PSEUDOCODE

© Dr Markus Lumpe, 2024

AN INFORMAL WAY OF PROGRAMMING

- Algorithms are generally described in English and in pseudocode designed to be readable by anyone who has done a little programming.
- Understanding and using pseudocode is an important programming skill. No matter what background you have (e.g., Ruby, C, or Python), pseudocode is an invaluable engineering tool when designing and translating an algorithm to a programming language and system of choice.
- Pseudocode entails a mix of conventions of programming languages (e.g., assignment statements, conditionals statements, and loop statements) and an informal, but generally self-explanatory notation of actions and conditions. Pseudocode shares many feature found in actual programming languages, but it is primarily intended for human reading and understanding.

© Dr Markus Lumpe, 2024

NOTES ON PSEUDOCODE CONVENTIONS

- Indentation indicates block structure. Using indentation rather than indicators, such as **begin** and **end** or { and }, reduces clutter while preserving clarity. In addition, optional line numbers allow for cross referencing pseudocode.
- The looping constructs **while**, **for**, **repeat-until** and the **if-else** conditional construct have interpretations similar to those in C/C++, C#, Java, Python and Pascal. We use the keyword **to** when a for loop increments in each iteration, and we use the keyword **downto** when a for loop decrements its loop counter. We use the optional **by** expression when the loop counter changes by an amount greater than 1. We may also use the keywords **break** and **continue** to exit a loop immediately and start the next iteration, respectively.
- The symbol **//** indicates a comment extending to the end of the current line.
- Variables are local to the current procedure unless indicated otherwise.
- Array access is indicated using square brackets. The notation '**..**' is used to indicate a range of values within an array.
- We typically organize compound data into record-like *objects*, which are composed of *attributes*. We use the record selection operator '**.**' for attribute access.
- Parameters are passed *by-value*. The called procedure receives its own copy of the parameter. When objects are passed, the pointer to the data representing the object is copied, not the attributes themselves. Arrays are objects, hence we pass the pointer to an array to a procedure.
- A **return** statement (with or without arguments) immediately transfers control back to the point of call in the calling procedure.
- We use '**:=**' for assignment and '**:=:**' for exchange of values. For comparison, we employ the standard operators like '**=**', '**!=**', '**and**', and '**or**'.

© Dr Markus Lumpe, 2024

SENDING A SEQUENCE 1..10 TO OUTPUT

```
1  i := 1
2  num := 10
3  while i <= num
4      output i
5      i := i + 1
```

- Line 1 initializes a variable called i with 1. This is an initialized variable declaration.
- Line 2 initializes variable num with 10. This variable defines the end of the sequence.
- Line 3 defines a while loop. The condition states that the body of the loop has to be executed as long as the value of i does not exceed the value of num.
- Lines 4 and 5 define the loop body. Line 4 send the current value of i to the output. An implementation has to map this statement to a corresponding I/O feature in the programming language of choice.
- Line 5 increments variable i by 1. This is an assignment statement. In an assignment the left-hand side (i.e., variable i) must have been previously declared (see line 1) in order to be a valid target.

© Dr Markus Lumpe, 2024

SENDING A SEQUENCE 1..10 TO OUTPUT

```
1  num := 10
2  for i := 1 to num
3      output i
4      i := i + 1
```

- We can design the algorithm also by using a for-statement:
    - Line 1 initializes variable num with 10. This variable defines the end of the sequence.
    - Line 2 defines a for loop. A for loop defines loop variable i, initialized to 1. The body body of the loop has to be executed as long as the value of i does not exceed the value of num. The loop variable is incremented by 1 in every iteration.
    - Lines 3 and 4 define the loop body as before.

© Dr Markus Lumpe, 2024

SUM OF AN ARRAY

```
1  let A be an array of numbers
2  sum := 0
3  for i := 0 to A.length - 1
4      sum := sum + A[i]
5  output sum
```

- Line 1 declares an array of numbers. The number of elements and their respective value are unimportant.
- Line 2 initializes the sum to 1.
- Line 3 defines a for loop. A for loop defines loop variable i, initialized to 0. We assume 0-based arrays. The body body of the loop has to be executed as long as the value of i does not exceed the value of A.length - 1. The expression A.length denotes the number of elements in A.
- Lines 4 sums the array elements.
- Line 5 send the sum to output.

© Dr Markus Lumpe, 2024

SUM OF AN ARRAY AS FUNCTION

```
    Sum(A)
2       sum := 0
3       **for** i := 0 **to** A.length - 1
4           sum := sum + A[i]
5       **return** sum
```

- The preceding algorithm can be expressed as a function Sum which takes an array as argument. Recall, we pass arrays as pointers.

- In line 5, the return statement transfers control back to the caller using sum as return value.

© Dr Markus Lumpe, 2024

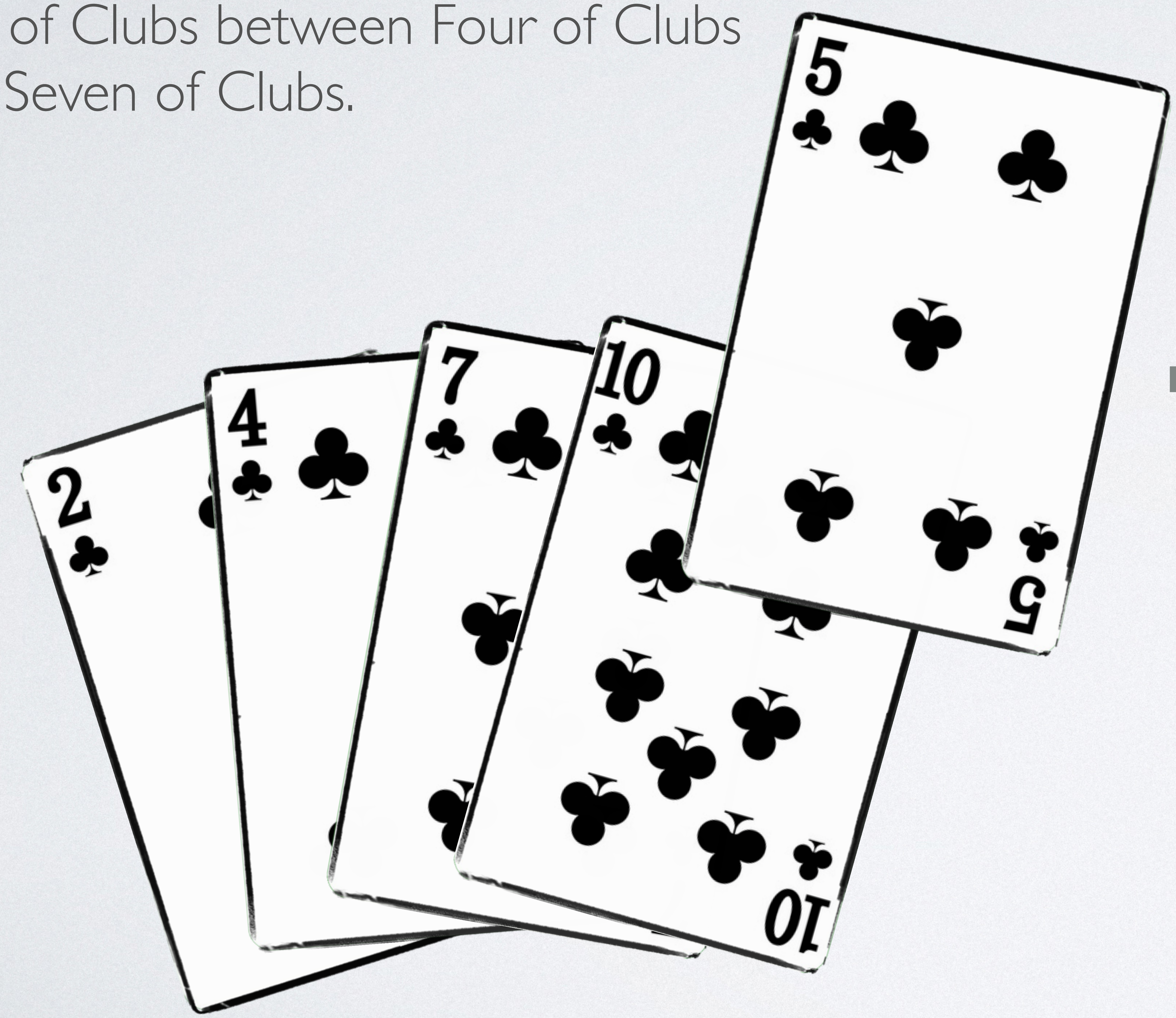
EXAMPLE: INSERTION SORT

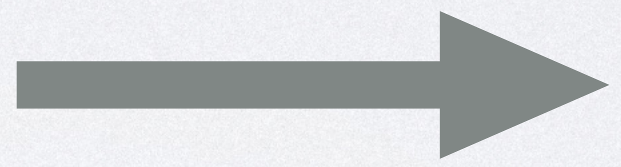
© Dr Markus Lumpe, 2024

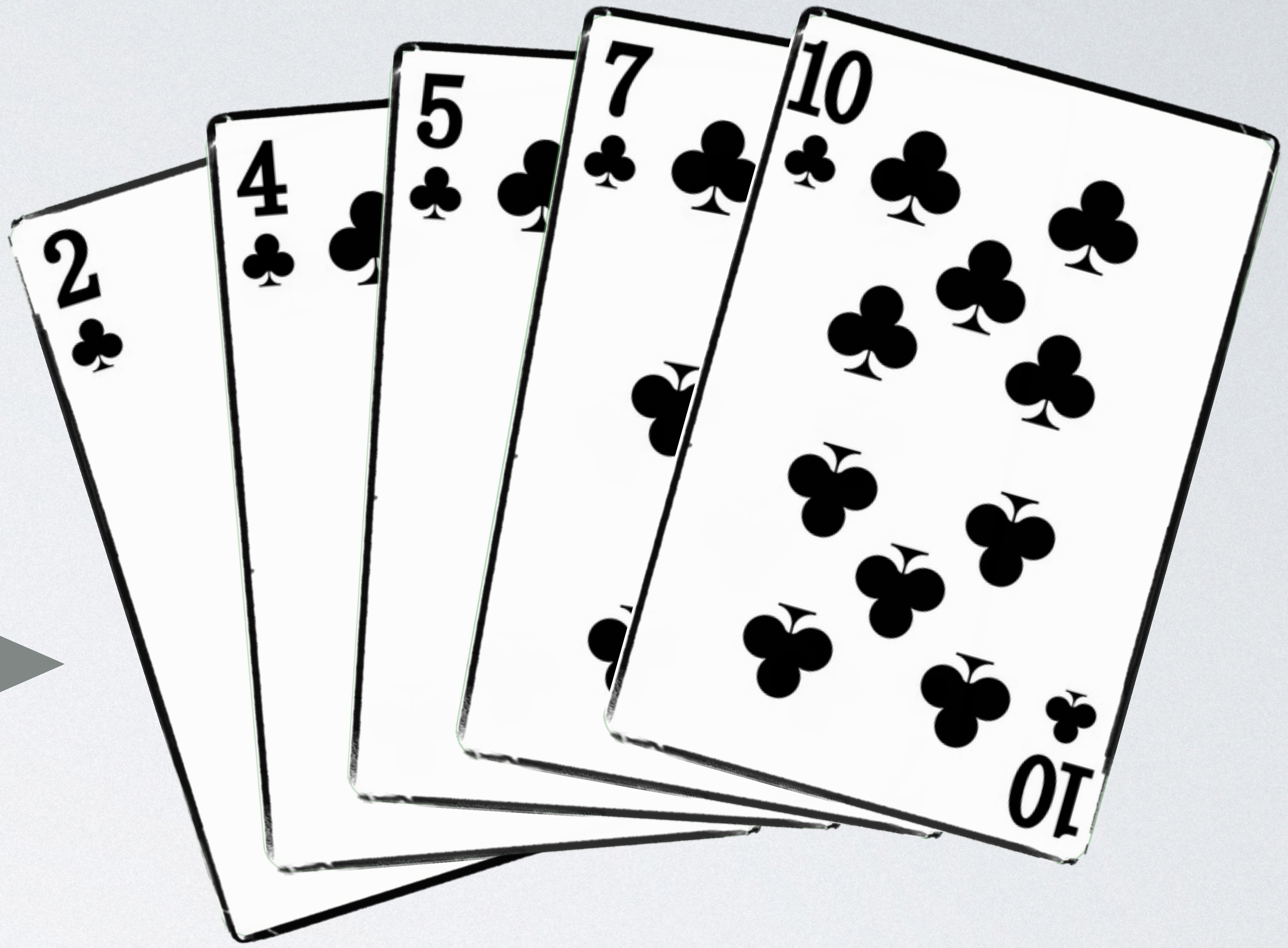
SORTING A HAND OF CARDS

Problem:

Given a sorted hand of cards insert Five of Clubs between Four of Clubs and Seven of Clubs.







Solution:

1) Add Five of Clubs at right end of the hand.
2) Move Five of Clubs to its proper position by swapping cards out of order from right to left.

© Dr Markus Lumpe, 2024

SORTING AN ARRAY: INSERTION SORT

- We can generalize the sorting of a hand of card to the problem of sorting an array of n elements using insertion sort.
- Let A be an array of n elements and let A[0..n-1] denote the sequence of elements in the array indexed from 0 to n-1. That is, arrays use 0-based indices to address elements in an array.
- For the purpose of sorting, we parse the array from left to right, that is, we start at index 0 and stop at index n for an array of n elements.
- When we consider the sorting problem of arrays, we observe that every array of length one is sorted. This is the starting point or base case of insertion sort.
- For arrays with length two or greater, let A[0..i-1] denote a sorted prefix of A[0..n-1]. The next element is at index i. If $i < n$, move A[i] into v and let j be i -1. Find the smallest index j with $i - 1 \geq j \geq 0$ and $A[j] > v$. If j has not yet been found, move A[j] into A[j+1]. Otherwise, move v into A[j+1]. Prefix A[0..i] is sorted.
- The process ends if i equals n. The array is sorted.

© Dr Markus Lumpe, 2024

DESIGN OF INSERTION-SORT

- We first define the insertion sort algorithm as a procedure *Insertion-Sort* in pseudocode.
- Pseudocode allows us to design an algorithm independent from a specific programming language and study its behavior prior implementation, say using hand execution.
- The procedure *Insertion-Sort* takes as parameter an array A[0..n-1] containing a sequence of length n to be sorted. In code, we denote n, the number of elements in the sequence, by A.length.
- The algorithm sorts the input *in place*: it rearranges the elements within the array A, with a constant number of elements stored outside the array at any given time.
- The input array A contains the sorted output sequence when the *Insertion-Sort* procedure is finished.

© Dr Markus Lumpe, 2024

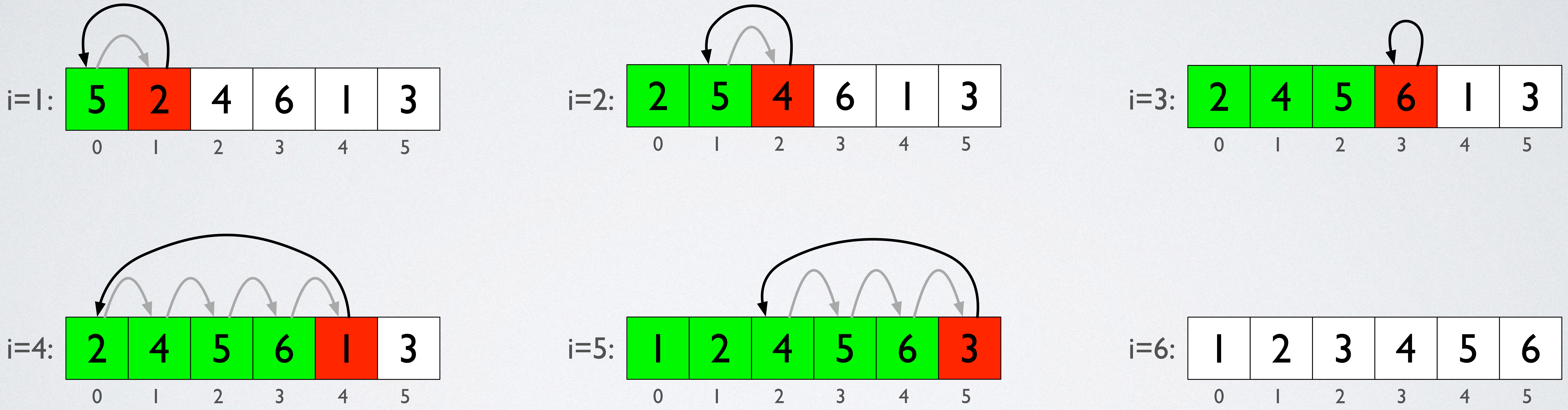
INSERTION SORT ALGORITHM: PSEUDOCODE

```
Insertion-Sort(A)
  **for** i := 1 **to** A.length - 1
      // *Insert A[i] into the sorted sequence A[0..i]*
      v := A[i]
      j := i - 1
      **while** $j \geq 0$ and $A[j] > v$
          A[j+1] := A[j]
          j := j - 1
      A[j+1] = v
```

© Dr Markus Lumpe, 2024

ILLUSTRATION OF INSERTION-SORT

- The operations of Insertion-Sort on an array A = [5,2,4,6,1,3]:



© Dr Markus Lumpe, 2024

MAPPING TO PYTHON

```
def InsertionSort(A):

    for i in range(1,len(A)):
        # Insert A[i] into the sorted sequence A[0..i]
        v = A[i]
        j = i - 1

        while j >= 0 and A[j] > v:
            A[j+1] = A[j]
            j = j - 1

        A[j+1] = v
```

```
if __name__ == '__main__':

    A = [5,2,4,6,1,3]

    print(A)
    InsertionSort(A)
    print(A)
```

```
[5, 2, 4, 6, 1, 3]
[1, 2, 3, 4, 5, 6]
```

© Dr Markus Lumpe, 2024

MAPPING TO C*

```
void InsertionSort(int A[], int Length)
{
    for ( int i = 1; i < Length; i++ )
    {
        // Insert A[i] into the sorted sequence A[0..i]
        int v = A[i];
        int j = i - 1;

        while ( j >= 0 && A[j] > v )
        {
            A[j+1] = A[j];
            j = j - 1;
        }

        A[j+1] = v;
    }
}
```

```
int main(int argc, const char * argv[])
{
    int A[] = {5, 2, 4, 6, 1, 3};

    PrintArray(A, 6);
    InsertionSort(A, 6);
    PrintArray(A, 6);

    return 0;
}
```

```
[5, 2, 4, 6, 1, 3]
[1, 2, 3, 4, 5, 6]
```

* We must also define a procedure PrintArray().

© Dr Markus Lumpe, 2024

MAPPING TO C# - TOP-LEVEL STATEMENTS

```
void InsertionSort(int[] A)
{
    for (int i = 1; i < A.Length; i++)
    {
        // Insert A[i] into the sorted sequence A[0..i]
        int v = A[i];
        int j = i - 1;

        while (j >= 0 && A[j] > v)
        {
            A[j + 1] = A[j];
            j--;
        }

        A[j + 1] = v;
    }
}
```

```
int[] myArray = new int [] { 5, 2, 4, 6, 1, 3 };

PrintArray(myArray);
InsertionSort(myArray);
PrintArray(myArray);
```

```
[5, 2, 4, 6, 1, 3]
[1, 2, 3, 4, 5, 6]
```

* We must also define a procedure PrintArray(). Requires C# top-level statements.

© Dr Markus Lumpe, 2024