SUPSI

Searching and sorting

Searching

Searching

- Searching is the process of finding a given element (or a set of elements) matching a given set of criteria
  - e.g. find all users which have logged during last week and who have spent more than 100 CHF
- In this context we want to find a specific item in a collection of data items

- **Sequence search** involves finding an item in a collection using a **search key**

- The search key can be either the element in the sequence (for simple types such as integers and reals) or an attribute of a complex data type (e.g. the social security number of a person)

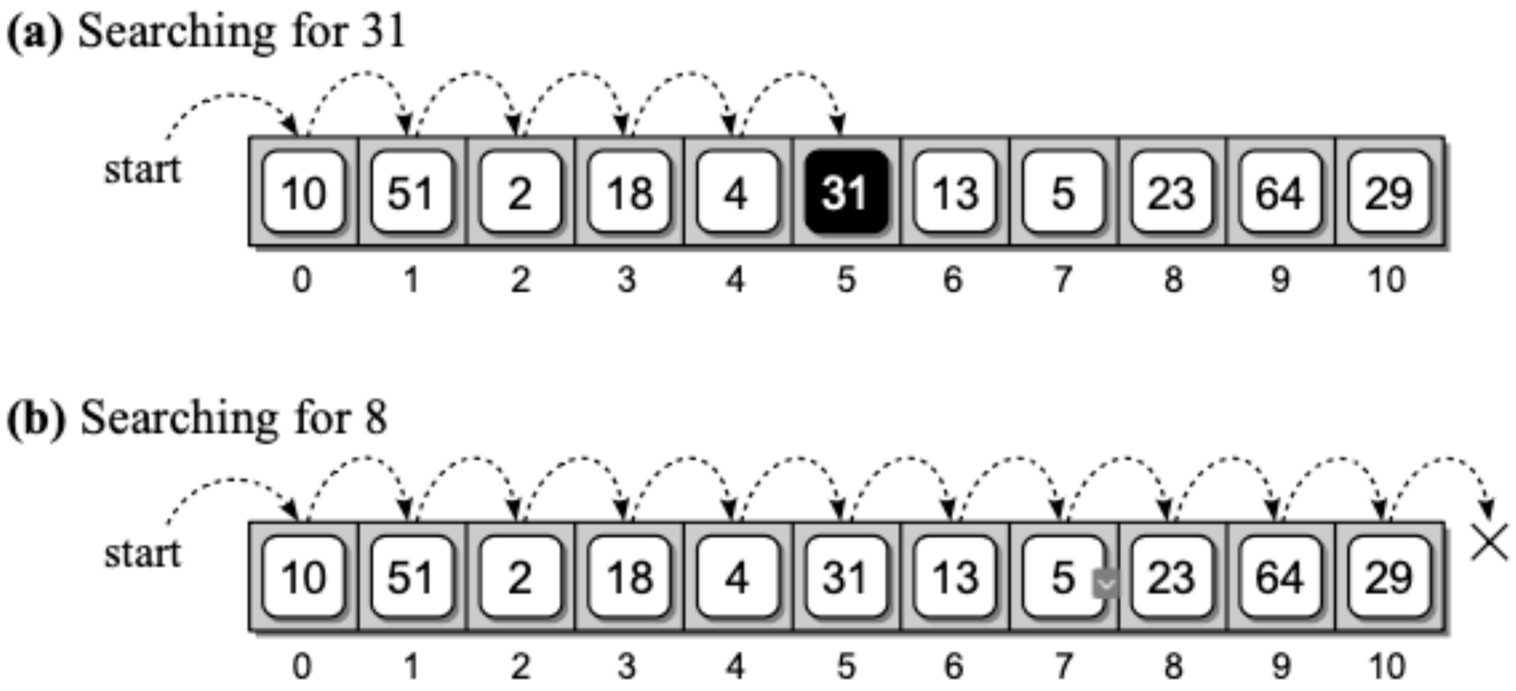
- In some cases the key is composed of multiple elements (e.g. height and weight of a person) and it is therefore called **compound key**

Linear search

- It is the simplest solution: scan the collection, one item at a time, compare the item with the key, if found, return it
- It is already available in Python thanks to the `in` operator

```
if key in theArray :
    print( "The key is in the array." )
else :
    print( "The key is not in the array." )
```

Linear search: how does it work?



You don’t know that 8 is not in the sequence until you have seen all values!

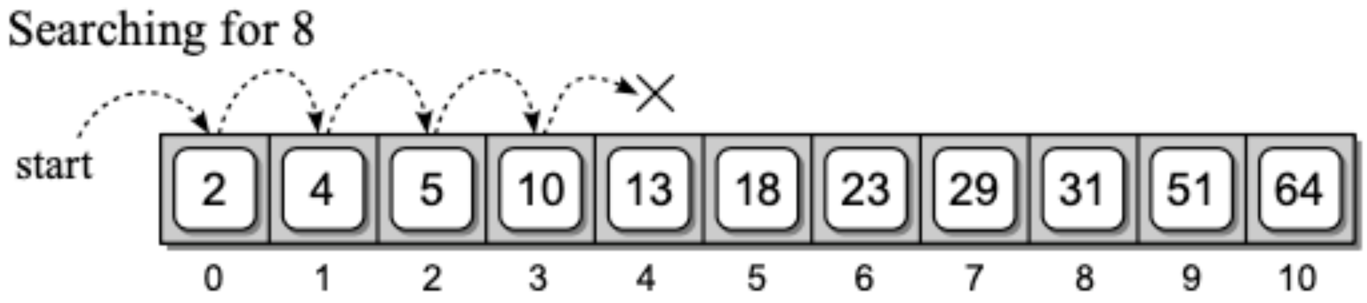
The implementation of linear search

```python
def linearSearch( theValues, target ):
    n = len( theValues )
    for i in range( n ) :
        # If the target is in the i-th element, return True
        if theValues[i] == target:
            return True
    return False # If not found, return False.
```

- In this algorithm the worst case occurs when the algorithm performs the maximum number of steps
- For linear search the worst case time complexity is O(n)

Searching a sorted sequence

- If the sequence is sorted, and we are looking for a given key, once we have been past where the key should have been, then we can quit!



Implementation of searching in a sorted sequence

```
def sortedLinearSearch( theValues, item ):
    n = len( theValues )
    for i in range(n):
        # If the target is found in the ith element, return True
        if theValues[i] == item:
            return True
        # If target is larger than the ith element, it's not in the sequence.
        elif theValues[i] > item:
            return False
    return False # The item is not in the sequence.
```

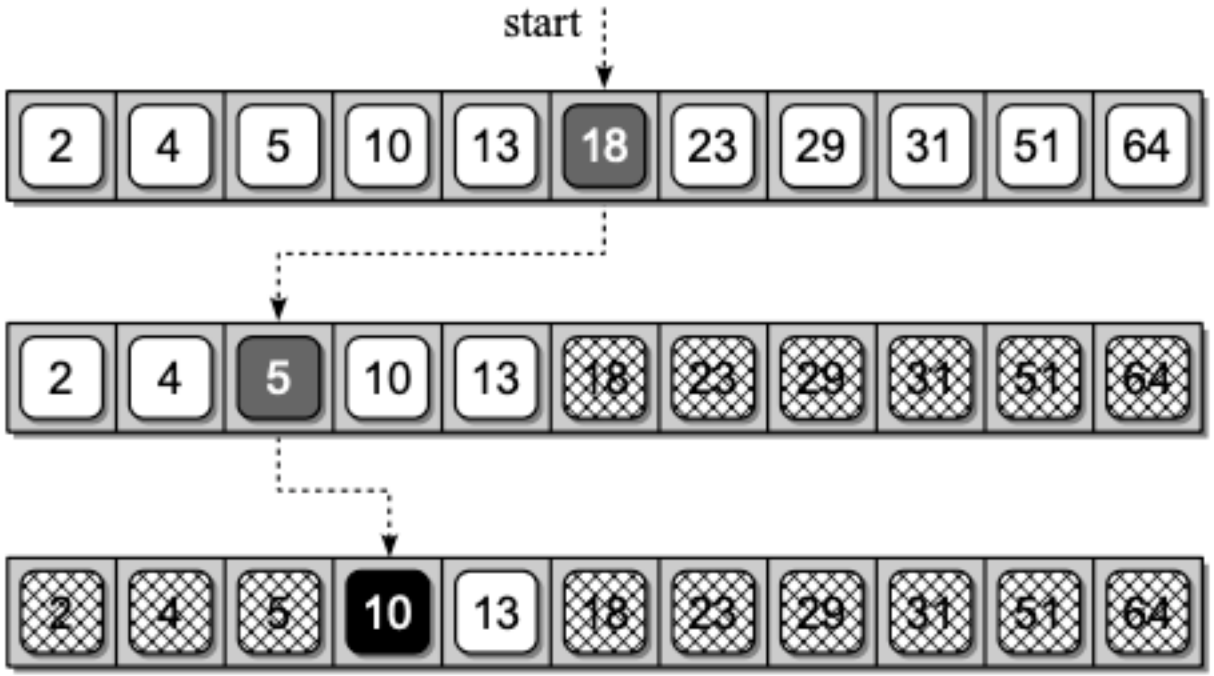
Searching for the minimum value

- It is like a linear search and the complexity is O(n) as it must traverse the whole array.

```
def findSmallest( theValues ):
    n = len( theValues )
    # Assume the first item is the smallest value.
    smallest = theValues[0]
    # Determine if any other item in the sequence is smaller.
    for i in range(1,n):
        if theValues[i] < smallest:
            smallest = theValues[i]
    return smallest # Return the smallest found.
```

Binary search

- Binary search assumes that the sequence is sorted and it exploits the structure thanks to a **divide and conquer** approach
- Example: search for key=10 in this sequence:



Implementation of binary search

- The complexity is $O(\log_2(n))$!

```
def binarySearch( theValues, target ) :
    # Start with the entire sequence of elements.
    low = 0
    high = len(theValues) - 1
    # Repeatedly subdivide the sequence in half until the target is found.
    while low <= high:
    # Find the midpoint of the sequence.
        mid = (high + low) // 2
        # Does the midpoint contain the target?
        if theValues[mid] == target:
            return True
        # Or does the target precede the midpoint?
        elif target < theValues[mid]:
            high = mid - 1
        # Or does it follow the midpoint?
        else:
            low = mid + 1
        # If the sequence cannot be subdivided further, we're done.
    return False
```

Sorting

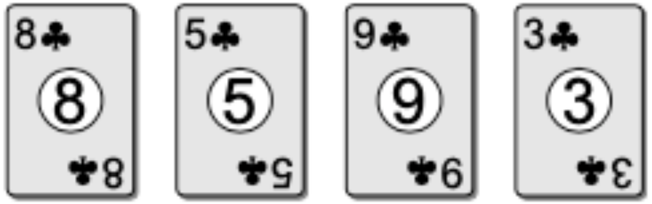
Sorting

- Sorting is the process of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed relationship.
- The ordering of the items is based on a **sort key**
- The ordering of the same data set can be performed according to different keys to present different perspectives

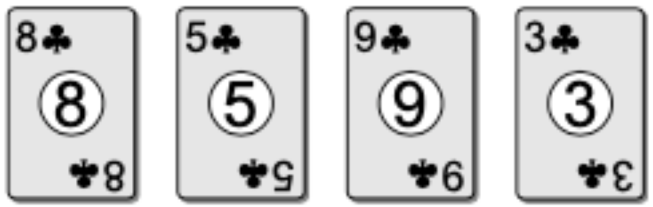
- We present three basic sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort

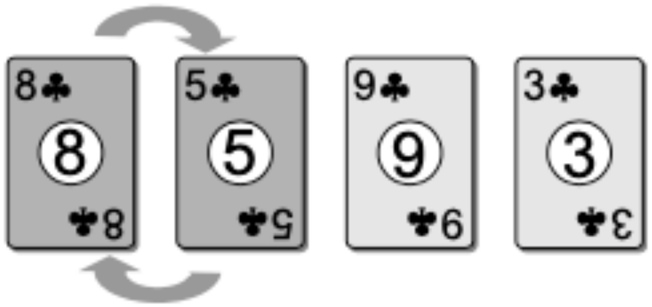
- More advanced algorithms will be introduced later

Bubble sort

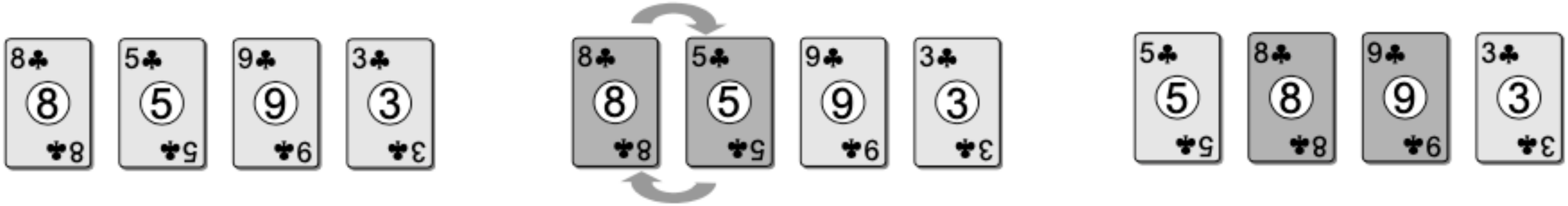


Bubble sort

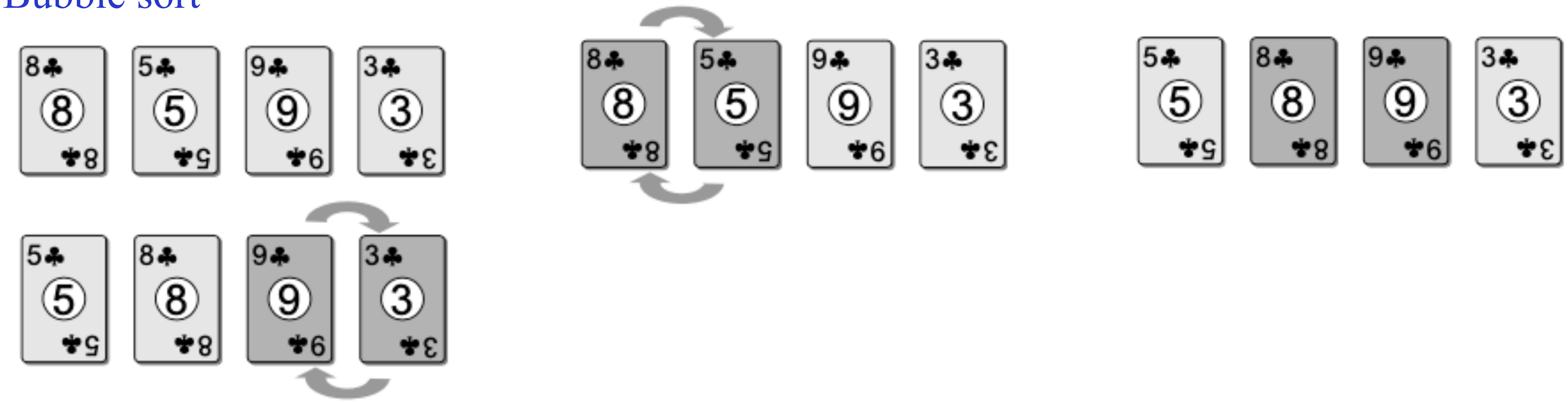




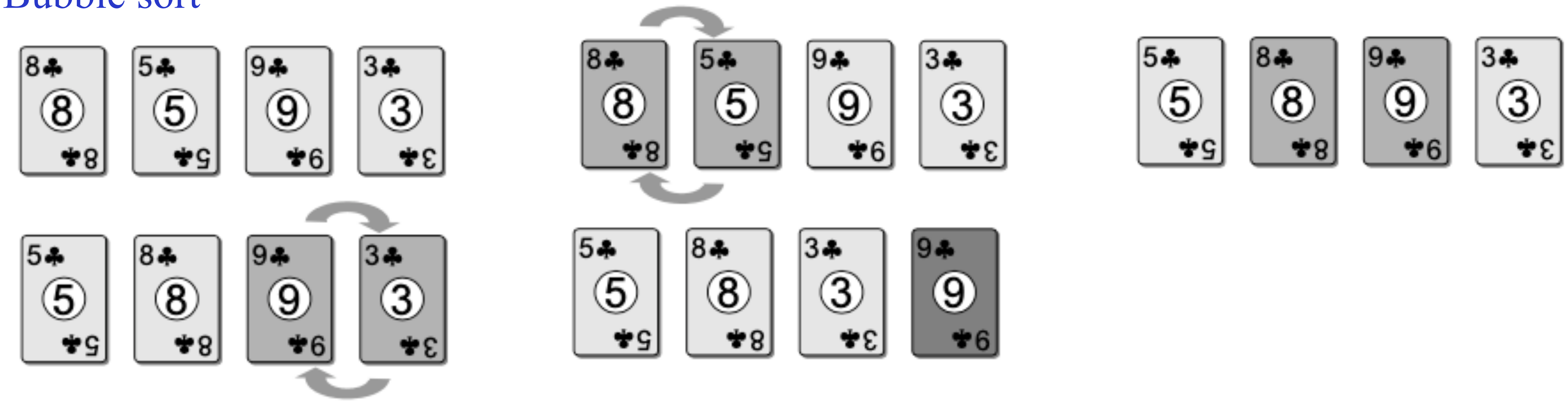
Bubble sort



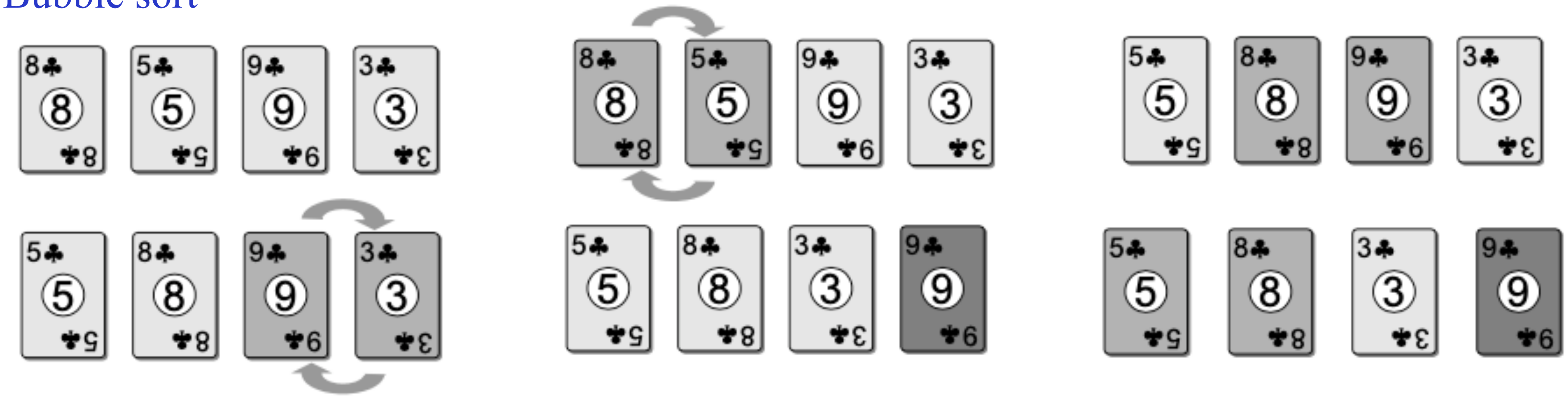
Bubble sort



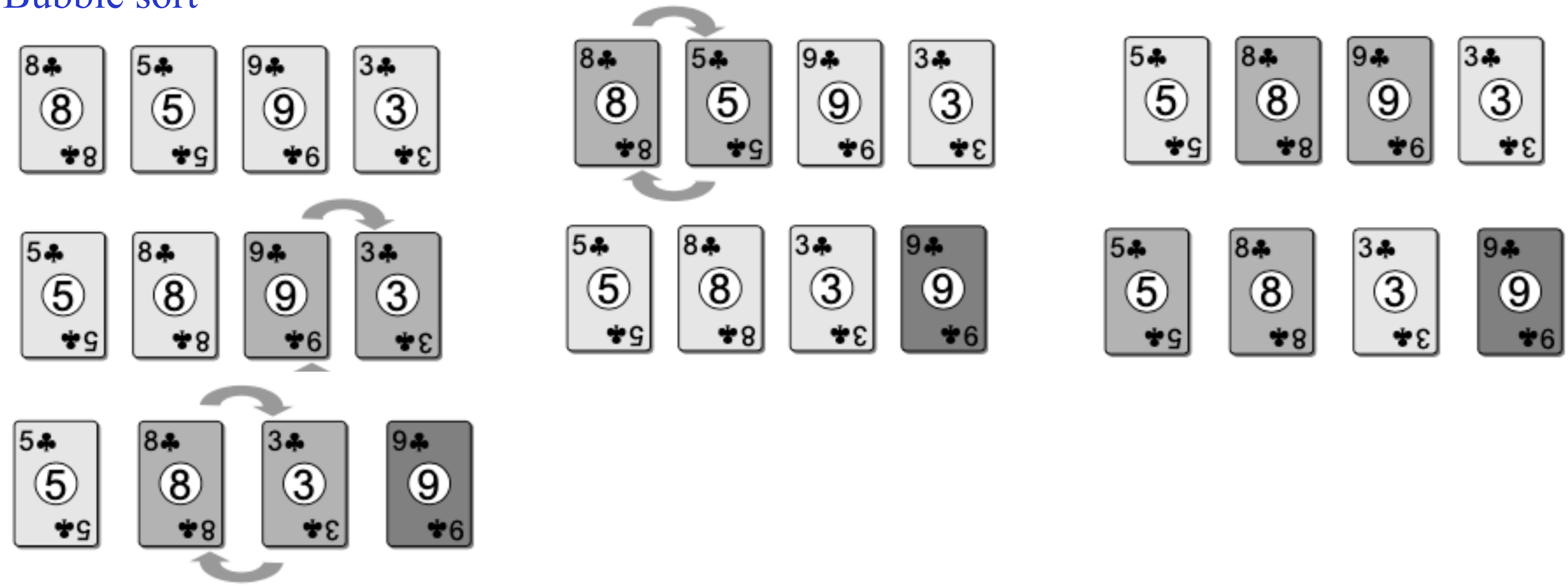
Bubble sort



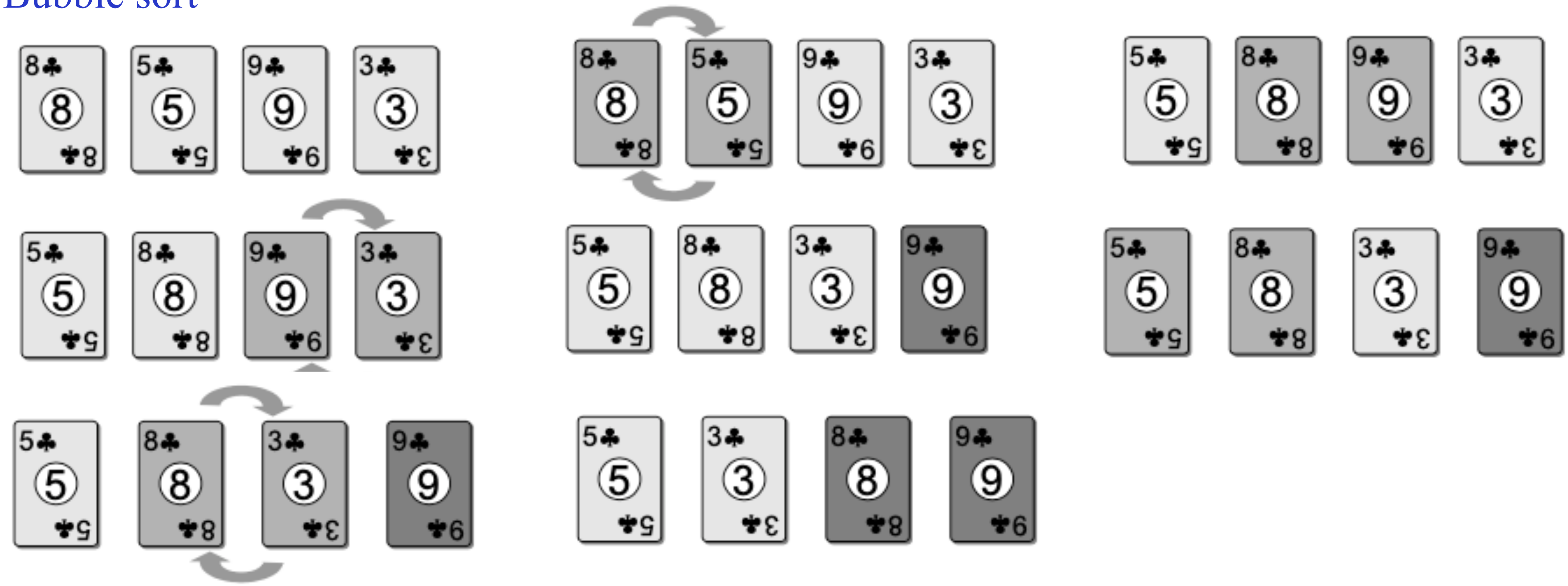
Bubble sort



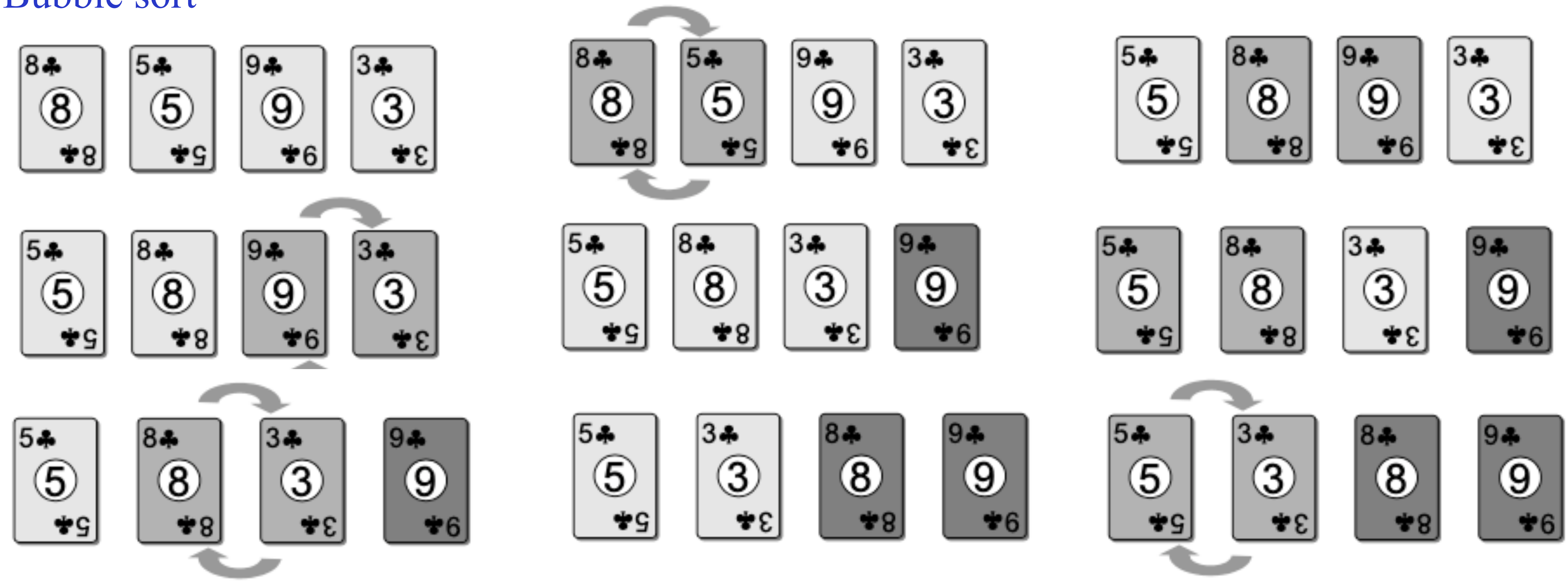
Bubble sort



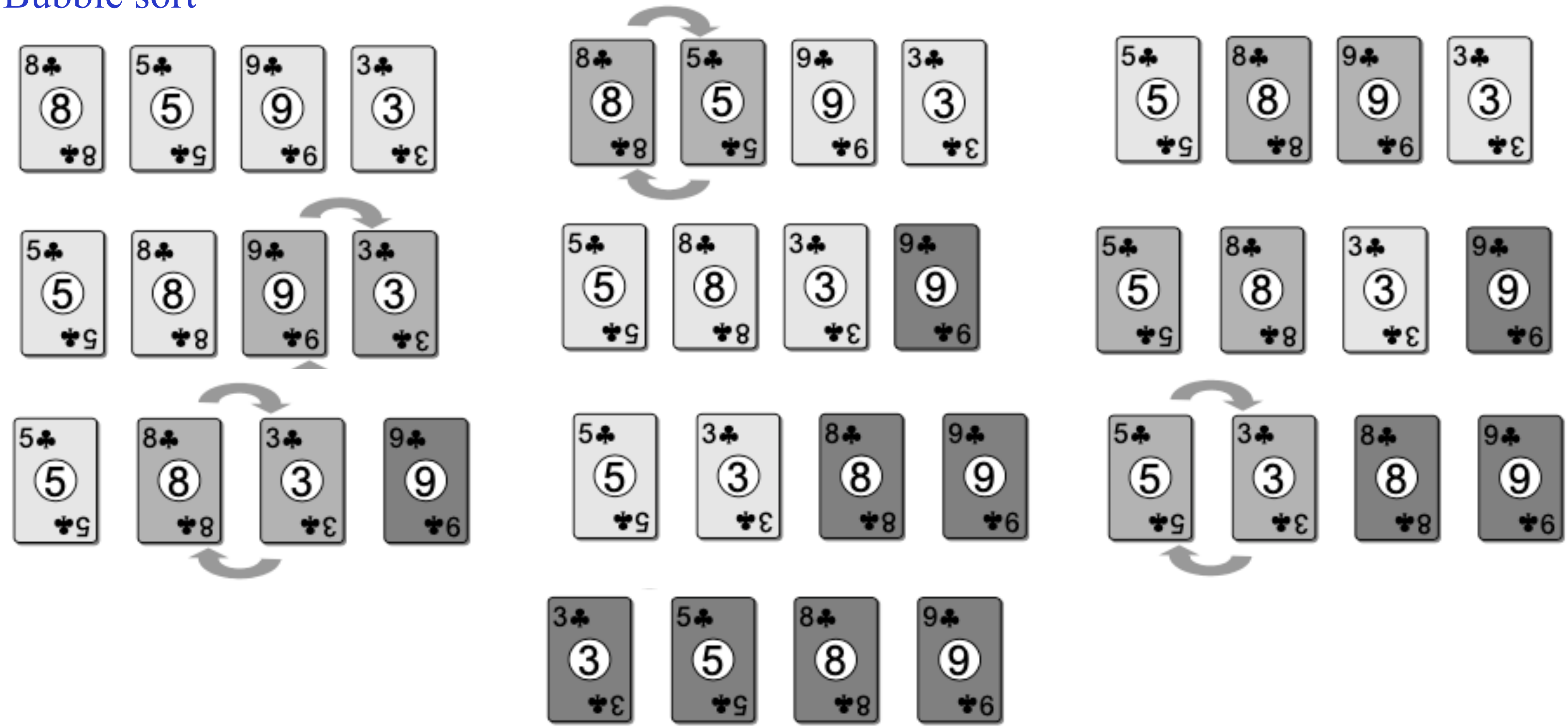
Bubble sort



Bubble sort

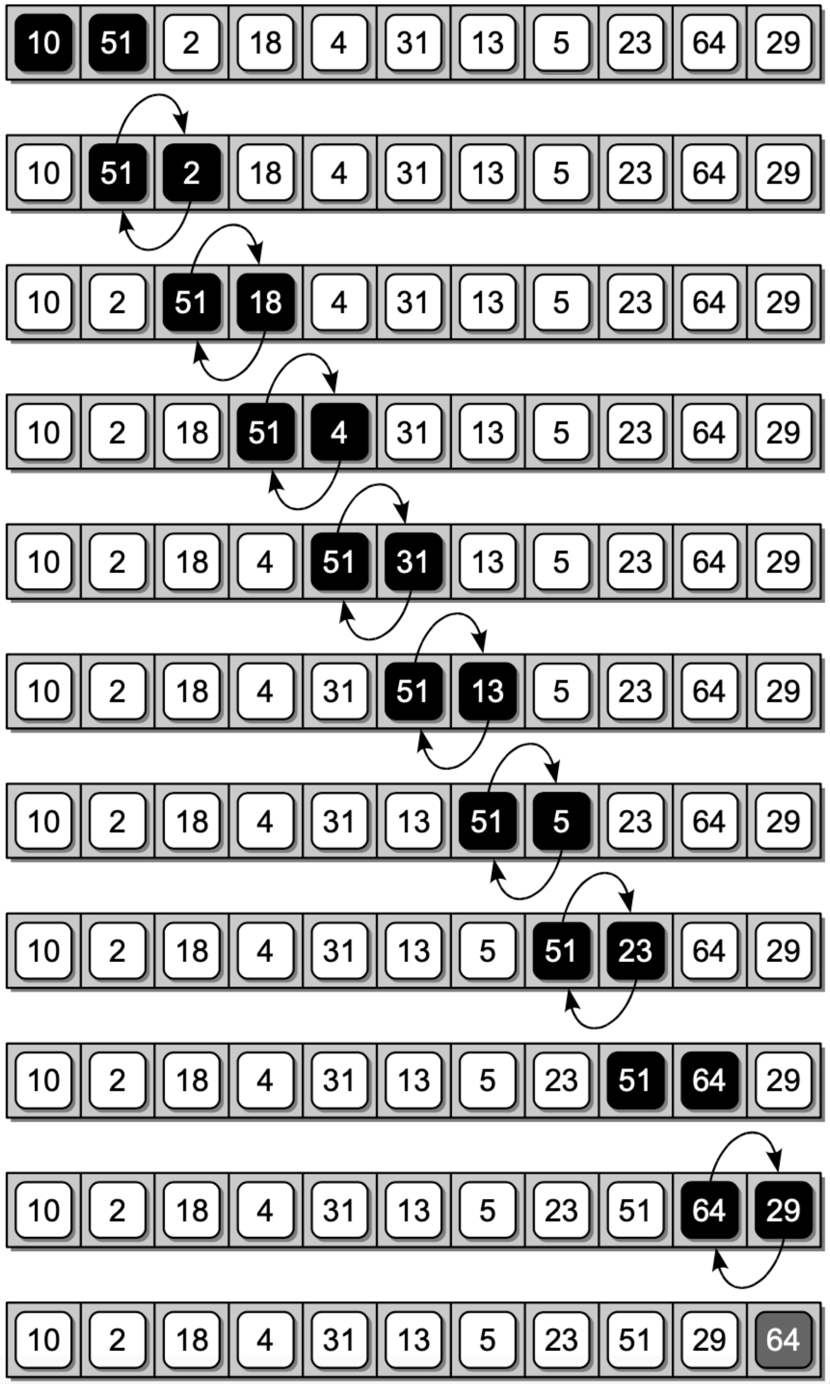


Bubble sort



Bubble sort: example of a sweep

- In this example we perform a first sweep on an unordered sequence using the bubble sort algorithm
- “Heavier” elements sink to the bottom



Bubble sort: outcome of each sweep

- Here we show the status of the sequence after each sweep

- It takes 11 sweeps to sort the sequence

10	2	18	4	31	13	5	23	51	29	64
2	10	4	18	13	5	23	31	29	51	64
2	4	10	13	5	18	23	29	31	51	64
2	4	10	5	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64
2	4	5	10	13	18	23	29	31	51	64

Bubble sort: implementation

```
# Sorts a sequence in ascending order using the bubble sort algorithm.
def bubbleSort(theSeq):
    n = len(theSeq)
    # Perform n-1 bubble operations on the sequence
    for i in range(n - 1):
        # Bubble the largest item to the end.
        for j in range(n - i - 1):
            if theSeq[j] > theSeq[j + 1]:  # swap the j and j+1 items.
                tmp = theSeq[j]
                theSeq[j] = theSeq[j + 1]
                theSeq[j + 1] = tmp
```

Bubble sort: time complexity

- The algorithm complexity is independent of the initial arrangement of the values
- The outer loop always executes n-1 times
- The inner loop executes a variable number of times: first n-1, then n-2, n-3, in general n-i-1
- The total number of iterations for the inner loop will be the sum of the first n-1 integers

- $$\frac{n(n-1)}{2} + n = \frac{n^2}{2} + \frac{n}{2}$$

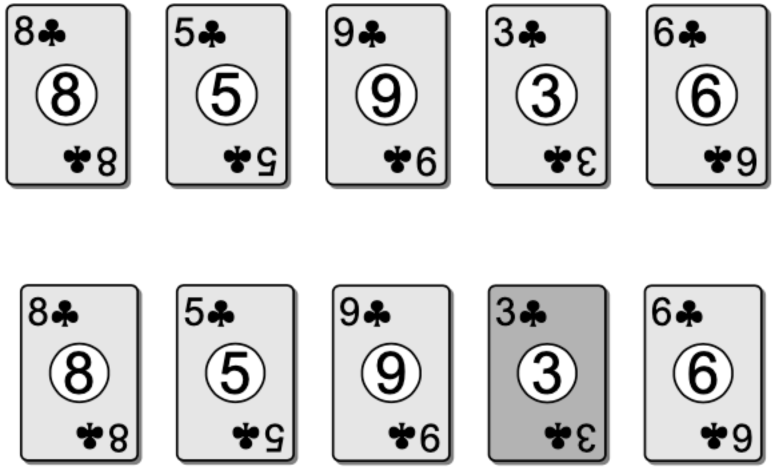
- The time complexity os O(n^2)
- Even if the sequence is already sorted the algorithm would take O(n^2) as it does not know that the sequence is sorted
- Problem: modify the algorithm to get a best case O(n) if the sequence is already sorted

Selection sort

- This method is more efficient than Bubble sort and it works a bit like we do when we sort a hand of cards

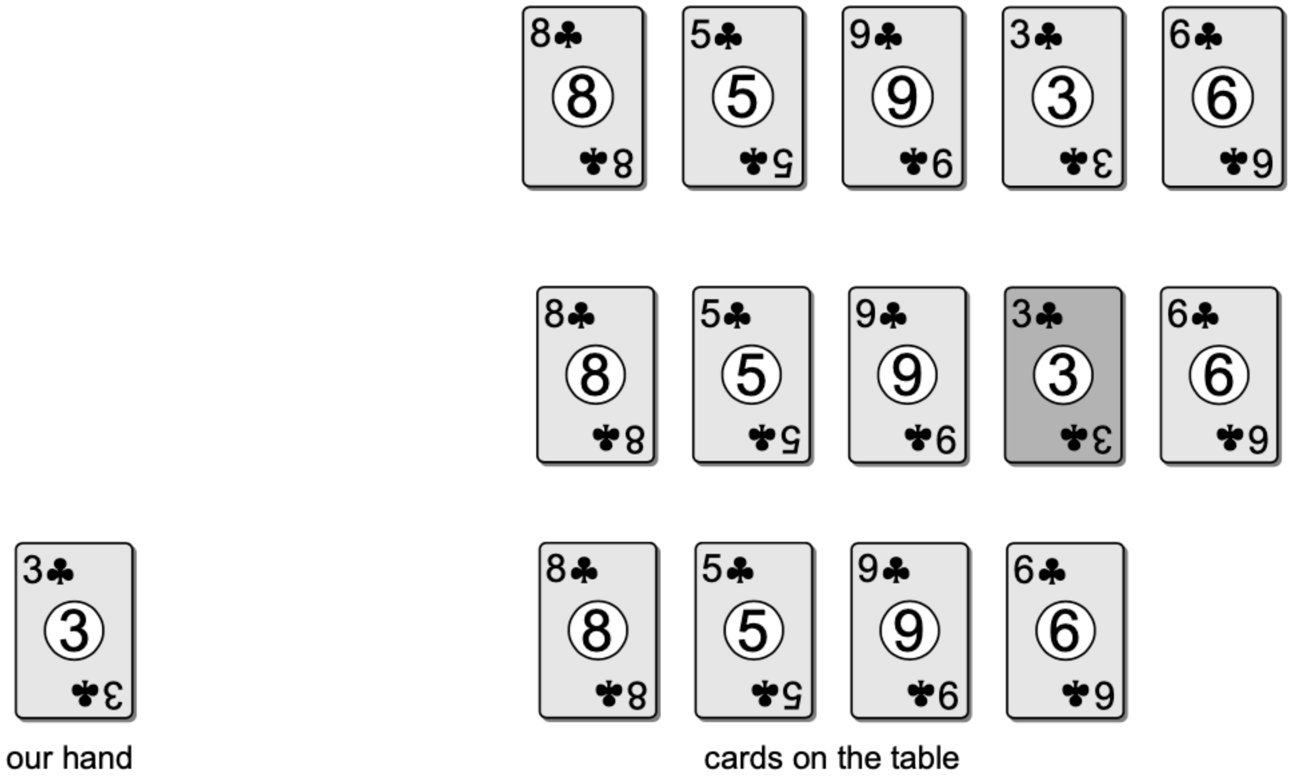
Selection sort

- This method is more efficient than Bubble sort and it works a bit like we do when we sort a hand of cards



Selection sort

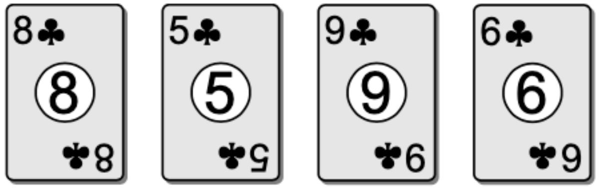
- This method is more efficient than Bubble sort and it works a bit like we do when we sort a hand of cards



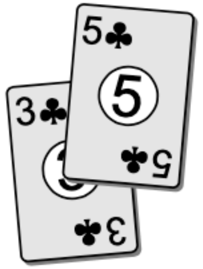
Selection sort

- This method is more efficient than Bubble sort and it works a bit like we do when we sort a hand of cards

our hand



cards on the table

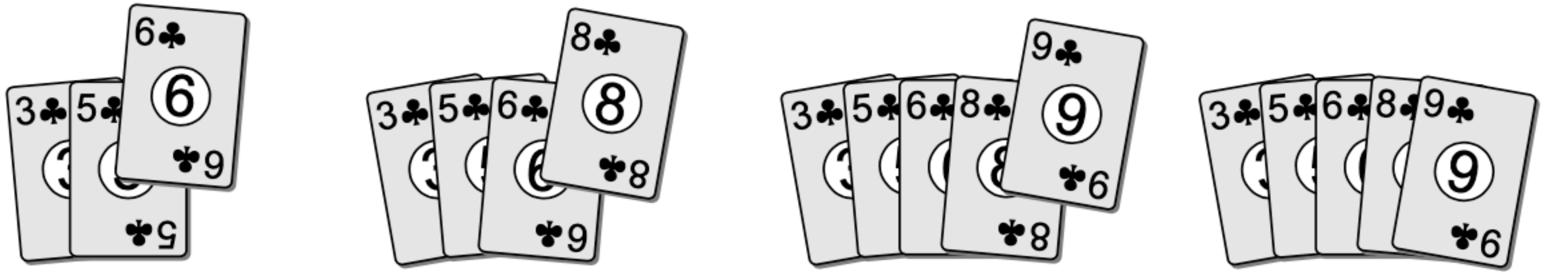


our hand

cards on the table

Selection sort

- This method is more efficient than Bubble sort and it works a bit like we do when we sort a hand of cards



pick up the next smallest card (6)

pick up the next smallest card (8)

pickup the last card (9)

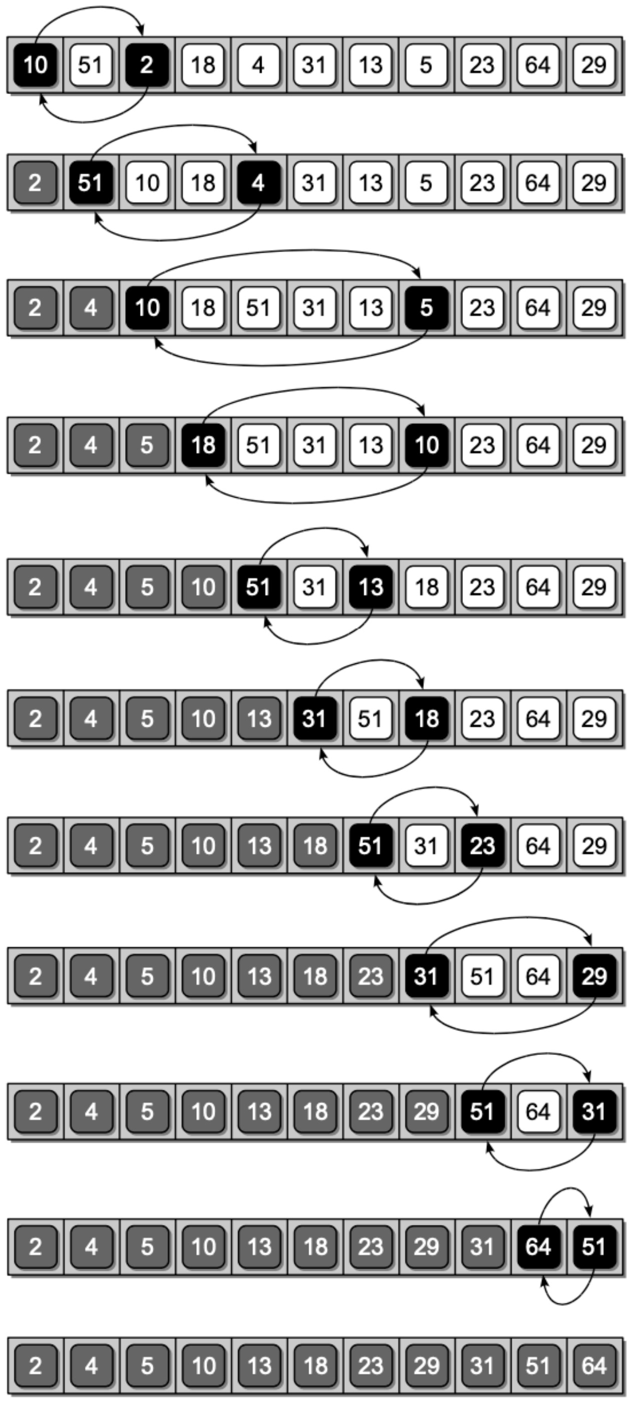
the resulting hand

Implementation

```
# Sorts a sequence in ascending order using the selection sort
algorithm.
def selectionSort( theSeq ):
    n = len( theSeq )
    for i in range( n - 1 ):
        # Assume the ith element is the smallest.
        smallNdx = i
        # Determine if any other element contains a smaller value.
        for j in range( i + 1, n ):
            if theSeq[j] < theSeq[smallNdx] :
                smallNdx = j
        # Swap the ith value and smallNdx value only if the
        # smallest value is not already in proper position.
        if smallNdx != i :
            tmp = theSeq[i]
            theSeq[i] = theSeq[smallNdx]
            theSeq[smallNdx] = tmp
```

Complexity of the selection sort algorithm

- It makes n-1 passes over the array to reposition n-1 values
- The complexity is also $O(n^2)$
- Yet, it makes less swaps that Bubble sort



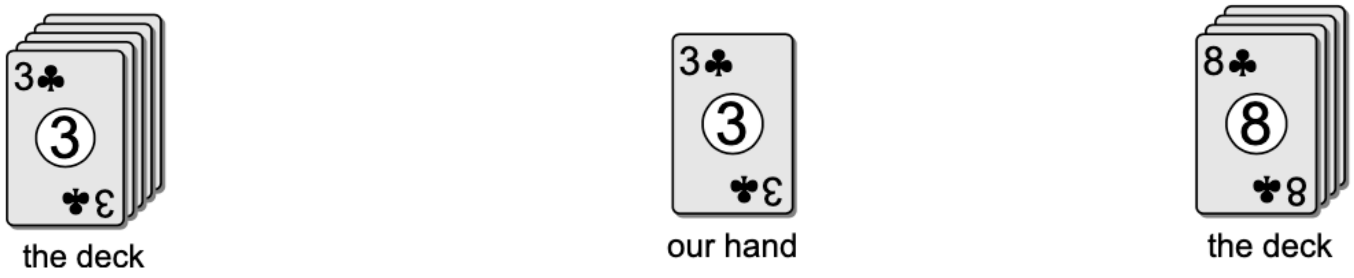
Insertion sort

- It is another common sorting algorithm.
- Let’s consider again the deck of cards analogy

the deck

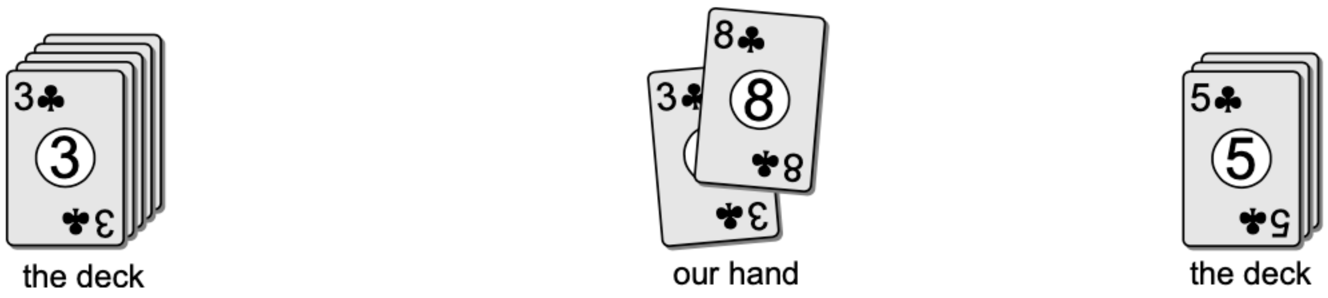
Insertion sort

- It is another common sorting algorithm.
- Let’s consider again the deck of cards analogy



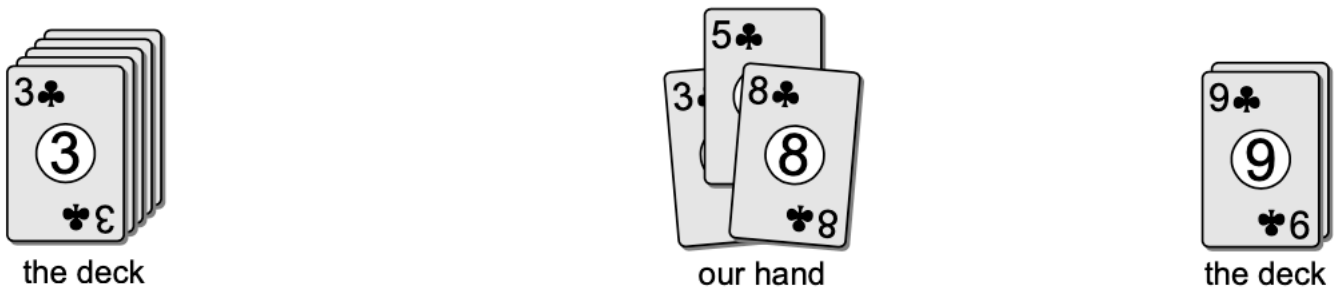
Insertion sort

- It is another common sorting algorithm.
- Let’s consider again the deck of cards analogy



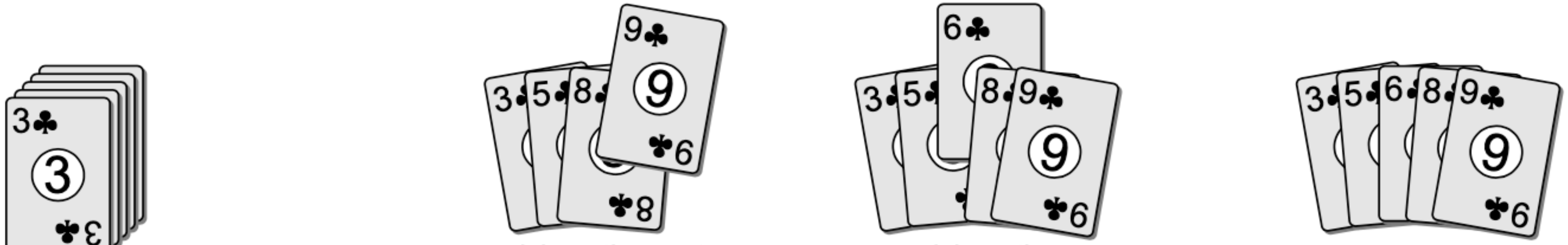
Insertion sort

- It is another common sorting algorithm.
- Let’s consider again the deck of cards analogy



Insertion sort

- It is another common sorting algorithm.
- Let’s consider again the deck of cards analogy



the deck

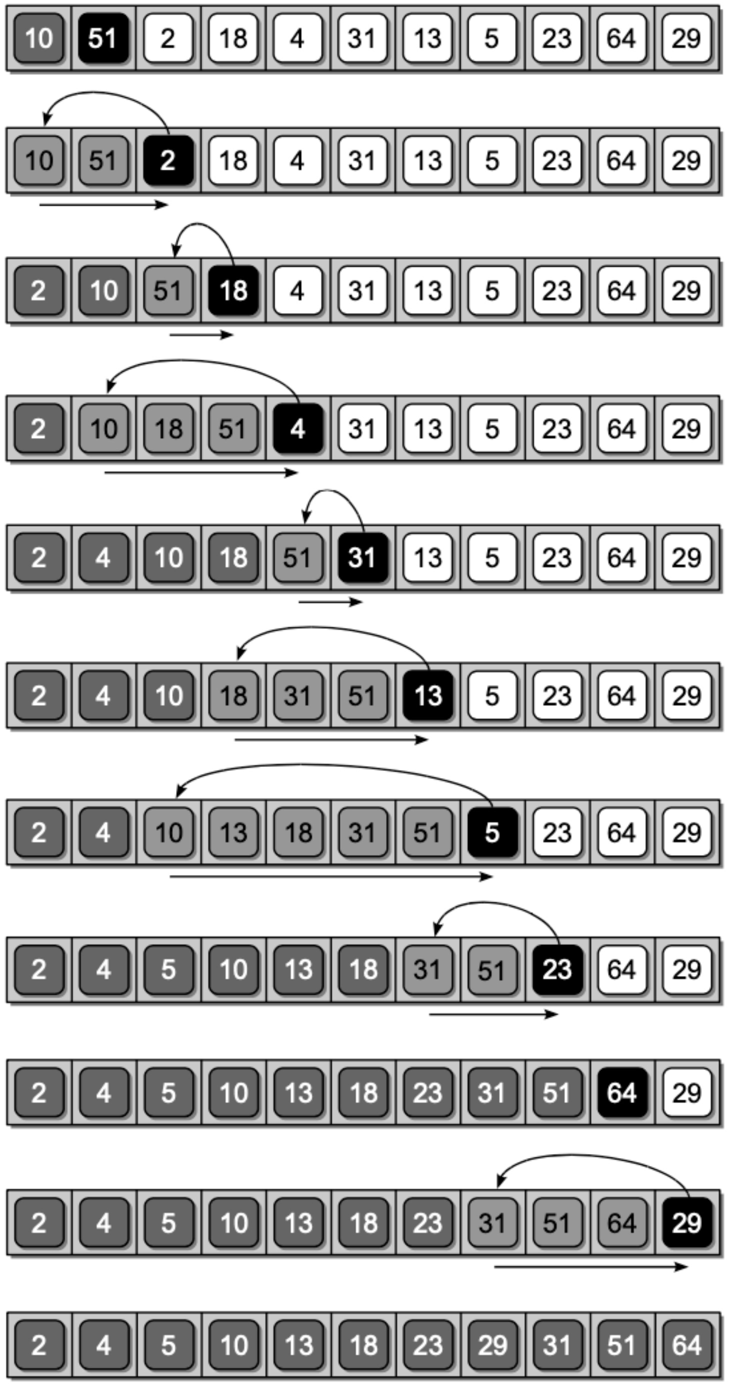
pick up the next card on top (9)

pick up the last card (6)

the resulting hand

Example: applying insertion sort to our sample array

- In the implementation the algorithm maintains both the unsorted and the sorted sequences in the same structure
- At the front the sorted one
- At the back the unsorted one
- To position the next item, the correct spot within the sequence of sorted values is found by performing a search.
- After finding the proper position, the slot has to be opened by shifting the items down one position.



Implementation

```python
def insertionSort( theSeq ):
    n = len( theSeq )
    # Starts with the first item as the only sorted entry.
    for i in range( 1, n ) :
        # Save the value to be positioned.
        value = theSeq[i]
        # Find the position where value fits in the ordered part of the list.
        pos = i
        while pos > 0 and value < theSeq[pos - 1] :
            # Shift the items to the right during the search.
            theSeq[pos] = theSeq[pos - 1]
            pos -= 1
        # Put the saved value into the open slot.
        theSeq[pos] = value
```

Working with sorted lists

Working with sorted list improves efficiency

- If we know that a sequence contains sorted, or partially sorted values, we can use this knowledge to our advantage
- For instance binary search took advantage of the sorted structure of the sequence
- Some sequences are not static, they can grow and shrink during their lifetime (think of the Set ADT)
- Re-sorting the list in the Set ADT just after appending a new item would be very inefficient
- Much better is to add the new element in the right position!

Maintaining a sorted list

- We want to insert the new item in the right position
- First we need to find **where** the item must be placed: use binary search returning the position
  - If the value to be inserted is already present in the list, the algorithm returns the position
  - If it is not already present, we must find the position where to insert it
    - In the binary search algorithm the while loop terminates when either the low or high range variable crosses the other, resulting in the condition low > high.
    - Upon termination of the loop, the low variable will contain the position where the new value should be placed.

Implementation of the enhanced binary search algorithm

```python
# Modified version of the binary search that returns the index within
# a sorted sequence indicating where the target should be located.
def findSortedPosition( theList, target ):
    low = 0
    high = len(theList) - 1
    while low <= high:
        mid = (high + low) // 2
        if theList[mid] == target:
            return mid
        elif target < theList[mid]:
            high = mid - 1
        else:
            # Index of the target.
            low = mid + 1
    return low  # Index where the target value should be.
```

Merging sorted lists

- The following piece of code merges two lists

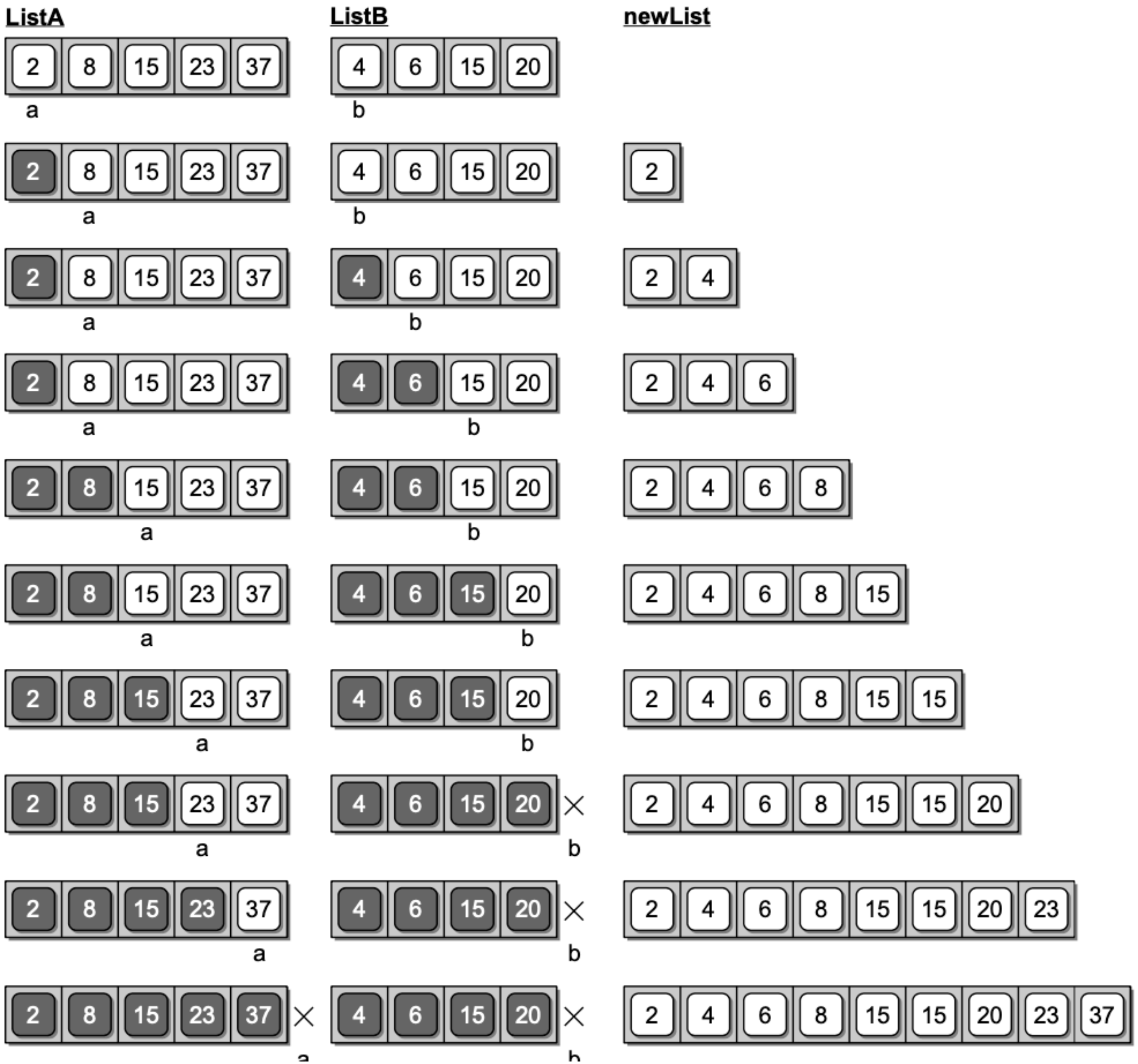
```
listA = [ 2, 8, 15, 23, 37 ]
listB = [ 4, 6, 15, 20 ]
newList = mergeSortedLists( listA, listB )
print( newList )
```

- And the newList should be

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```

Merging sorted lists: solution

- There are two indices scanning the two lists (cursors)
- The items under the cursors are compared
- The samples is moved to the new list and the cursor is advanced



Implementation

```python
# Merges two sorted lists to create and return a new sorted list.
def mergeSortedLists( listA, listB ) :
    # Create the new list and initialize the list markers.
    newList = list()
    a=0
    b=0
    # Merge the two lists together until one is empty.
    while a<len(listA) and b<len(listB):
        if listA[a] < listB[b] :
            newList.append( listA[a] )
            a += 1
        else :
            newList.append( listB[b] )
            b += 1
   # If listA contains more items, append them to newList.
    while a<len(listA):
        newList.append( listA[a] )
        a += 1

   # Or if listB contains more items, append them to newList.
    while b<len(listB):
        newList.append( listB[b] )
        b += 1
    return newList
```

Runtime analysis

- The first loop iterates until all of the values in one of the two original lists have been copied to the third.
  - After the first loop terminates, only one of the next two loops will be executed, depending on which list still contains values.

- The first loop performs the maximum number of iterations when the selection of the next value to be copied alternates between the two lists (maximum: 2n-1 iterations)

- The minimum number of iterations performed by the first loop occurs when all values from one list are copied to the newList and none from the other.
  - If the first loop copies the entire contents of listA to the newList, it will require n iterations followed by n iterations of the third loop to copy the values from listB. If the first loop copies the entire contents of listB to the newList, it will require n iterations followed by n iterations of the second loop to copy the values from listA.

- In total we have 2n iterations maximum and the algorithm is O(n)

The Set ADT revisited

A Set ADT implementation based on a sorted sequence

- Using a sorted sequence allows to save time:
  - looking for an item
  - adding an item
  - Making operations on sets (intersection, union, difference)

Implementation

```
# Determines if an element is in the set.
def __contains__(self, element):
    ndx = self._findPosition(element)
    return ndx < len(self) and
self._theElements[ndx] == element
```

Implementation

```
# Adds a new unique element to the set.
def add(self, element):
    if element not in self:
        ndx = self._findPosition(element)
        self._theElements.insert(ndx, element)
```

```
# Removes an element from the set.
def remove(self, element):
    assert element in self, "The element must be in the set."
    ndx = self._findPosition(element)
    self._theElements.pop(ndx)
```

Implementation

- Find positions performs a binary search, which has a complexity of $O(\log_2(n))$

```python
# Finds the position of the element within the ordered list.
def _findPosition(self, element):
    low = 0
    high = len(self._theElements) - 1
    while low <= high:
        mid = (high + low) // 2
        if self._theElements[mid] == element:
            return mid
        elif element < self._theElements[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return low
```

Implementation: the new __eq__() operator

- The new == operator has a complexity O(n) while the previous equal operator had a complexity $O(n^2)$ since it used the isSubset() operator

```
# verifies if two sets are equal
def __eq__(self, setB):
    if len(self) != len(setB):
        return False
    else:
        for i in range(len(self)):
            if self._theElements[i] != setB._theElements[i]:
                return False
        return True
```

Complexity of the old __eq__ operator

- It is based on `isSubsetOf`() which uses the `in` operator (O(log n) if the set is ordered) it's called n times, therefore the time-complexity would be O(n log n).

```
# Determines if two sets are equal.
def __eq__(self, setB):
    if len(self) != len(setB):
        return False
    else:
        return self.isSubsetOf(setB)
```

```
# Determines if this set is a subset of setB.
def isSubsetOf(self, setB):
    for element in self:
        if element not in setB:
            return False
    return True
```

Implementation: the new union() operator

```python
# Creates a new set from this set and setB.
def union(self, setB):
    newSet = Set()
    a = 0
    b = 0
    # Merge the two lists together until one is empty.
    while a < len(self) and b < len(setB):
        valueA = self._theElements[a]
        valueB = setB._theElements[b]
        if valueA < valueB:
            newSet._theElements.append(valueA)
            a += 1
        elif valueA > valueB:
            newSet._theElements.append(valueB)
            b += 1
```

```python
        else:
        # Only one of the two duplicates are appended.
            newSet._theElements.append(valueA)
            a += 1
            b += 1
        # If listA contains more items,
         #append them to newList.
    while a < len(self):
        newSet._theElements.append(self._theElements[a])
        a += 1
    # Or if listB contains more, append them to newList.
    while b < len(setB):
        newSet._theElements.append(setB._theElements[b])
        b += 1
    return newSet
```

Comparing the operations

Operation	Unordered	Ordered
constructor	O(1)	O(1)
`len(s)`	O(1)	O(1)
`x in s`	O(n)	O(log n)
`s.add(x)`	O(n)	O(n)
`s.isSubsetOf(t)`	$O(n^2)$	O(n)
`s==t`	$O(n^2)$	O(n)
`s.union(t)`	$O(n^2)$	O(n)