On this course we concentrate on the algorithmic ideas and therefore usually represent the algorithms in pseudocode without legality checks, error handling etc.

Let's take, for example, an algorithm suitable for sorting small arrays called INSERTION-SORT:

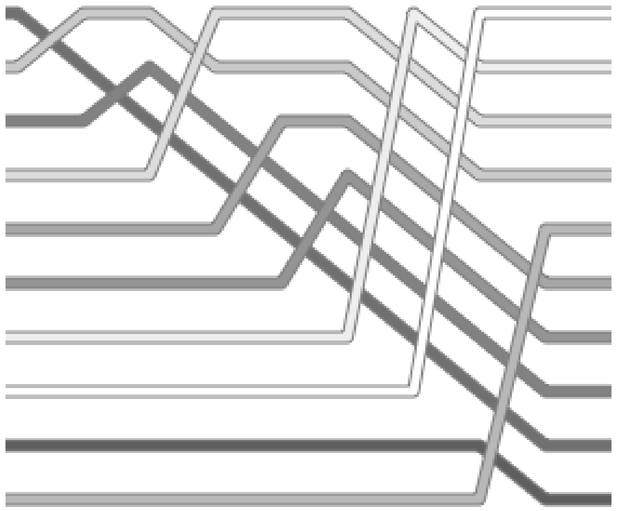
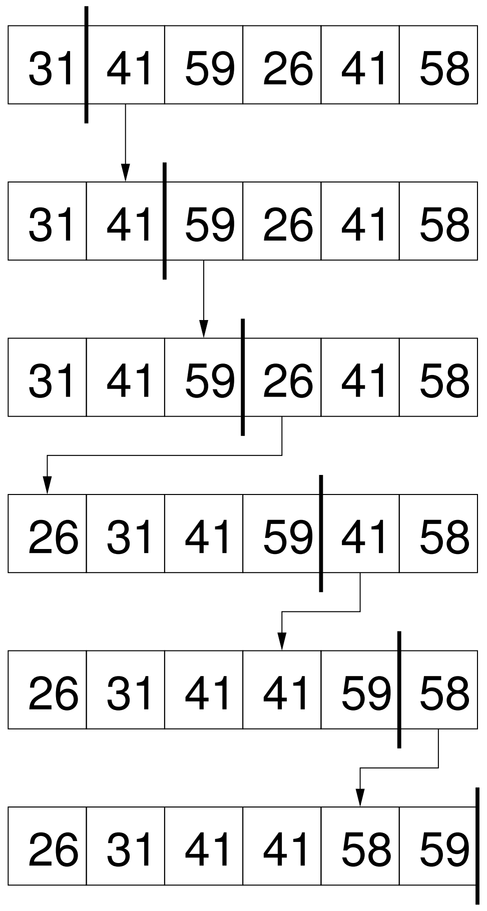


Figure 1: picture from Wikipedia

- the basic idea:
  - during execution the leftmost elements in the array are sorted and the rest are still unsorted
  - the algorithm starts from the second element and iteratively steps through the elements upto the end of the array
- on each step the algorithm searches for the point in the sorted part of the array, where the first element in the unsorted range should go to.
  - room is made for the new element by moving the larger elements one step to the right
  - the element is placed to it's correct position and the size of the sorted range in the beginning of the array is incremented by one.



In pseudocode used on the course INSERTION-SORT looks like this:

```
INSERTION-SORT($A$)                         (input in array $A$)
1  for $j$ := $2$ to $A.length$ do          (increment the size of the sorted range)
2      $key$ := $A[j]$                      (handle the first unsorted element)
3      $i$ := $j - 1$
4      while $i > 0$ and $A[i]$ > $key$ do  (find the correct location for the new element)
5          $A[i + 1]$ := $A[i]$             (make room for the new element)
6          $i$ := $i - 1$
7      $A[i + 1]$ := $key$                  (set the new element to its correct location)
```

- indentation is used to indicate the range of conditions and loop structures
- *(comments)* are written in parentheses in italics
- the ":=" is used as the assignment operator ("=" is the comparison operator)
- the lines starting with the character ▷ give textual instructions

- members of structure elements (or objects) are referred to with the dot notation.
  - e.g. $student.name$, $student.number$
- the members of a structure accessed through a pointer x are referred to with the $\rightarrow$ character
  - e.g. $x{\rightarrow}name$, $x{\rightarrow}number$
- variables are local unless mentioned otherwise
- a collection of elements, an array or a pointer, is a **reference** to the collection
  - larger data structures like the ones mentioned should always be passed by reference
- a pass-by-value mechanism is used for single parameters (just like C++ does)
- a pointer or a reference can also have no target: NIL

2.3 Implementing algorithms

In the real world you need to be able to use theoretical knowledge in practise.

For example: apply a given sorting algorithm ins a certain programming problem

- numbers are rarely sorted alone, we sort structures with
  - a *key*
  - *satellite data*
- the key sets the order
  $\Rightarrow$ it is used in the comparisons
- the satellite data is not used in the comparison, but it must be moved around together with the key

The INSERTION-SORT algorithm from the previous chapter would change as follows if there were some satellite data used:

```
for $j$ := $2$ to $A.length$ do
    $temp$ := $A[j]$
    $i$ := $j - 1$
    while $i > 0$ and $A[i].key > temp.key$ do
        $A[i+1]$ := $A[i]$
        $i$ := $i - 1$
    $A[i+1]$ := $temp$
```

- An array of pointers to structures should be used with a lot of satellite data. The sorting is done with the pointers and the structures can then be moved directly to their correct locations.

The programming language and the problem to be solved also often dictate other implementation details, for example:

- Indexing starts from 0 (in pseudocode often from 1)
- Is indexing even used, or some other method of accessing data (or do we use arrays or some other data structures)
- (C++) Is the data really inside the array/datastructure, or somewhere else at the end of a pointer (in which case the data doesn't have to be moved and sharing it is easier). Many other programming languages always use pointers/references, so you don't have to choose.
- If you refer to the data indirectly from elsewhere, does it happen with
  - Pointers (or references)
  - Smart pointers (C++, `shared_ptr`)
  - Iterators (if the data is inside a datastructure)
  - Index (if the data is inside an array)
  - Search key (if the data is insde a data structure with fast search)

- Is recursion implemented really as recursion or as iteration
- Are algorithm "parameters" in pseudocode really parameters in code, or just variables

In order to make an executable program, additional information is needed to implement INSERTION-SORT

- an actual programming language must be used with its syntax for defining variables and functions
- a main program that takes care of reading the input, checking its legality and printing the results is also needed
  - it is common that the main is longer than the actual algorithm

The implementation of the program described abowe in C++:

```
#include <iostream>
#include <vector>
typedef std::vector<int> Array;

void insertionSort( Array & A ) {
  int key, i; unsigned int j;
  for( j = 1; j < A.size(); ++j ) {
    key = A.at(j); i = j-1;
    while( i >= 0 && A.at(i) > key ) {
      A.at(i+1) = A.at(i); --i;
    }
    A.at(i+1) = key;
  }
}

int main() {
  unsigned int i;
  // getting the amount of elements
  std::cout << "Give the size of the array 0...: "; std::cin >> i;
  Array A(i); // creating the array
  // reading in the elements
  for( i = 0; i < A.size(); ++i ) {
    std::cout << "Give A[" << i+1 << "]: ";
    std::cin >> A.at(i);
  }
  insertionSort( A );   // sorting

  // print nicely
  for( i = 0; i < A.size(); ++i ) {
    if( i % 5 == 0 ) {
      std::cout << std::endl;
    }
    else {
      std::cout << " ";
    }
    std::cout << A.at(i);
  }
  std::cout << std::endl;
}
```

The program code is significantly longer than the pseudocode. It is also more difficult to see the central characteristics of the algorithm.

This course concentrates on the principles of algorithms and data structures. Therefore using program code doesn't serve the goals of the course.

$\Rightarrow$ From now on, program code implementations are not normally shown.

3 Algorithm design techniques

3.1 Algorithm Design Technique: Decrease and conquer

The most straightforward algorithm *design technique* covered on the course is *decrease and conquer*.

- initially the entire input is unprocessed
- the algorithm processes a small piece of the input on each round

  $\Rightarrow$ the amount of processed data gets larger and the amount of unprocessed data gets smaller
- finally there is no unprocessed data and the algorithm halts

These types of algorithms are easy to implement and work efficiently on small inputs.

The Insertion-Sort seen earlier is a “decrease and conquer” algorithm.

- initially the entire array is (possibly) unsorted
- on each round the size of the sorted range in the beginning of the array increases by one element
- in the end the entire array is sorted

INSERTION-SORT

```
INSERTION-SORT($A$)                                   (input in array $A$)
1  for $j$ := $2$ to $A.length$ do                    (move the limit of the sorted range)
2      $key$ := $A[j]$                                (handle the first unsorted element)
3      $i$ := $j - 1$
4      while $i > 0$ and $A[i]$ > $key$ do            (find the correct location of the new element)
5          $A[i+1]$ := $A[i]$                         (make room for the new element)
6          $i$ := $i - 1$
7      $A[i+1]$ := $key$                              (set the new element to it's correct location)
```

3.2 Algorithm Design Technique: Divide and Conquer

We've earlier seen the *decrease and conquer* algorithm design technique and the algorithm INSERTION-SORT as an example of it.

Now another technique called *divide and conquer* is introduced. It is often more efficient than the decrease and conquer approach.

- the problem is divided into several subproblems that are like the original but smaller in size.
- small subproblems are solved straightforwardly
- larger subproblems are further divided into smaller units
- finally the solutions of the subproblems are combined to get the solution to the original problem

Let's get back to the claim made earlier about the complexity notation not being fixed to programs and take an everyday, concrete example

3.3 QUICKSORT

Let's next cover a very efficient sorting algorithm QUICKSORT.

QUICKSORT is a divide and conquer algorithm.

The division of the problem into smaller subproblems

- Select one of the elements in the array as a *pivot*, i.e. the element which partitions the array.
- Change the order of the elements in the array so that all elements smaller or equal to the pivot are placed before it and the larger elements after it.
- Continue dividing the upper and lower halves into smaller subarrays, until the subarrays contain 0 or 1 elements.

Smaller subproblems:

- Subarrays of the size 0 and 1 are already sorted

Combining the sorted subarrays:

- The entire (sub) array is automatically sorted when its upper and lower halves are sorted.
  - all elements in the lower half are smaller than the elements in the upper half, as they should be

QUICKSORT-algorithm

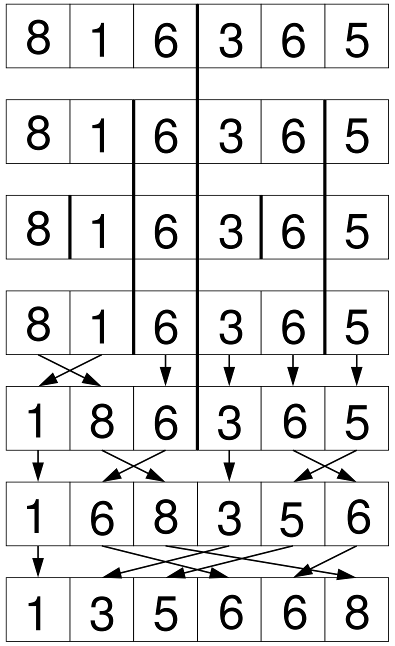
```
QUICKSORT($A, left, right$)
1   if $left < right$ then                       (do nothing in the trivial case)
2       $pivot$ := PARTITION($A, left, right$)   (partition in two)
3       QUICKSORT($A, left, pivot - 1$)          (sort the elements smaller than the pivot)
4       QUICKSORT($A, pivot + 1, right$)         (sort the elements larger than the pivot)
```

The *partition algorithm* rearranges the subarray in place

```
PARTITION($A, left, right$)
1   $pivot$ := $A[right]$                        (choose the last element as the pivot)
2   $i$ := $left - 1$                            (use $i$ to mark the end of the smaller elements)
4   for $j$ := $left$ to $right - 1$ do          (scan to the second to last element)
6       if $A[j] \leq pivot$                     (if $A[j]$ goes to the half with the smaller elements...)
9           $i$ := $i + 1$                       (... increment the amount of the smaller elements...)
12          exchange $A[i] \leftrightarrow A[j]$ (... and move $A[j]$ there)
12  exchange $A[i+1] \leftrightarrow A[right]$   (place the pivot between the halves)
13  return $i + 1$                               (return the location of the pivot)
```

MERGE-SORT

- divide the elements in the array into two halves.
- continue dividing the halves further in half until the subarrays contain atmost one element
- arrays of 0 or 1 length are already sorted and require no actions
- finally merge the sorted subarrays



```
MERGE-SORT($A, left, right$)
1   if $left < right$ then                           (if there are elements in the array...)
2       $middle$ := $\lfloor (left + right)/2 \rfloor$       (... divide it into half)
3       MERGE-SORT($A, left, middle$)                 (sort the upper half...)
4       MERGE-SORT($A, middle + 1, right$)            (... and the lower)
5       MERGE($A, left, middle, right$)               (merge the parts maintaining the order)
```

- the MERGE-algorithm for merging the subarrays:

```
MERGE($A, left, middle, right$)
1  for $i$ := $left$ to $right$ do      (scan through the entire array...)
2      $B[i]$ := $A[i]$                  (... and copy it into a temporary array)
3  $i$ := $left$                         (set $i$ to indicate the endpoint of the sorted part)
4  $j$ := $left$; $k$ := $middle + 1$   (set $j$ and $k$ to indicate the beginning of the subarrays)
5  while $j \leq middle$ and $k \leq right$ do      (scan until either half ends)
6      if $B[j] \leq B[k]$ then          (if the first element in the lower half is smaller...)
7          $A[i]$ := $B[j]$              (... copy it into the result array...)
8          $j$ := $j + 1$                (... increment the starting point of the lower half)
9      else                              (else...)
10         $A[i]$ := $B[k]$              (... copy the first element of the upper half...)
11         $k$ := $k + 1$                (... and increment its starting point)
12     $i$ := $i + 1$                    (increment the starting point of the finished set)
13 if $j > middle$ then
14     $k$ := $0$
15 else
16     $k$ := $middle - right$
17 for $j$ := $i$ to $right$ do          (copy the remaining elements to the end of the result)
18     $A[j]$ := $B[j + k]$
```