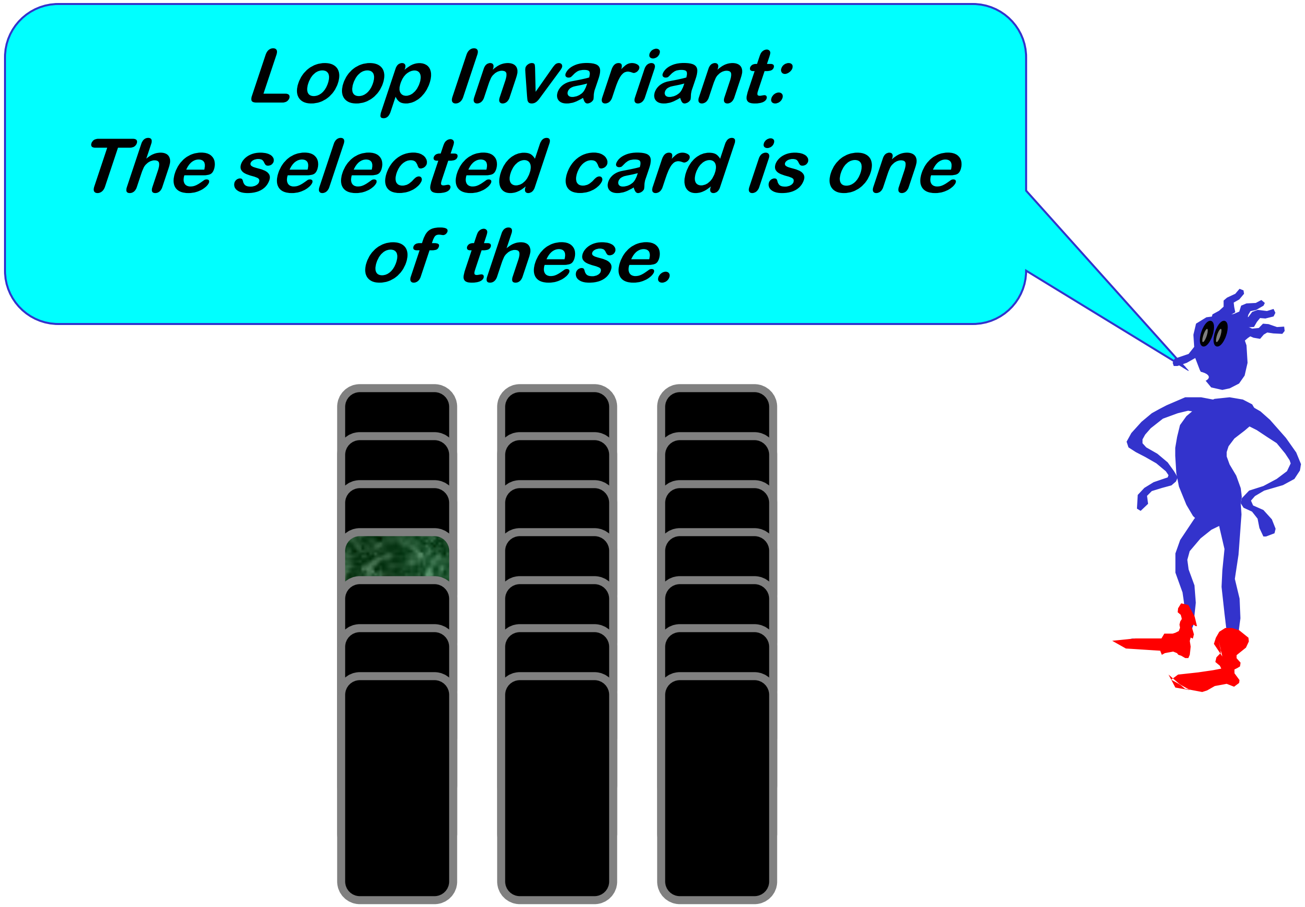


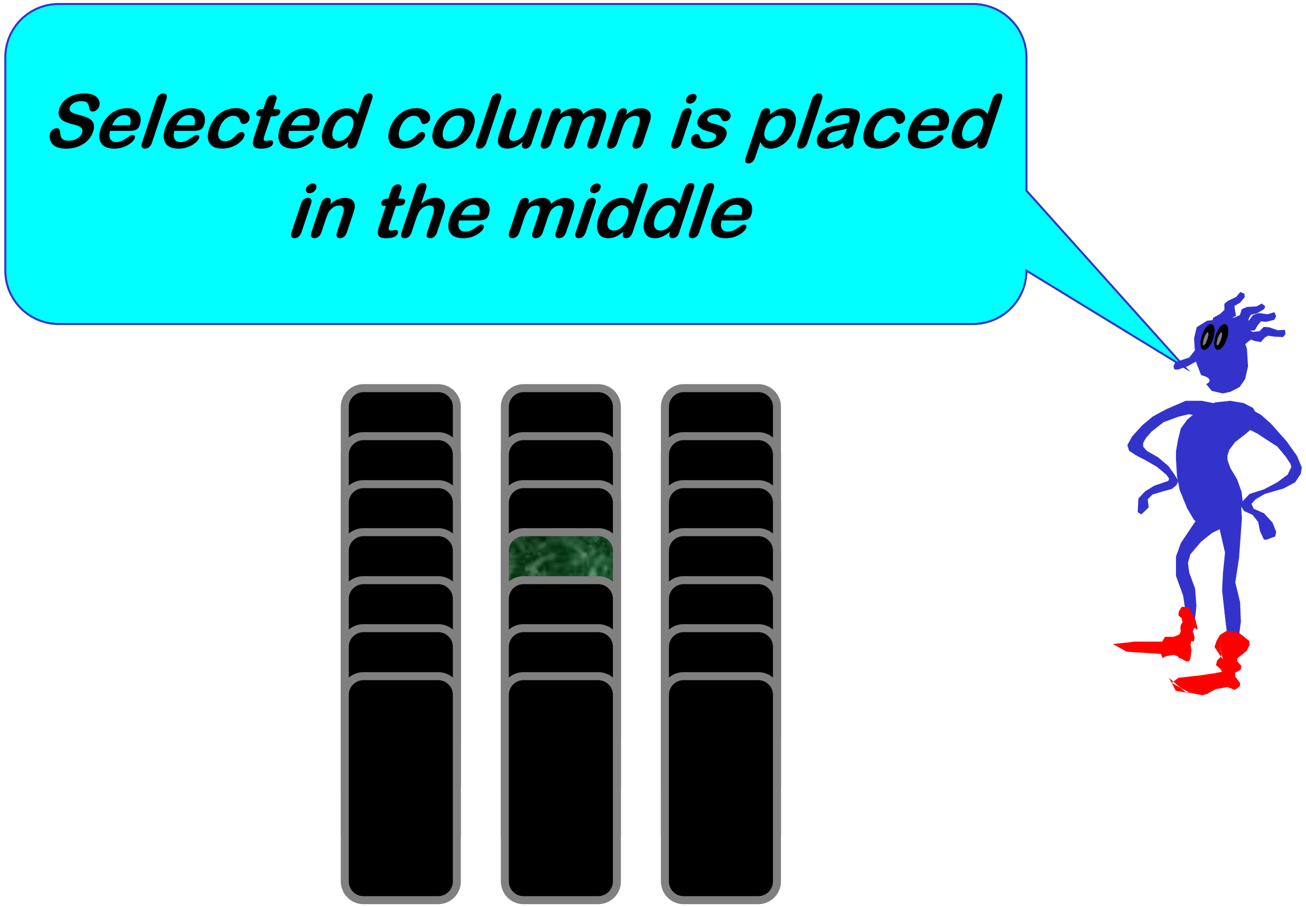
Selected column is placed in the middle

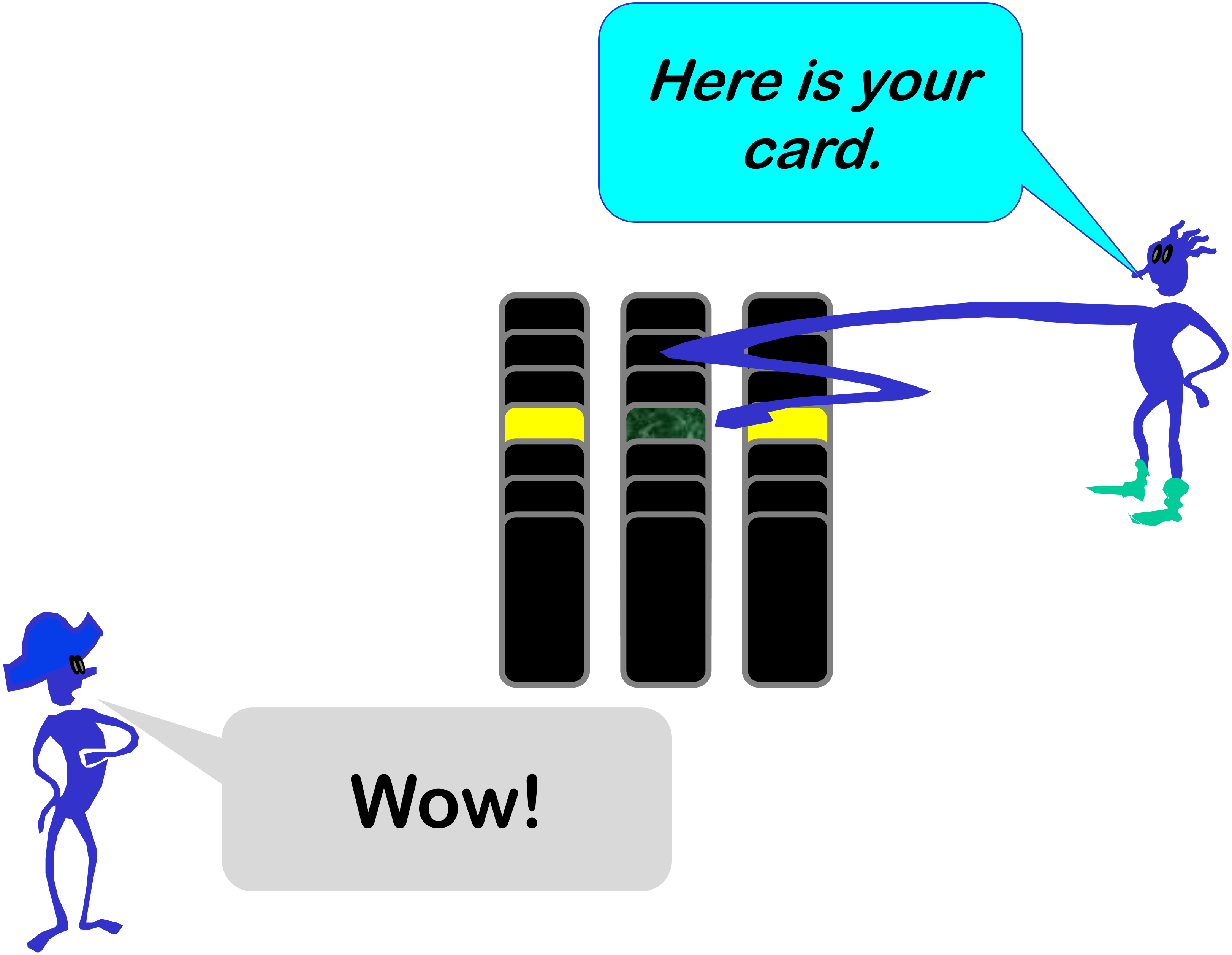



I will rearrange the cards


Which column?
left

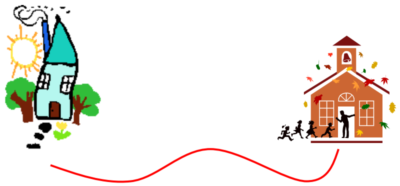

Loop Invariant:
The selected card is one
of these.


Selected column is placed
in the middle


Here is your card.
Wow!

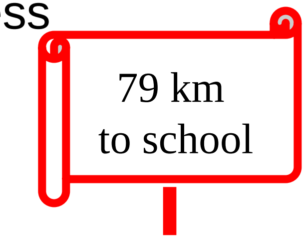
Algorithm Definition Completed

Define Problem

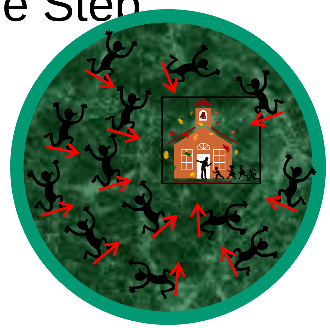


Define Loop Invariants

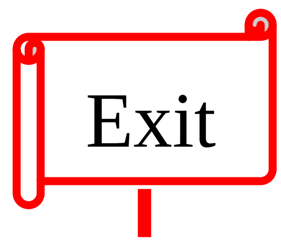
Define Measure of Progress



Define Step

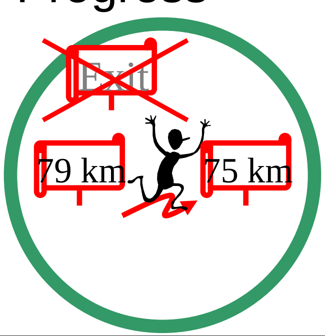


Define Exit Condition

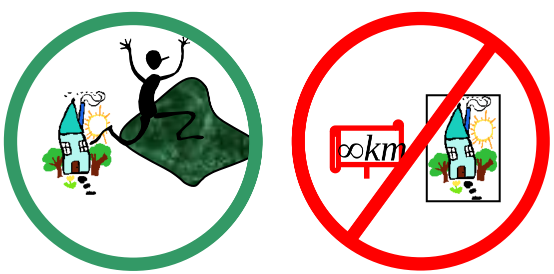


Maintain Loop Inv

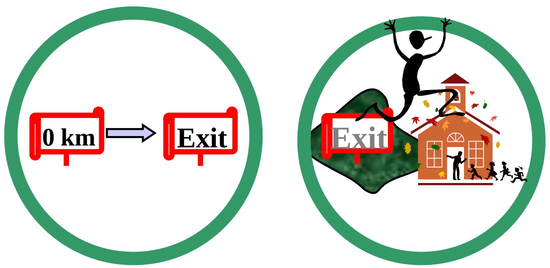
Make Progress



Initial Conditions



Ending



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

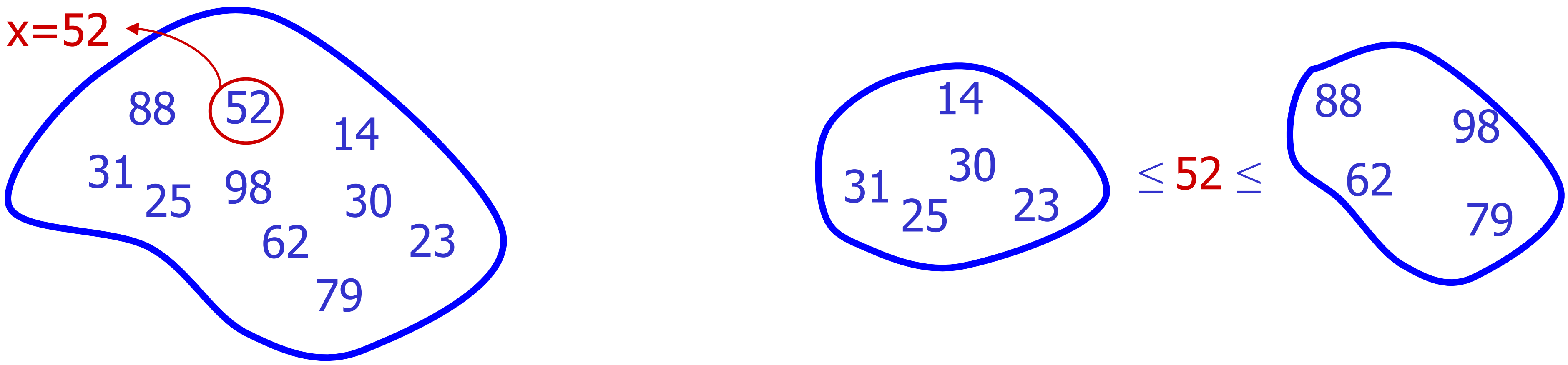
(Not a search problem:

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

The “Partitioning” Problem

Input:

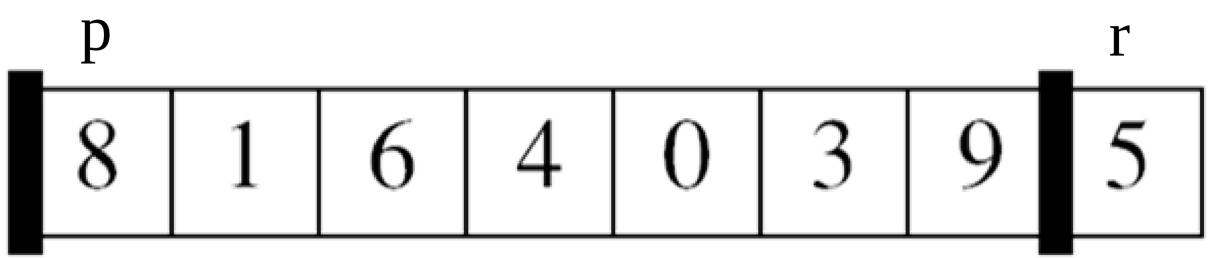
Output:



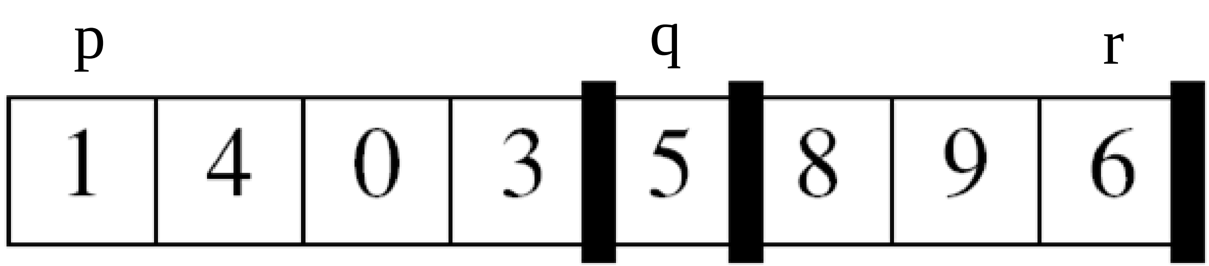
Problem: Partition a list into a set of small values and a set of large values.

Precise Specification

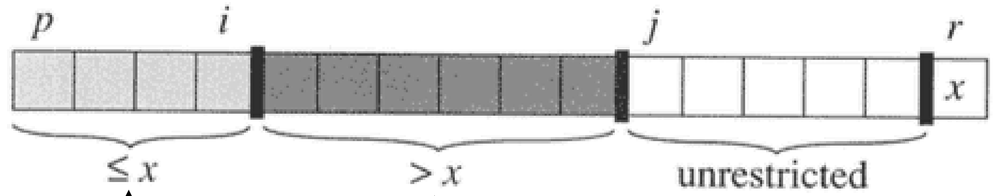
Precondition: $A[p \ldots r]$ is an arbitrary list of values. $x = A[r]$ is the pivot.



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

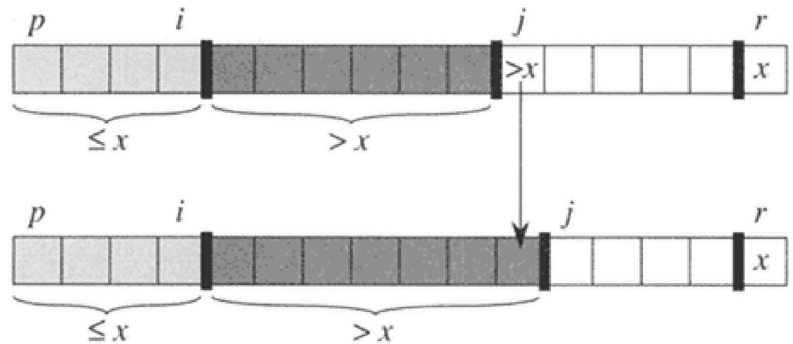
Loop invariant:

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

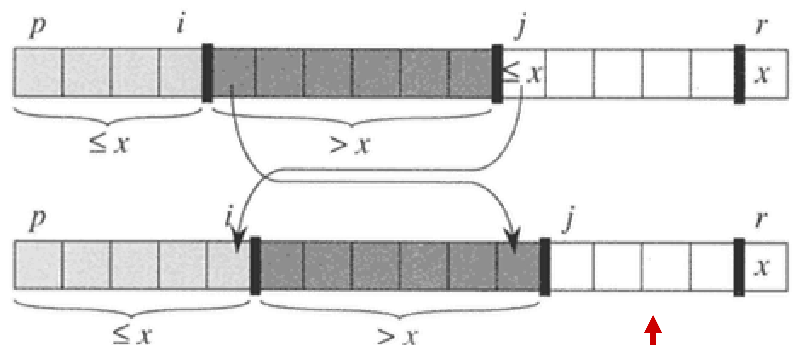
Maintaining Loop Invariant

- Consider element at location j

  – If A[j] > pivot x, incorporate into '> set' by incrementing j.



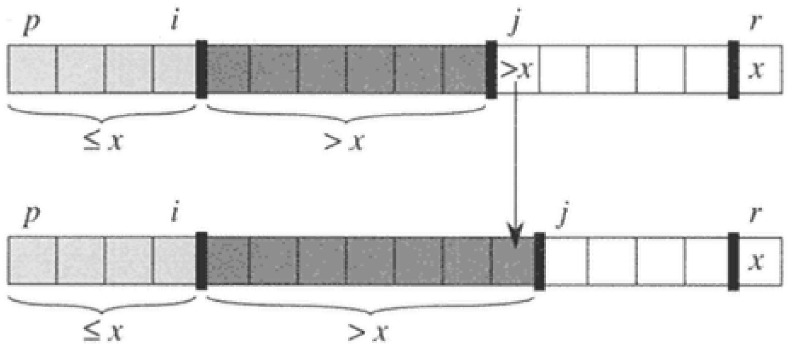
  – If A[j] ≤ pivot x, incorporate it into '≤ set' by swapping with element at location i+1 and incrementing both i and j.

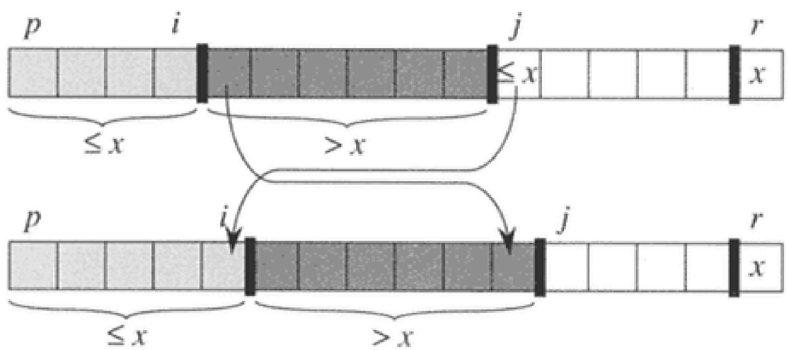


  – Measure of progress: size of unprocessed set.

Maintaining Loop Invariant

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
      do if A[j] ≤ x
            then i ← i + 1
                  exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```





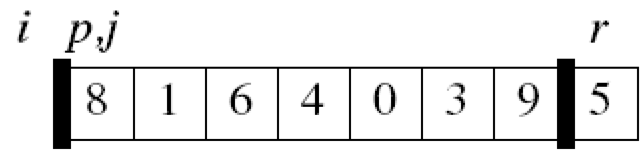
Loop invariant:

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 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 ← A[r]
i ← p − 1
for j ← p to r − 1            // Exit "for loop" when j = r
    do if A[j] ≤ x
        then i ← i + 1
             exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

p i $r = j$ on exit

1	4	0	3	6	8	9	5

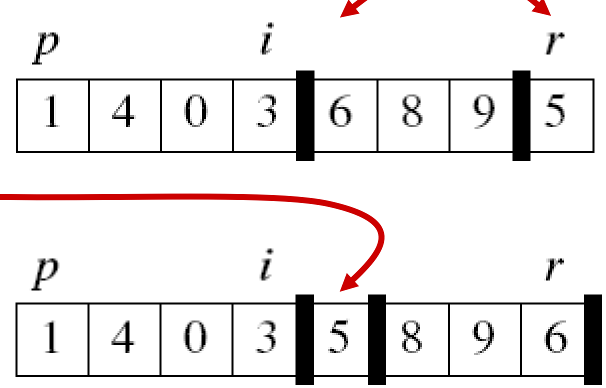
Exhaustive on exit

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

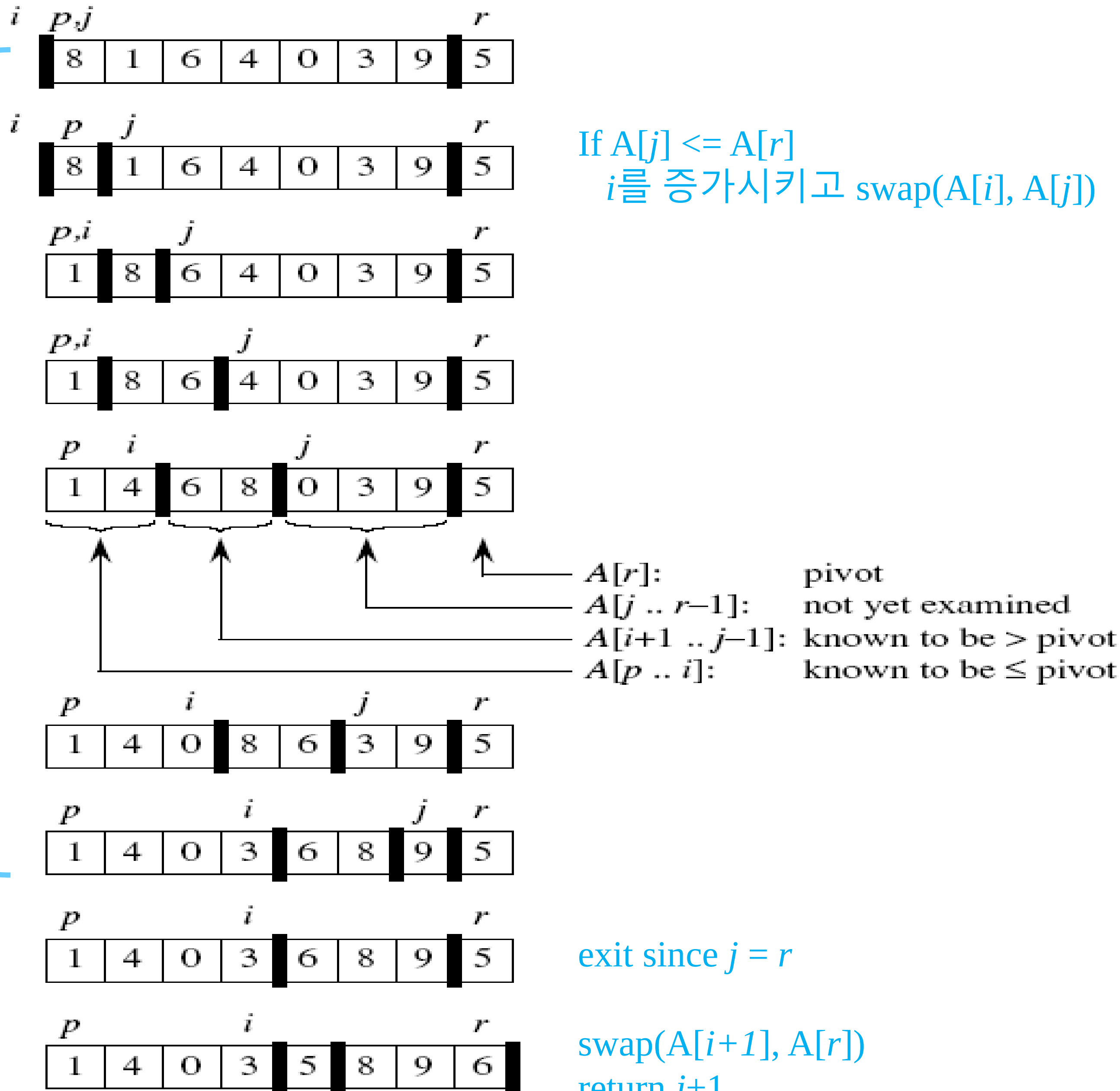
```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
        do if A[j] ≤ x
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```



Example

For loop

If A[j] <= A[r]
i를 증가시키고 swap(A[i], A[j])

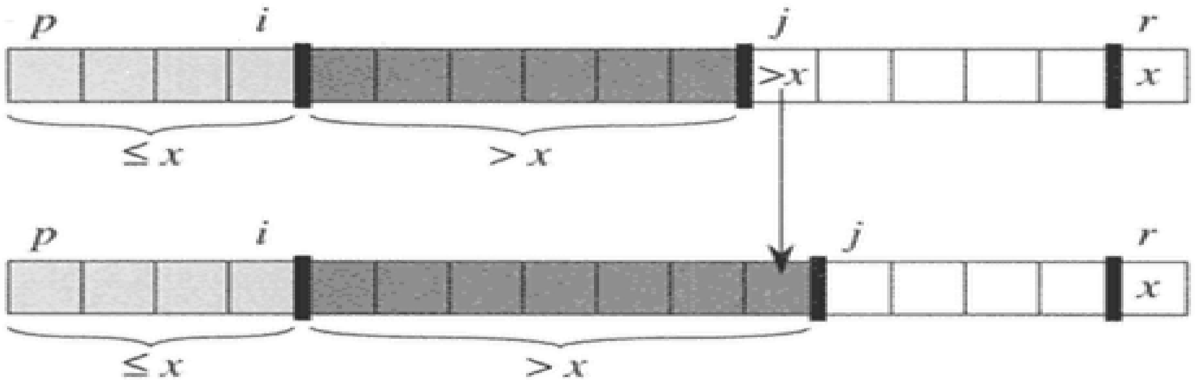


exit since $j = r$

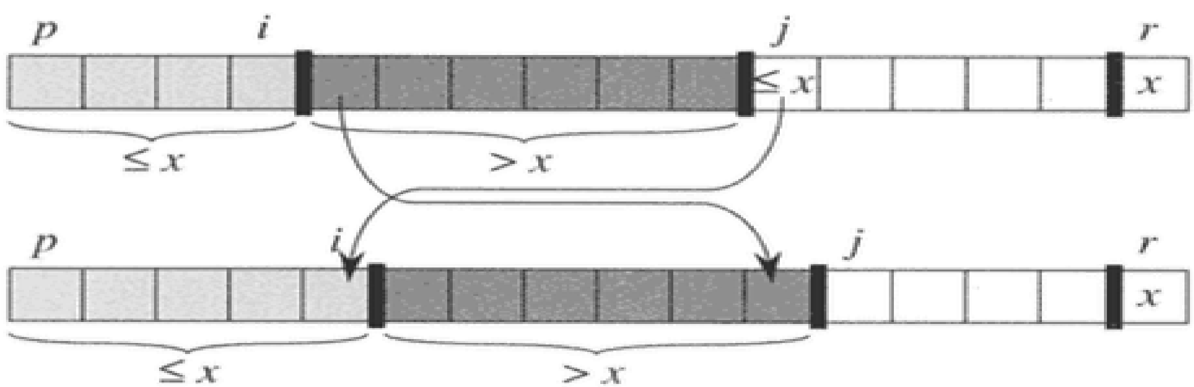
swap(A[$i+1$], A[r])
return $j+1$

Running Time

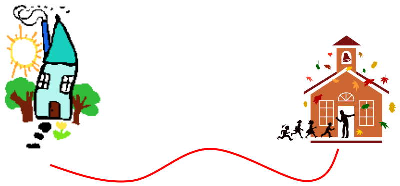
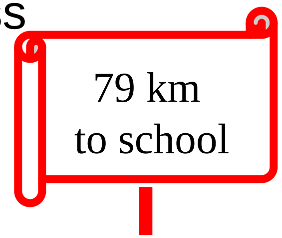
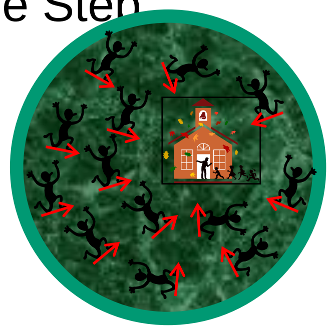
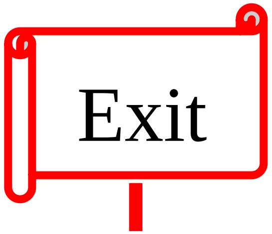
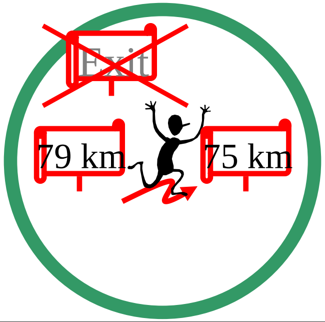
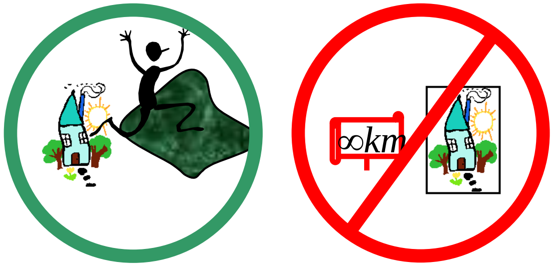
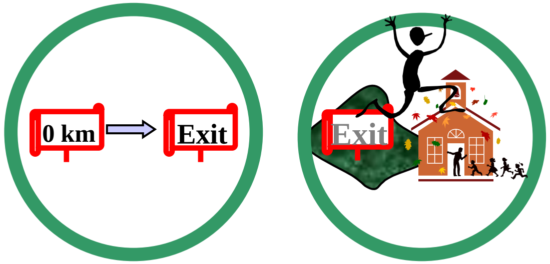
Each iteration takes θ(1) time → Total = θ(n)



or



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 

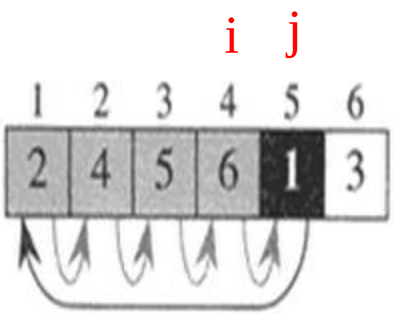
More Examples of Iterative Algorithms

Using Constraints on Input to Achieve Linear-Time Sorting

Example: Insertion Sort

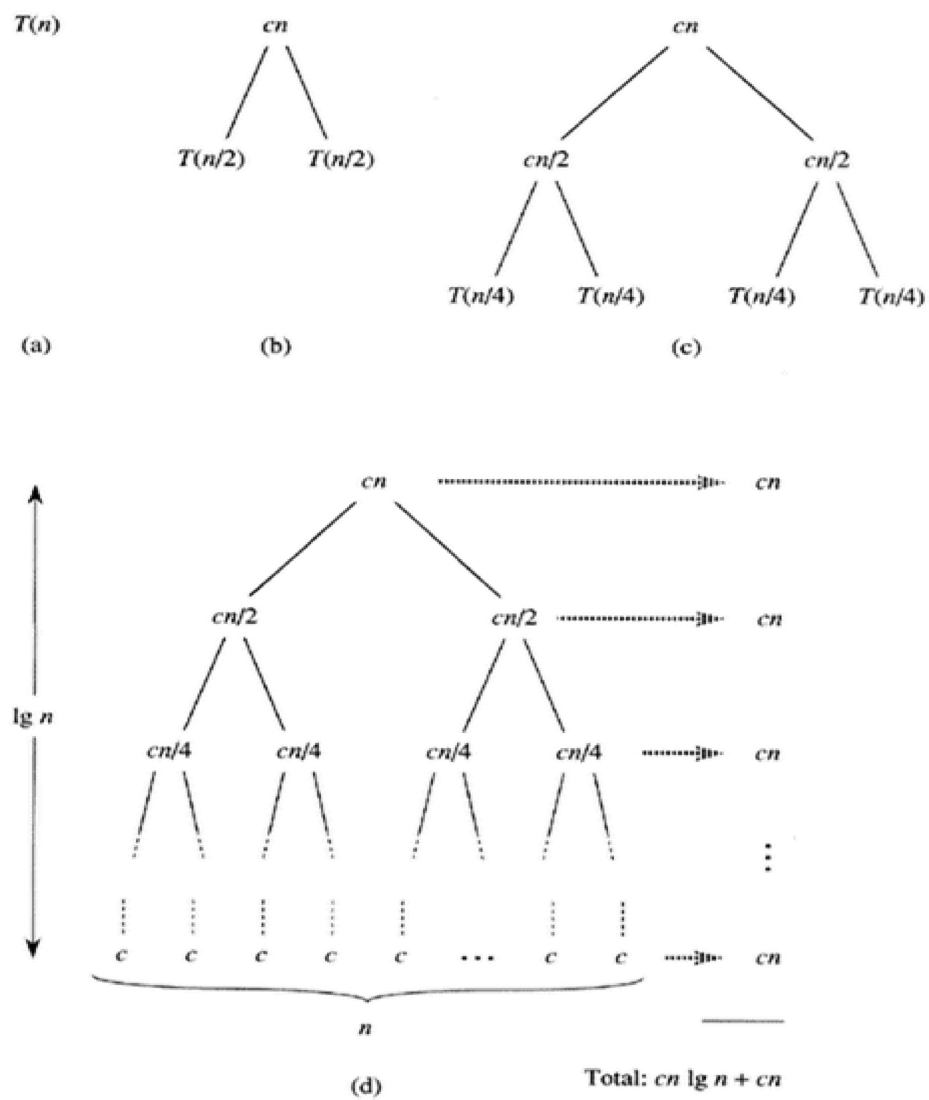
```
Insertion-Sort(A)
for j = 2 to A.length
        key = A[j]
        // Insert key into the sorted A[1..j-1]
        i = j – 1
        while i > 0 and A[j] > key
                A[i+1] = A[i]
                i = i - 1
        A[i+1] = key
```

cost	times
c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	$\sum_{j=2}^{n} (t_j - 1)$
c_8	$n-1$



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

Recall: MergeSort



$$T(n) = 2T\left({}^{n}/_{2}\right) + n$$

$$= 2\left(2T\left({}^{n}/_{2^2}\right) + n/2\right) + n$$
$$= \ 2^2T\left({}^{n}/_{2^2}\right) + 2n$$

$$= 2^2\left(2T\left({}^{n}/_{2^3}\right) + {}^{n}/_{2^2}\right) + 2n$$
$$= 2^3T\left({}^{n}/_{2^3}\right) + 3n$$
$$\ldots$$
$$= 2^kT\left({}^{n}/_{2^k}\right) + kn$$
$$since\ k = \ \log n\ if\ n \ = \ 2^k,$$

$$= nT(1) + n\log n$$
$$\in O(n\log n)$$

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 (ch. 8)

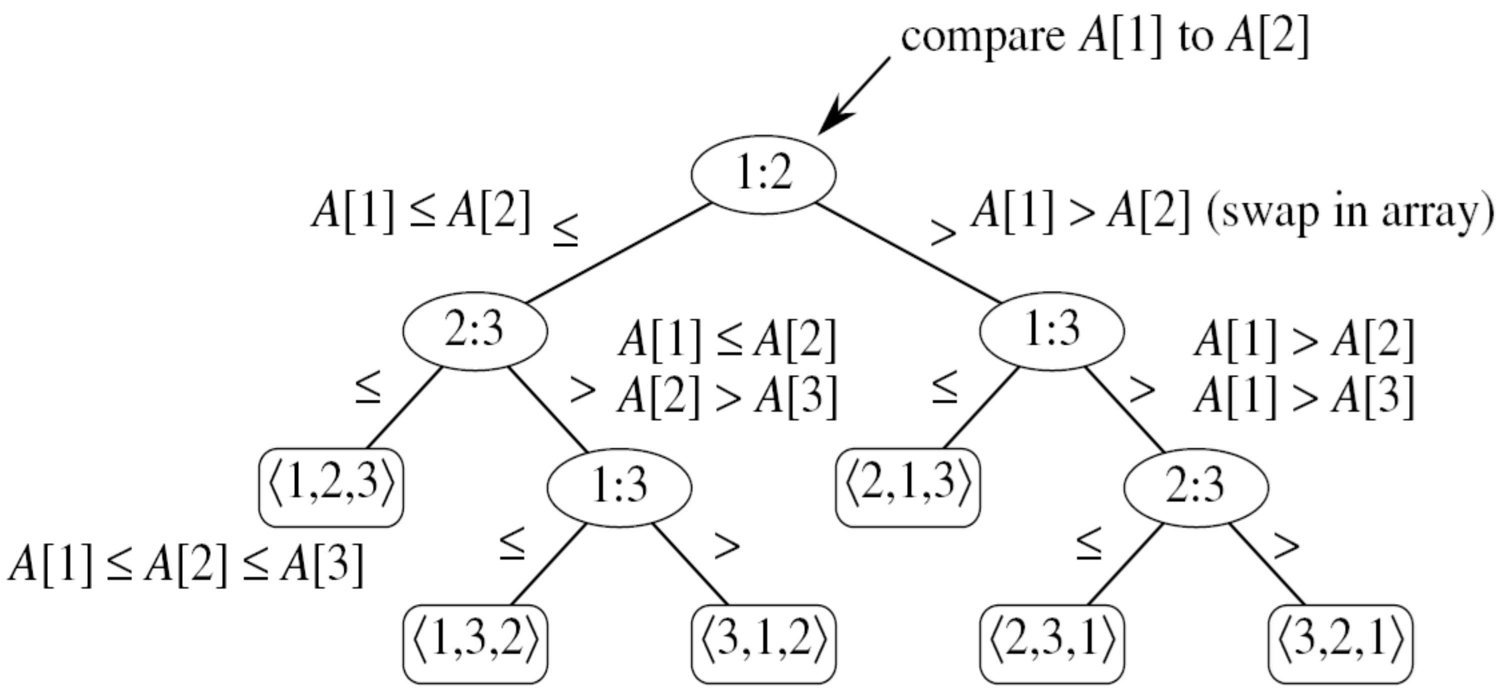
InsertionSort is $\theta(n^2)$.

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

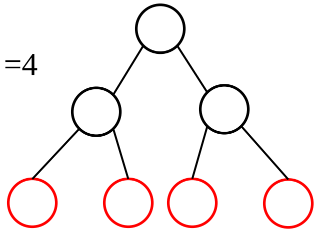
Can we do better?

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 (k) is the log of the number of leaves (L).
  - **$L = 2^{k-1}$ if L is the number of leaves in full binary tree with depth k**
- The leaves of the decision tree represent all possible permutations of the input. How many are there? (n!)
- **Thus, we have:**
  - **$n! \le 2^{k-1}$, where k = the depth of the decision tree**
  - **Then, $k \ge \log_2 n! + 1$ => $k \ge \log_2 n!$**
- **Since $\log_2 n! \in \Omega(n\log n)$, MergeSort is asymptotically optimal.**

Linear Sorts

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 (입력의 성질을 제한한다면)

Example 1. Counting Sort

- **Counting Sort** applies when the elements to be sorted (input) 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 in [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 (키이 값을 변환 위치를 결정하기 위해 각 키 값의 빈도를 구합니다.)
  - Go through the records in order putting them where they go (입력 값들이 놓일 곳에 그들은 위치시키면서 순서대로 읽어 나갑니다).

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	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!

Counting Sort

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? $\Theta(N)$

Counting Sort

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? $\Theta(k)$

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	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.

Counting Sort

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
	0	5	14	17

Location of first record
with digit v.

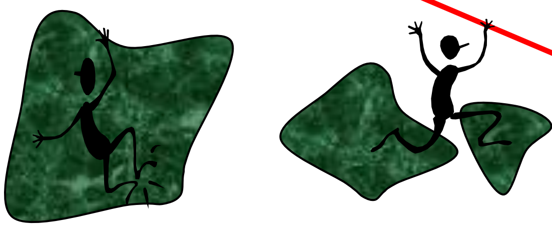
Algorithm: Go through the records in order
putting them where they go.

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 *[0..k-1]*.

Counting Sort

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



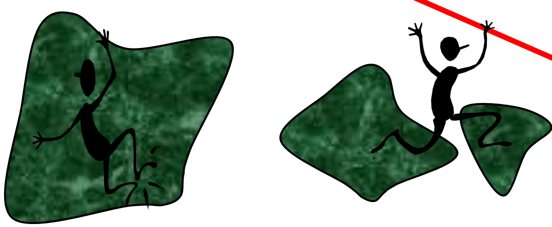
Value v:	0	1	2	3
	0	5	14	17

Location of next record with digit v.

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

Counting Sort

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



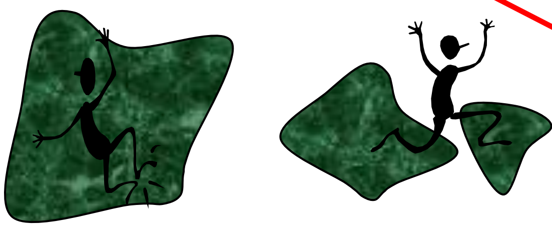
Value v:	0	1	2	3
	0	6	14	17

Location of next record with digit v.

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

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



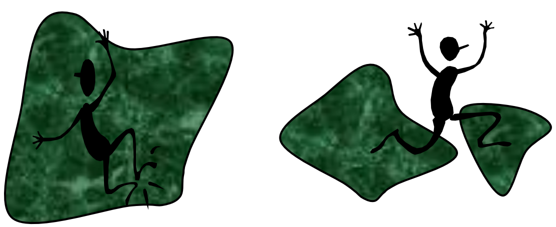
Value v:	0	1	2	3
	1	6	14	17

Location of next record with digit v.

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

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



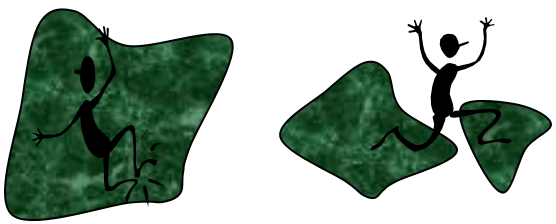
Value v:	0	1	2	3
	2	6	14	17

Location of next record with digit v.

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

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



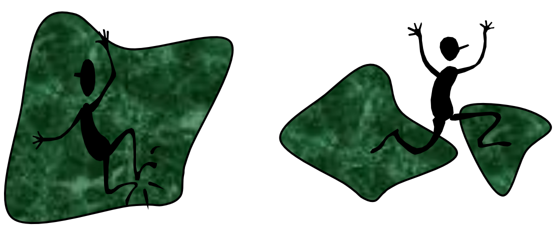
Value v:	0	1	2	3
	2	7	14	17

Location of next record with digit v.

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

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



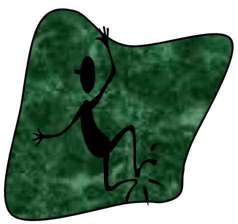
Value v:	0	1	2	3
	2	7	14	18

Location of next record with digit v.

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

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				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



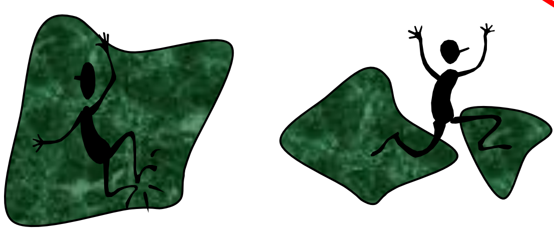
Value v:	0	1	2	3
	2	8	14	18

Location of next record with digit v.

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

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				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



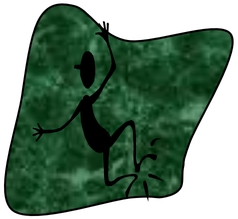
Value v:	0	1	2	3
	2	9	14	18

Location of next record with digit v.

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

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				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



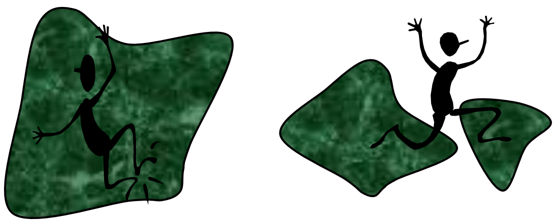
Value v:	0	1	2	3
	2	9	14	19

Location of next record with digit v.

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

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			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



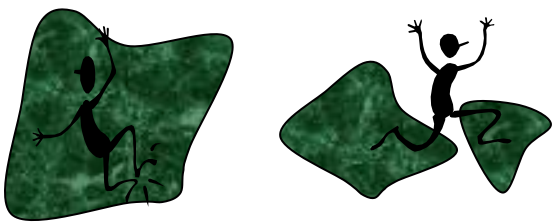
Value v:	0	1	2	3
	2	10	14	19

Location of next record with digit v.

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

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			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



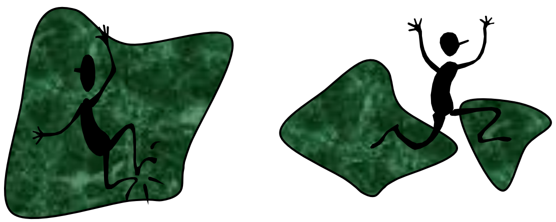
Value v:	0	1	2	3
	3	10	14	19

Location of next record with digit v.

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

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			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



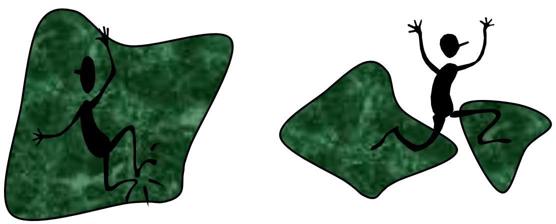
Value v:	0	1	2	3
	3	10	15	19

Location of next record with digit v.

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

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			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



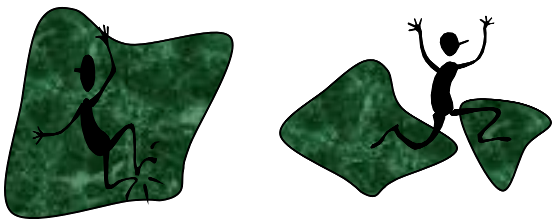
Value v:	0	1	2	3
	3	10	15	19

Location of next record with digit v.

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

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	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 = $\Theta(N)$

Total = $\Theta(N+k)$

Example 2. RadixSort

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).

344
125
333
134
224
334
143
225
325
243

125
224
225
325
333
134
334
344
143
243

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

RadixSort

344	125	125
125	134	224
333	143	225
134	224	325
224	225	134
334	243	333
143	344	334
225	333	143
325	334	243
243	325	344

Sort wrt which digit first?

The most significant.

Sort wrt which digit Second?

The next most significant.

The meaning in first sort is 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

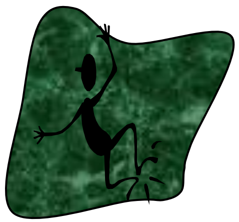
i+1

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

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 sorted wrt high order digit

RadixSort

2 24

1 25

2 25

3 25

3 33

1 34

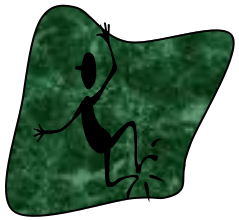
3 34

1 43

2 43

3 44

i+1



Is sorted wrt first i digits.

Sort wrt (i+1)st 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 & stable sort maintained

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

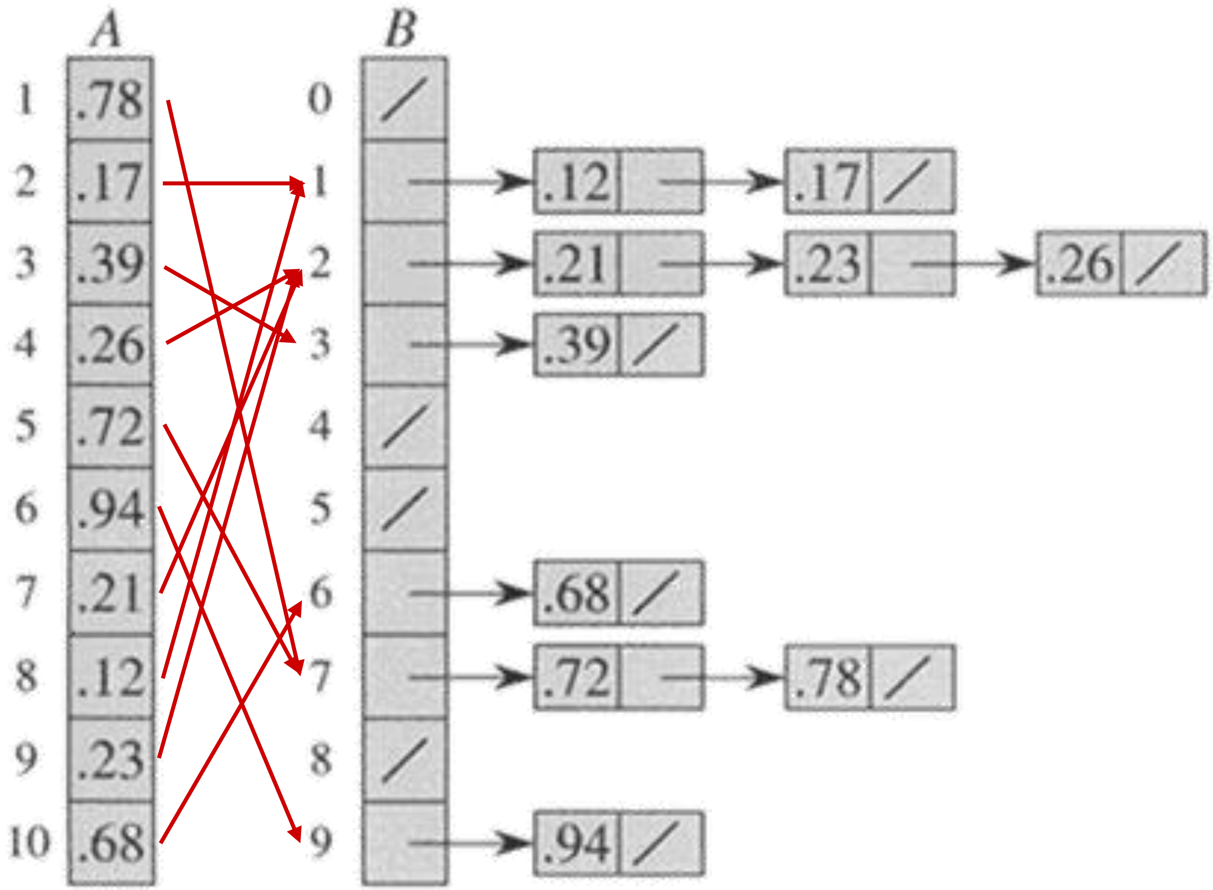
k = # of possible values for each digit

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 $\Theta(n)$.

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

전체 정렬할 갯수가 n이기 때문에 리스트 B[i]를 정렬하는 복잡도는 평균 θ(1)이다.

$\text{BUCKET-SORT}(A, n)$

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$ ← $\Theta(1)$

for $i \leftarrow 0$ **to** $n - 1$

  do sort list $B[i]$ with insertion sort ← $\Theta(1) \times n$

concatenate lists $B[0], B[1], \ldots, B[n-1]$ ← $\Theta(n)$

return the concatenated lists

$\Theta(n)$

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