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

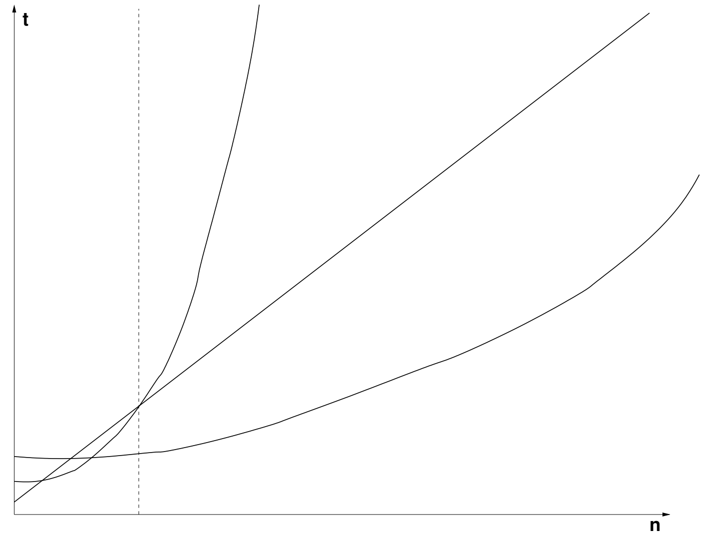
INSERTION-SORT

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

4 Measuring efficiency

This chapter dicusses the analysis of algorithms: the efficiency of algorithms and the notations used to describe the *asymptotic* behavior of an algorithm.

In addition the chapter introduces two algorithm design techniques: *decrease and conquer* and *divide and conquer*.



4.1 Asymptotic notations

It is occasionally important to know the exact time it takes to perform a certain operation (in real time systems for example).

Most of the time it is enough to know how the running time of the algorithm changes as the input gets larger.

- The advantage: the calculations are not tied to a given processor, architecture or a programming language.
- In fact, the analysis is not tied to programming at all but can be used to describe the efficiency of any behaviour that consists of successive operations.

- The time efficiency analysis is simplified by assuming that all operations that are independent of the size of the input take the same amount of time to execute.
- Furthermore, the amount of times a certain operation is done is irrelevant as long as the amount is constant.
- We investigate how many times each row is executed during the execution of the algorithm and add the results together.

- The result is further simplified by removing any constant coefficients and lower-order terms.

  $\Rightarrow$ This can be done since as the input gets large enough the lower-order terms get insigficant when compared to the leading term.

  $\Rightarrow$ The approach naturally doesn't produce reliable results with small inputs. However, when the inputs are small, programs usually are efficient enough in any case.

- The final result is the efficiency of the algorithm and is denoted it with the greek alphabet theta, Θ.

$$f(n) = 23n^2 + 2n + 15 \Rightarrow f \in \Theta(n^2)$$

$$f(n) = \frac{1}{2}n \lg n + n \Rightarrow f \in \Theta(n \lg n)$$

Example 1: addition of the elements in an array

```
1  for i := 1 to A.length do
2      sum := sum + A[i]
```

- if the size of the array A is n, line 1 is executed n + 1 times
- line 2 is executed n times
- the running time increases as n gets larger:

n	time = 2n + 1
1	3
10	21
100	201
1000	2001
10000	20001

- notice how the value of n dominates the running time

- let's simplify the result as described earlier by taking away the constant coefficients and the lower-order terms:

  $f(n) = 2n + 1 \Rightarrow n$

  $\Rightarrow$ we get $f \in \Theta(n)$ as the result

  $\Rightarrow$ the running time depends *linearly* on the size of the input.

Example 2: searching from an unsorted array

```
for i := 1 to A.length do
    if A[i] = x then
        return i
```

- the location of the searched element in the array affects the running time.
- the running time depends now both on the size of the input and on the order of the elements
  $\Rightarrow$ we must separately handle the best-case, worst-case and average-case efficiencies.

- in the best case the element we're searching for is the first element in the array.
  $\Rightarrow$ the element is found in *constant time*, i.e. the efficiency is $\Theta(1)$
- in the worst case the element is the last element in the array or there are no matching elements.
- now line 1 gets executed n + 1 times and line 2 n times
  $\Rightarrow$ efficiency is $\Theta(n)$.
- determining the average-case efficiency is not as straightforward

- first we must make some assumptions on the average, typical inputs:
  - the probability p that the element is in the array is ($0 \leq p \leq 1$)
  - the probability of finding the first match in each position in the array is the same
- we can find out the average amount of comparisons by using the probabilities
- the probability that the element is not found is 1 - p, and we must make n comparisons
- the probability for the first match occuring at the index i, is p/n, and the amount of comparisons needed is i
- the number of comparisons is:

$$[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

- if we assume that the element is found in the array, i.e. p = 1, we get (n+1)/2 which is $\Theta(n)$

  $\Rightarrow$ since also the case where the element is not found in the array has linear efficiency we can be quite confident that the average efficiency is $\Theta(n)$

- it is important to keep in mind that all inputs are usually not as probable.

  $\Rightarrow$ each case needs to be investigated separately.

Example 3: finding the common element in two arrays

```
1 for i := 1 to A.length do
2     for j := 1 to B.length do
3         if A[i] = B[j] then
4             return A[i]
```

- line 1 is executed 1 – (n + 1) times
- line 2 is executed 1 – ($n \cdot (n$ + 1)) times
- line 3 is executed 1 – ($n \cdot n$) times
- line 4 is executed at most once

- the algorithm is fastest when the first element of both arrays is the same
  $\Rightarrow$ the best case efficiency is $\Theta(1)$
- in the worst case there are no common elements in the arrays or the last elements are the same
  $\Rightarrow$ the efficiency is $2n^2 + 2n + 1 = \Theta(n^2)$
- on average we can assume that both arrays need to be investigated approximately half way through.
  $\Rightarrow$ the efficiency is $\Theta(n^2)$ (or $\Theta(nm)$ if the arrays are of different lengths)