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$(A, left, right)$

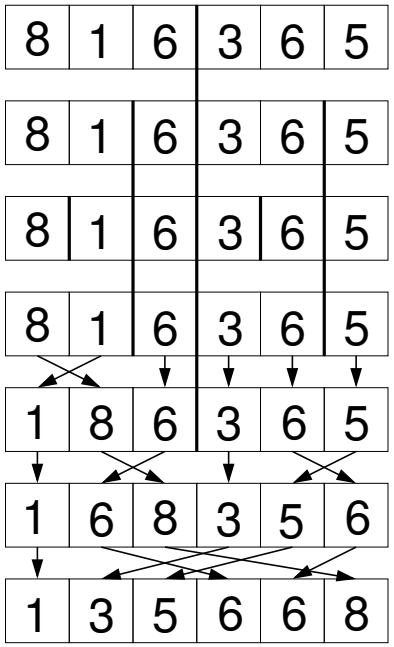
```
1  if left < right then                          (do nothing in the trivial case)
2      pivot := PARTITION(A, left, right)        (partition in two)
3      QUICKSORT(A, left, pivot − 1)             (sort the elements smaller than the pivot)
4      QUICKSORT(A, pivot + 1, right)            (sort 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)
3  for j := left to right − 1 do        (scan to the second to last element)
4      if A[j] ≤ pivot                  (if A[j] goes to the half with the smaller elements...)
5          i := i + 1                   (... increment the amount of the smaller elements...)
6          exchange A[i] ↔ A[j]         (... and move A[j] there)
7  exchange A[i + 1] ↔ A[right]         (place the pivot between the halves)
8  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 := ⌊(left + right)/2⌋        (... 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 ≤ middle and k ≤ right do      (scan until either half ends)
6       if B[j] ≤ 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]
```