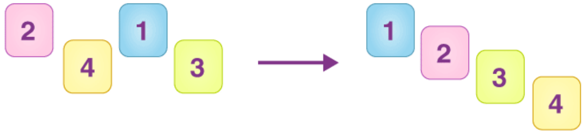
The National School of Artificial Intelligence
المدرسة الوطنية العليا للذكاء الاصطناعي



Data Structure & Algorithms 1

CHAPTER 4
STATIC DATA STRUCTURE (PART4): SORTING ALGORITHMS

Sep – Dec 2023

Outline

- **Sorting Algorithms**
  - Definition
  - Different sorting algorithms
  - Selection Sort
    - Analysis
    - Modular decomposition
    - Application
    - Complexity

Introduction

The goal of a sorting operation is to **arrange** elements in an array according to a specific order or criterion, either in ascending or descending order.

There are two types of sorting:

- **Internal sorting**, where all elements are in main memory (RAM).
- **External sorting**, where only a portion of the elements is in main memory, while the others are in secondary memory (Hard Disk, SSD).

Introduction

There are several sorting algorithms:

1. **Odd-Even Transposition Sort**

   Principle: Two adjacent elements are compared and swapped if the second element is smaller than the first. In this case, a backward check is performed to ensure that the order has not been modified, and if so, it is restored.

   It consists of 2 phases – the odd phase and even phase:

   - **Odd phase:** Every odd indexed element is compared with the next even indexed element(considering 1-based indexing).
   - **Even phase:** Every even indexed element is compared with the next odd indexed element.

Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

Step 1(odd):	2	1	4	9	5	3	6	10
Step 2(even):	1	2	4	9	3	5	6	10
Step 3(odd):	1	2	4	3	9	5	6	10
Step 4(even):	1	2	3	4	5	9	6	10
Step 5(odd):	1	2	3	4	5	6	9	10
Step 6(even):	1	2	3	4	5	6	9	10
Step 7(odd):	1	2	3	4	5	6	9	10
Step 8(even):	1	2	3	4	5	6	9	10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Introduction

1. Bubble Sort

Principle: Badly sorted elements move up in the array like bubbles rising to the surface of a liquid. The swapping of elements occurs with permutations if elem1 > elem2. It is evident that multiple passes over all elements are necessary. This method is less efficient.

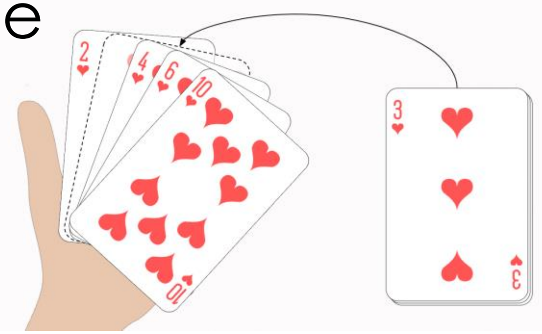
- Compare consecutive pairs of elements
- Swap elements in pair such that smaller is first
- When reach end of list, start over again
- Stop when no more swap have been made
- Largest unsorted element always at end after pass, so at most n passes

Introduction

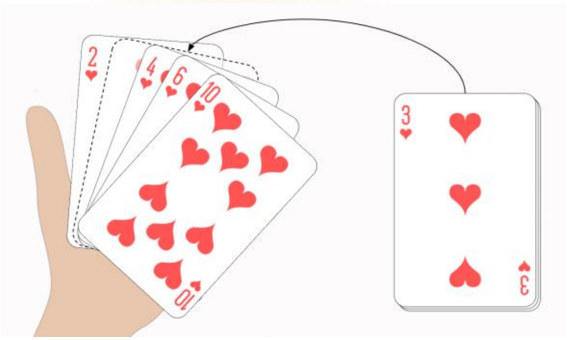
3. Insertion Sort

Principle: This algorithm involves iterating through the values of the array, one by one, and inserting them at the right place in the sorted array consisting of previously inserted and sorted values.

The values are inserted in the order in which they appear in the array. The challenge with this algorithm is that it requires traversing the sorted array to determine where to insert the new element, and then shifting all values greater than the element to be inserted by one position.



Introduction

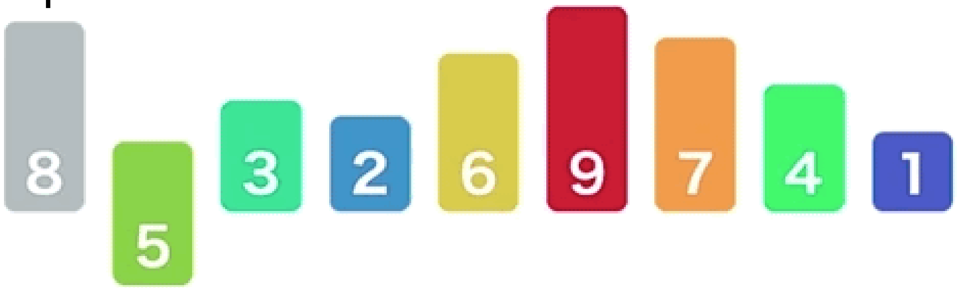


3. Insertion Sort

1. The list is divided into two parts: sorted and unsorted

2. In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.

3. A list of *n* elements will take at most *n-1* passes to sort the data



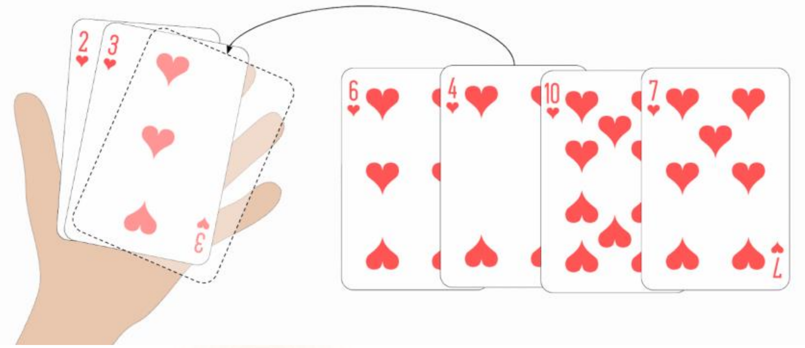
Introduction

4. Selection Sort (Successive Minimum Sort)

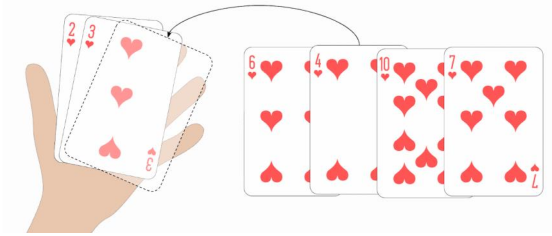
sorts an array of N integers in ascending order.

Principle:

The smallest element of the array is swapped with the first element of the array. Then, the smallest element of the remaining array is swapped with the second element, and so on.

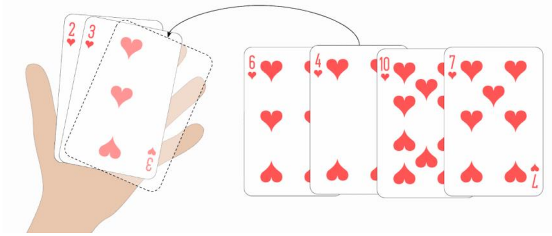


Selection Sort (Analysis)



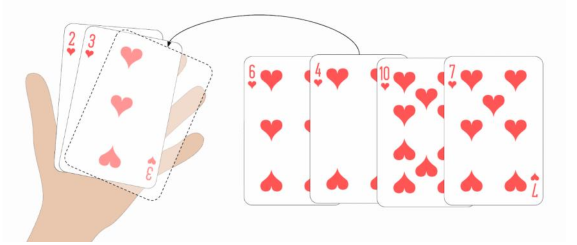
1. First step
   - Extract minimum element
   - Swap it with element at index 0
2. Subsequent step
   - In remaining sublist, extract minimum element
   - Swap it with the element at index 1
3. Keep the left portion of the list sorted
   - At i'th step, first i elements in list are sorted
   - All other elements are bigger than first i elements

Selection Sort (Analysis)



- The principle of selection sort is to exchange, at each iteration `i`, ranging from `0 to Size-1`, the integer `A[i]` with the smallest integer in the part of the array ranging from index `i to Size-1` (size of the array).
- **Modular Breakdown:** We will build a module, a procedure that performs the selection sort (SELSORT), which will require the following modules:
  - A module (MinInd) that gives us the index of the minimum element in a portion of the array.
  - A module (SWAP) that swaps two elements in the array.
  - We will also need modules to read the initial array (READ1D) and display the sorted array (WRITE1D).

Selection Sort (Modules)



PROCEDURE

Integer A

Integer B

SWAP

Var Integer A

Var Integer B

Swap the content of A and B

FUNCTION

Integer A[]

Integer B1

Integer B2

MinInd

Integer

Return the index of the smallest element of a portion (B1, B2) of 1D array A

PROCEDURE

Integer A[]

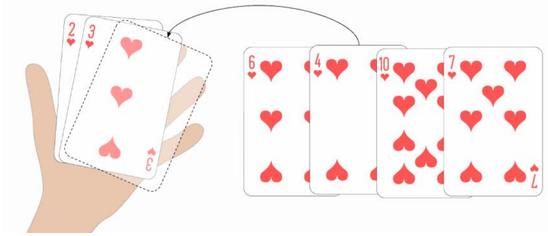
Integer Size

SELSORT

Var Integer A[]

Sort the element of A in ascending order using the selection method

Selection Sort (Modules)

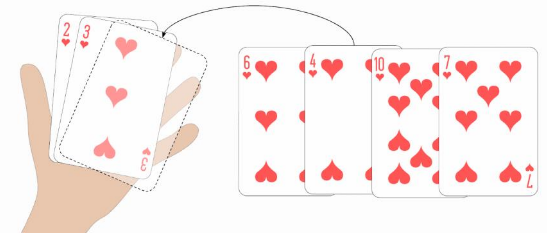


Module MinInd
Analysis:

- We assume that `B1 < B2 < Size`.
- Save `B1` in `Ind`.
- Traverse the array by varying the index: `i = B1+1`, B1+2, …, B2. At each iteration:
    - IF the element `A[i]` is smaller than `A[Ind]` (a new minimum is found), THEN
        - Update the index: `Ind = i`.

```
Integer Function MinInd (Integer A[], Integer B1,
Integer B2)
Variable Integer Ind, i
BEGIN
    Ind = B1
    FOR i FROM B1+1 TO B2 DO
        IF A[i] < A[Ind] THEN
            Ind = i
        END IF
    END FOR
    MinInd = Ind
END
```

Selection Sort (Modules)

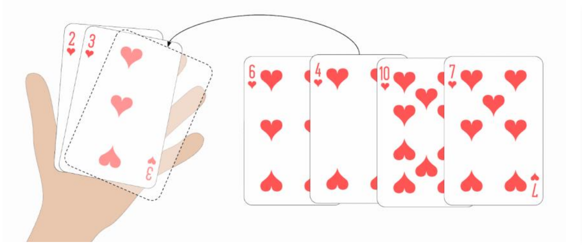


Module SELSORT
Analysis:

- Traverse the array by varying i: 0, 2, 3, …, Size-1. At each iteration:
- Search for the index of the smallest element between `i` and `Size-1`: (`MinInd(A, i, Size-1)`).
- Swap this element with `A[i]`.

```
Procedure SELSORT (Var Integer A[], Integer Size,)
Variable Integer Ind, I
Procedure SWAP
….Procedure Body…
Function MinInd
….Function Body…
BEGIN
    FOR i FROM 0 TO Size-1 DO
        Ind = MinInd(A, i, Size-1)
        SAWP (A[i], A[Ind])
    END FOR
END
```

Selection Sort (Modules)

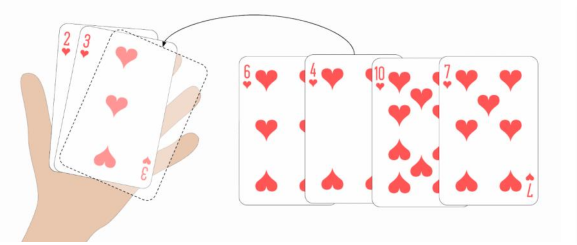


Construct the main algorithm
Analysis:

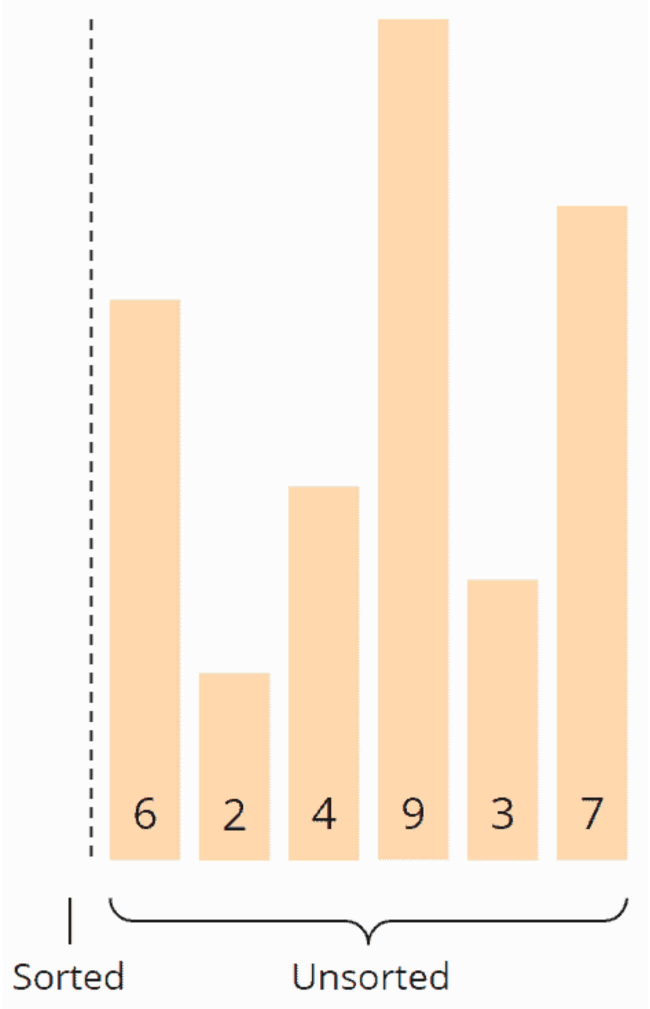
- Read the array A
- Sort the array A
- Display the array A

```
Algorithm Selection_Sort
Constant MAX = 100
Variable Integer A[MAX], Size
Procedures SWAP, READ1D, WRITE1D, SELSORT
   ….Procedures Body…
BEGIN
   READ1D(A, Size)
   SELSORT(A, Size)
   WRITE1D(A, Size)
END
```

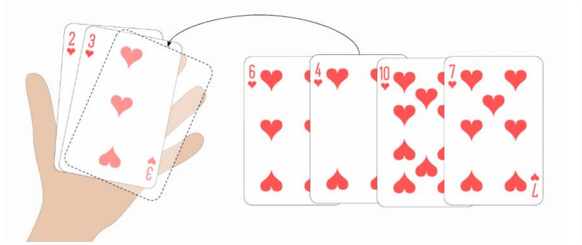
Selection Sort (example)



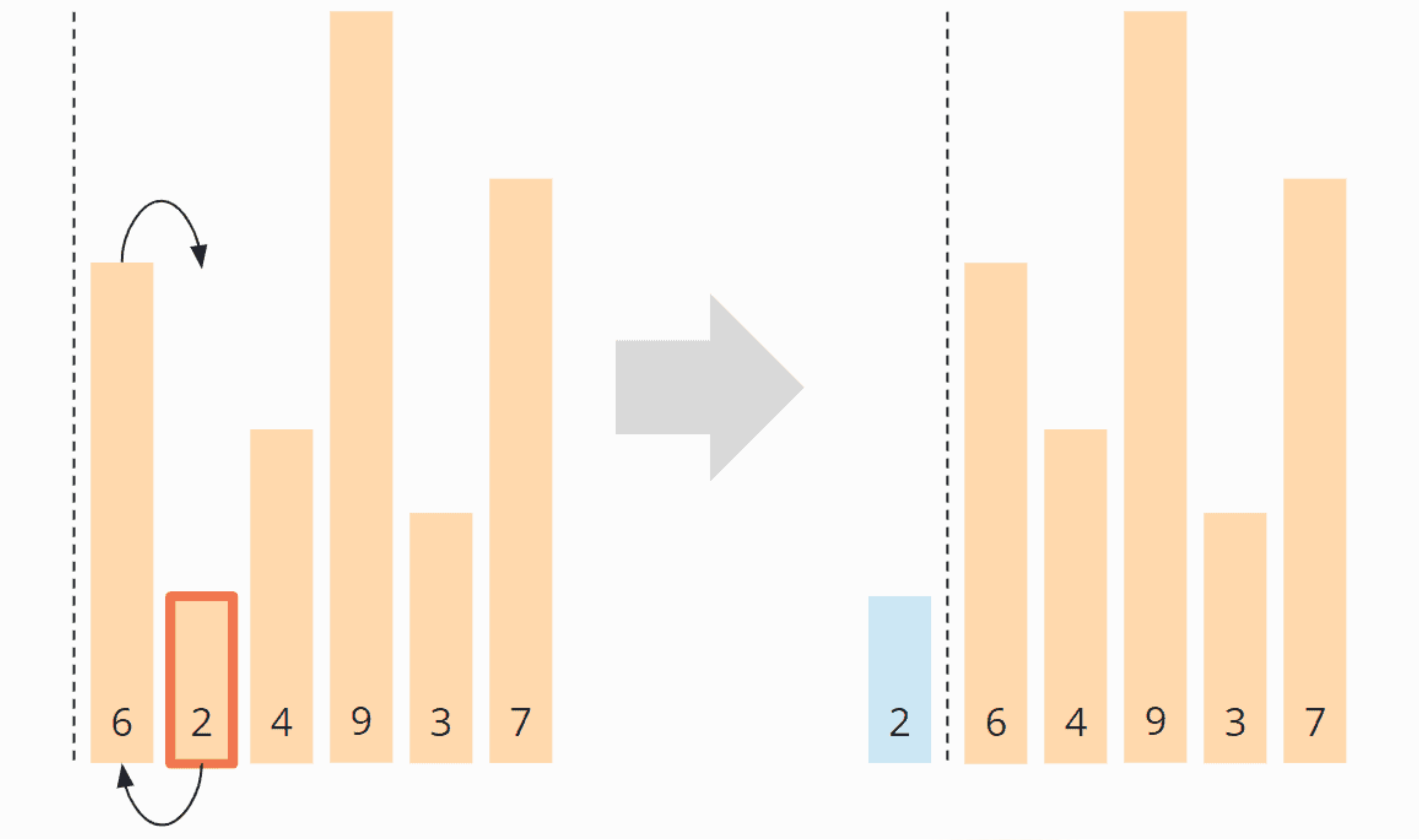
We divide the array into a left, sorted part and a right, unsorted part. The sorted part is empty at the beginning:



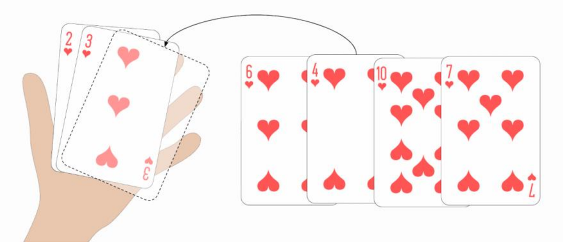
Selection Sort (example)



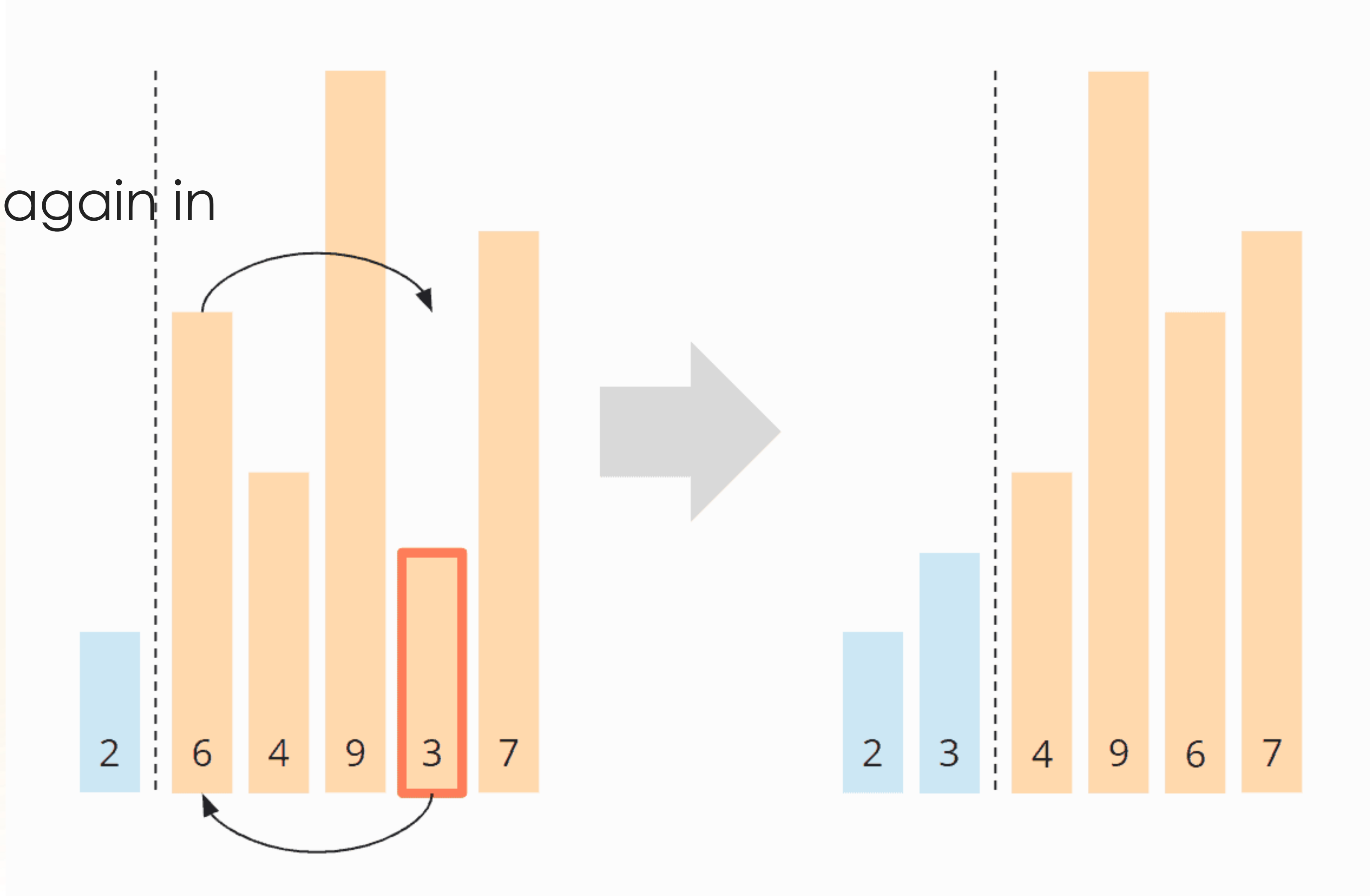
search for the smallest element in the right, unsorted part.



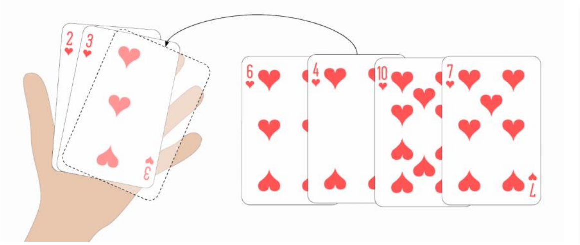
Selection Sort (example)



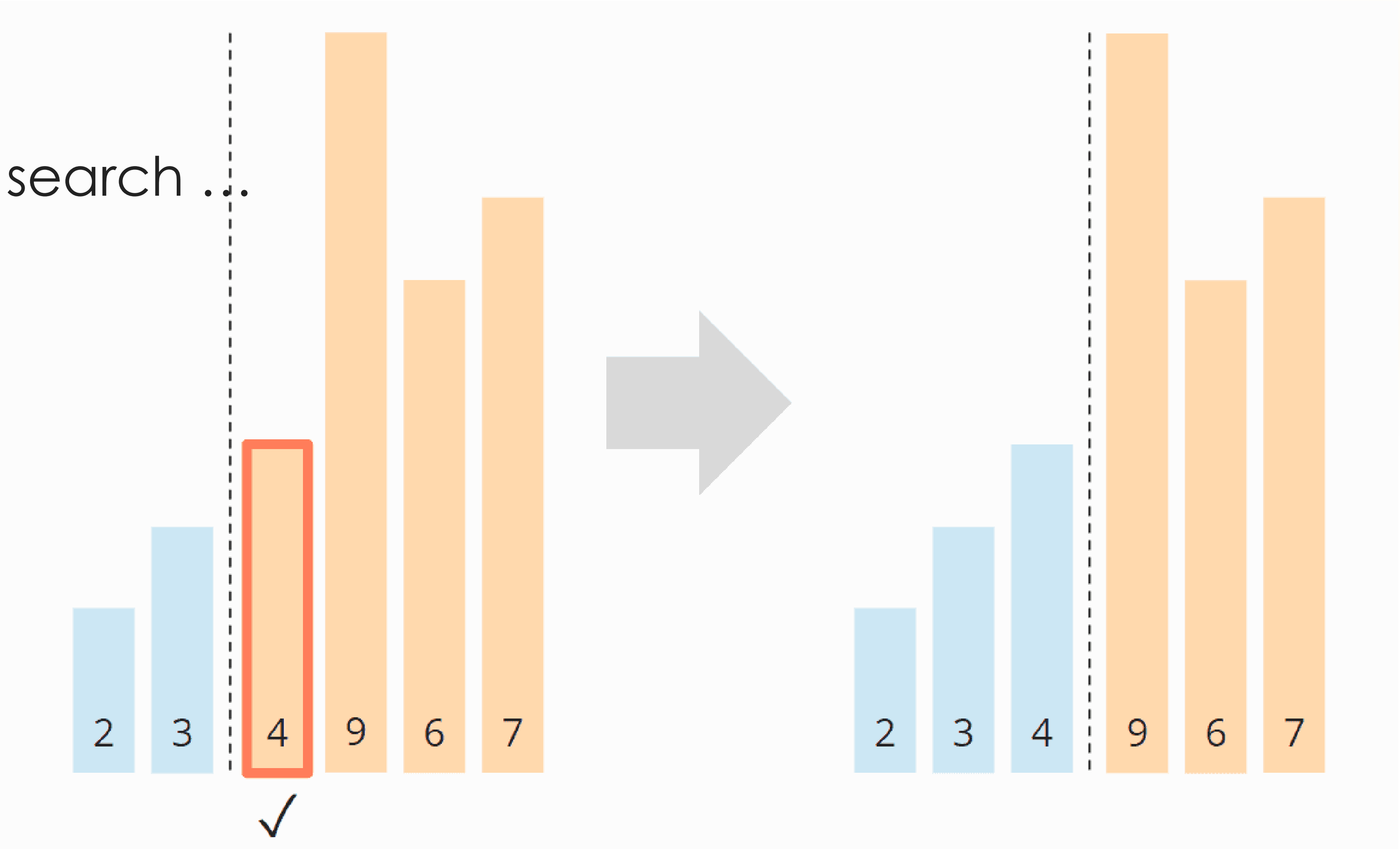
We search again in the right…



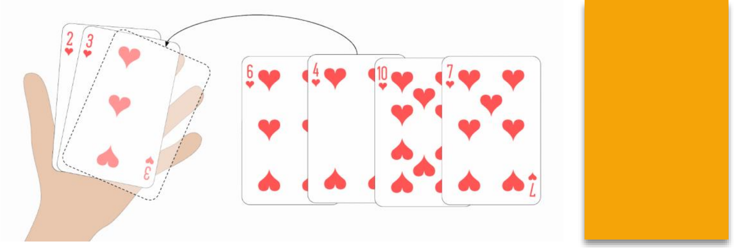
Selection Sort (example)



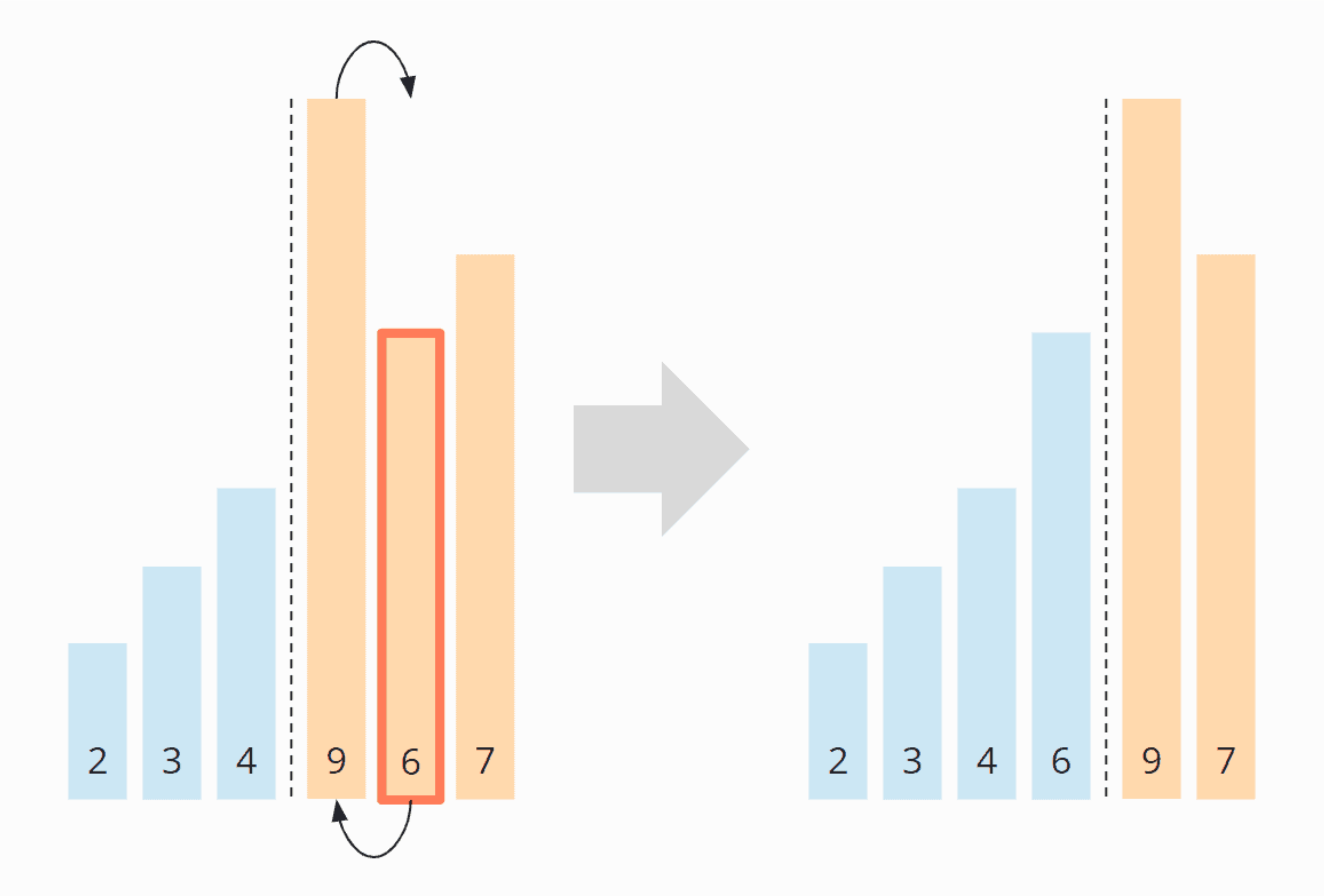
Again, we search …



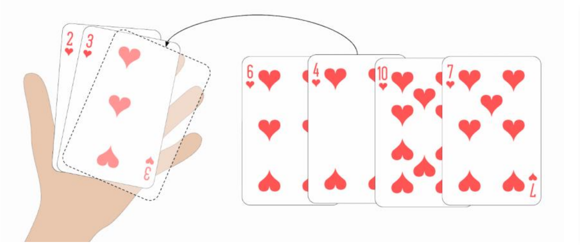
Selection Sort (example)



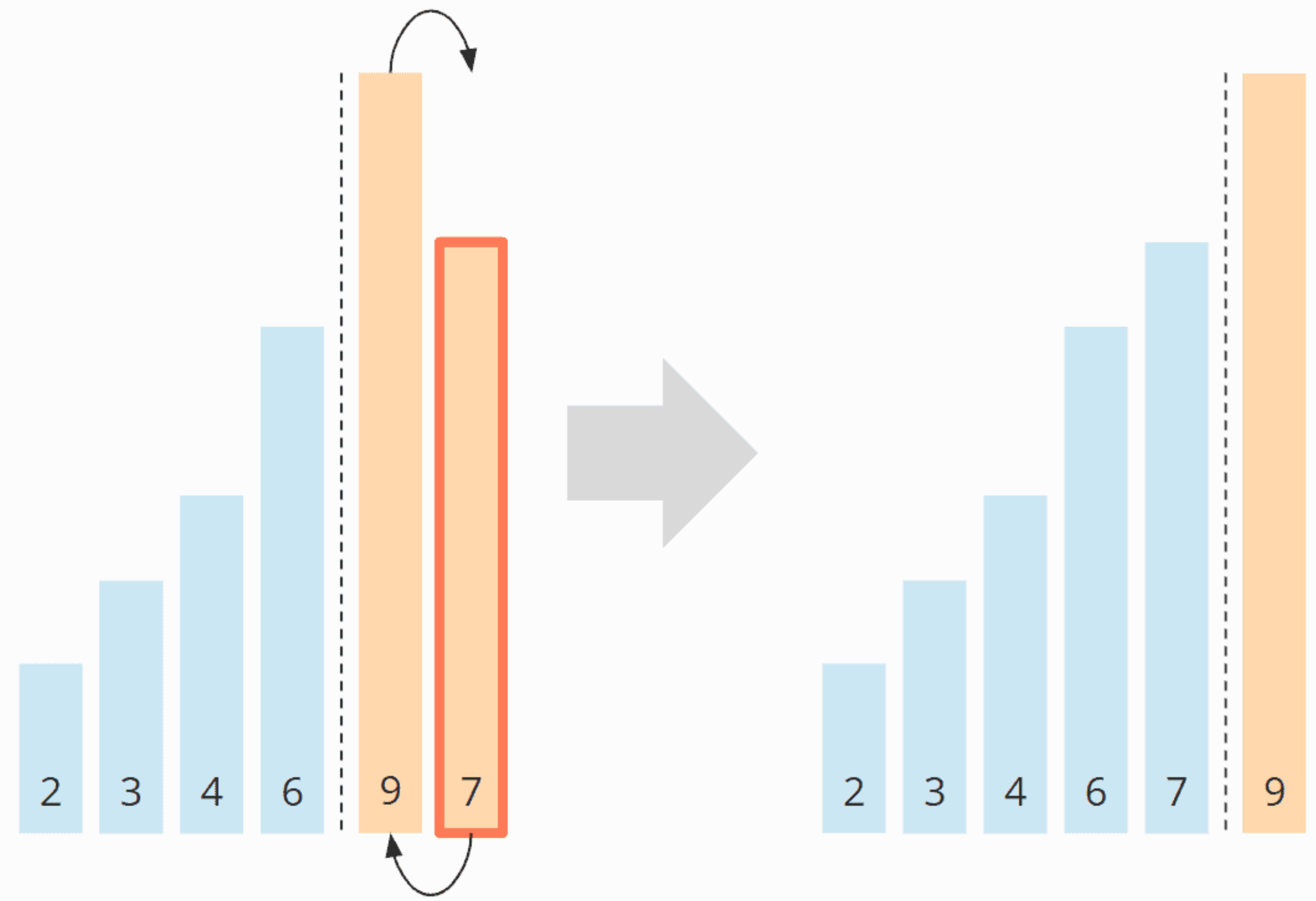
Again …



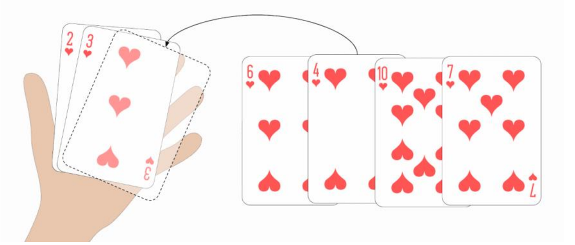
Selection Sort (example)

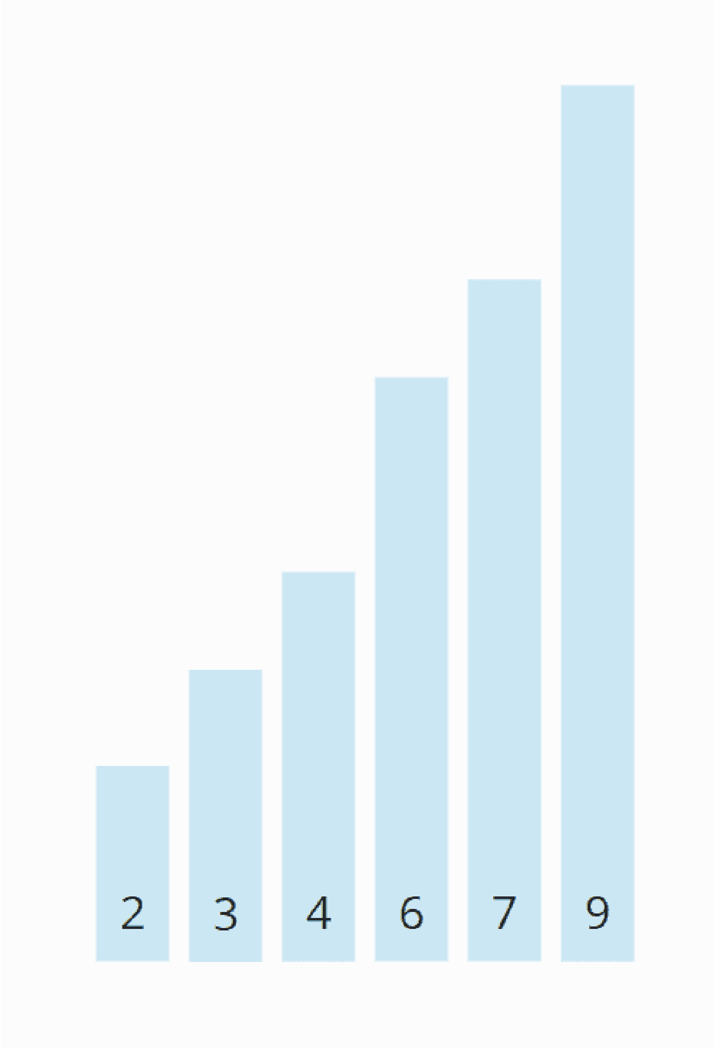


Again …

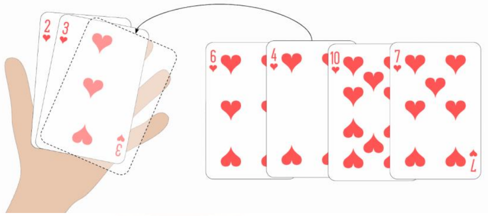


Selection Sort (example)





Selection Sort (Application)



Provide the solution that allows searching by dichotomy for a given value V in a sorted array of up to 100 integers.

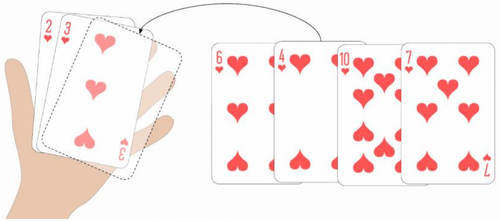
Modular Breakdown:

- We need a module SearchDichV
- Basic modules
  - READ1D and WRITE1D

Analysis of Module `SearchDichV`

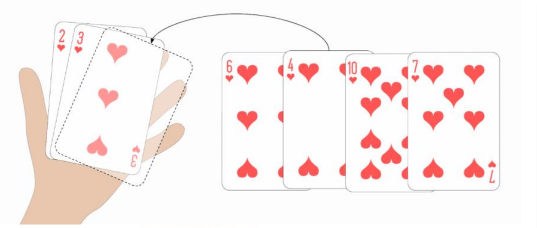
- We assume having a sorted array
- Initialization of a variable `Found` to false;
- Initialization of two variables `Min` to `0` and `Max` to `Size-1`;
- `While (Min < Max)` we perform the following:
  - We assign to a variable `mid` the value `(Min + Max) DIV 2`
  - We compare the element `A[mid]` with the value `V`
    - If `A[mid] < V`, we set another minimum, which is "`mid + 1`"
    - Otherwise, we set another maximum, which is "`mid`"
    - We compare `if A[mid]` is equal to `V`
      - If it is equal, we assign `True` to `Found`
- We assign: `RechDichV = Found`

Selection Sort (Application)



```
Boolean FUNCTION SearchDichV (Integer A[], Integer Size, Integer V)
Variable Integer Min, Max, mid
         Boolean Found
BEGIN
    Found = False
    Min = 0
    Max = Size -1
    WHILE Min < Max AND NOT Found DO
        mid = (Min + Max) DIV 2
        IF A[mid] < V THEN
            Min = mid + 1
        ELSE
            Max = mid
        END IF
        IF A[mid] == V THEN
            Found  = True
        END IF
    END WHILE
    SearchDichV = Found
END
```

Selection Sort (Application)

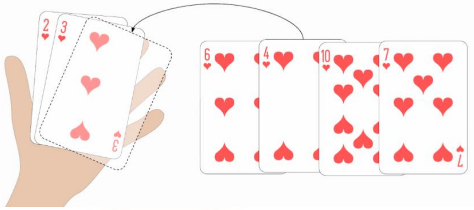


Construct the main algorithm
Analysis:

- Read the array A
- Read the searched value V
- Call the Function **SearchDichV**
- Display the result (V found or not)

```
Algorithm Selection_Sort
Constant MAX = 100
Variable Integer A[MAX], Size, V
         Boolean Result
Procedures READ1D, WRITE1D, SearchDichV
   ….Procedures Body…
BEGIN
   READ1D(A, Size)
   WRITE (‘The searched value is: ’)
   READ (V)
   Result = SearchDichV(A, Size, V)
   IF Result == True THEN
      WRITE (V, “ exist in the array A”)
   ELSE
      WRITE (V, “ does not exist in the array A”)
   END IF
END
```

Complexity of Selection Sort

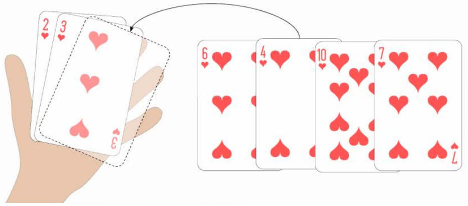


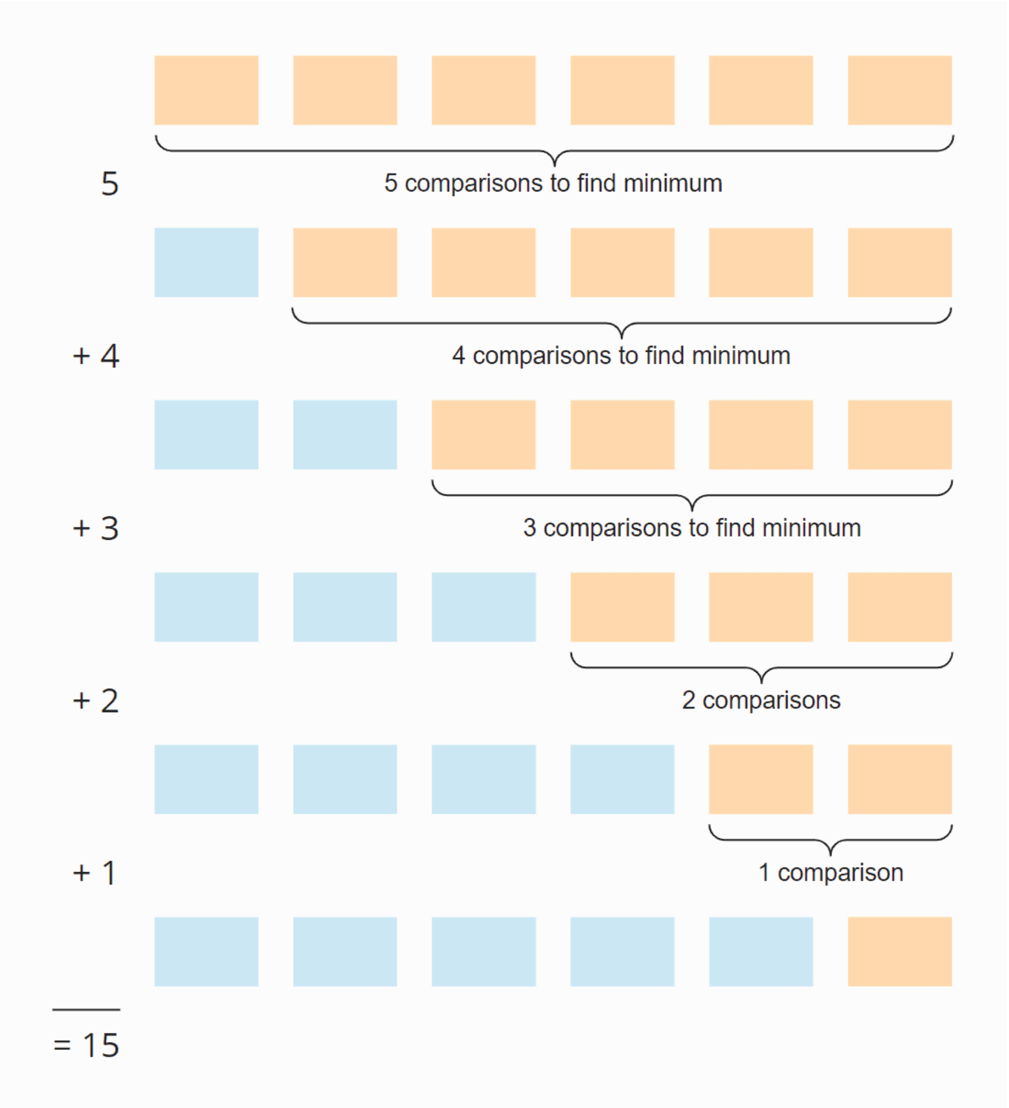
To assess the efficiency and speed of a sorting algorithm, the concept of **complexity** is used, which provides an approximate time for the algorithm expressed in the number of operations performed.

It is an order of magnitude expressed in terms of the amount of data to be processed.

Sorting Algorithms	Time Complexity		
	Best Case	Average Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)

Complexity of Selection Sort





$6 \times 5 \times ½ \quad = \quad 30 \times ½ \quad = \quad 15$

If we replace 6 with *n*, we get

$n \times (n - 1) \times ½$

When multiplied, that's:

$\boldsymbol{½ n^2 - ½ n}$

Complexity:

$O(n^2)$ –"quadratic time"

The average, best-case, and worst-case time complexity of Selection Sort is: *$O(n^2)$*

Selection Sort (Conclusion)