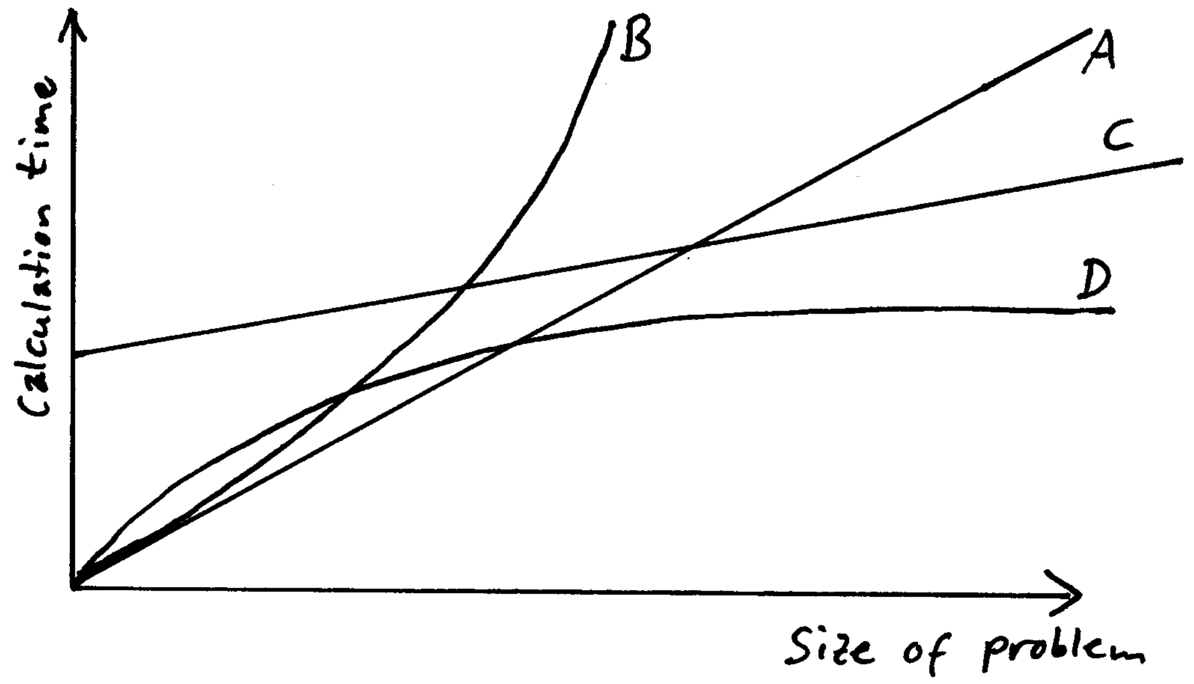
7. Time complexity of algorithms

"Complexity" can have different meanings.

In computer science most often:
"time complexity" = amount of calculation time which an algorithm needs
"space complexity" = amount of memory which is needed by an algorithm

Algorithms *solving the same problem* can behave very differently:



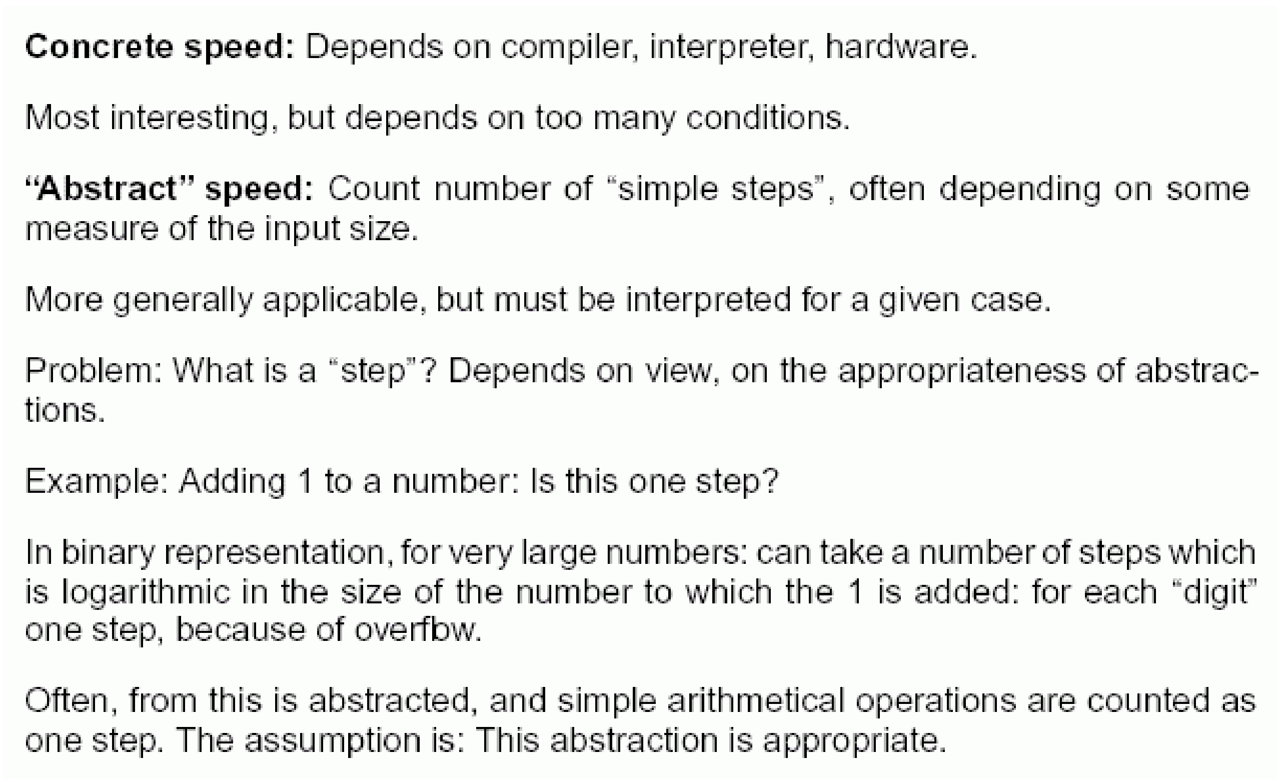
Algorithm A needs an amount of time proportional to the size of the problem (input size) – it "scales linearly"

B has nonlinear behaviour and is worse than A

C: linear, worse than A for small problem size, but better for large problem size
D is even better for large problems

How to make these considerations more precise?

What shall be measured?



Concrete speed: Depends on compiler, interpreter, hardware.

Most interesting, but depends on too many conditions.

“Abstract” speed: Count number of “simple steps”, often depending on some measure of the input size.

More generally applicable, but must be interpreted for a given case.

Problem: What is a “step”? Depends on view, on the appropriateness of abstractions.

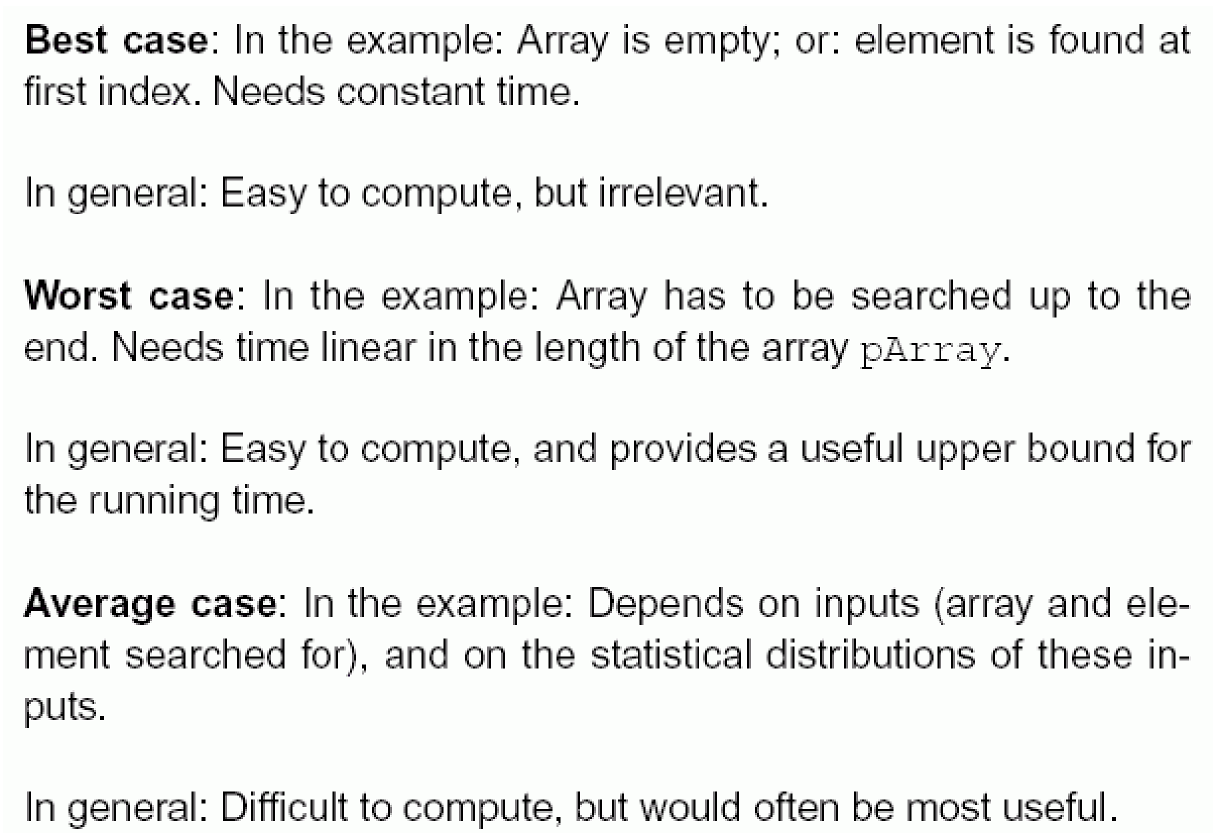
Example: Adding 1 to a number: Is this one step?

In binary representation, for very large numbers: can take a number of steps which is logarithmic in the size of the number to which the 1 is added: for each “digit” one step, because of overfbw.

Often, from this is abstracted, and simple arithmetical operations are counted as one step. The assumption is: This abstraction is appropriate.

"size of input" as parameter – but input can be very different, even if it has the same size
(e.g., array of numbers: sorted or unsorted)

Variants of time complexity:



Best case: In the example: Array is empty; or: element is found at first index. Needs constant time.

In general: Easy to compute, but irrelevant.

Worst case: In the example: Array has to be searched up to the end. Needs time linear in the length of the array `pArray`.

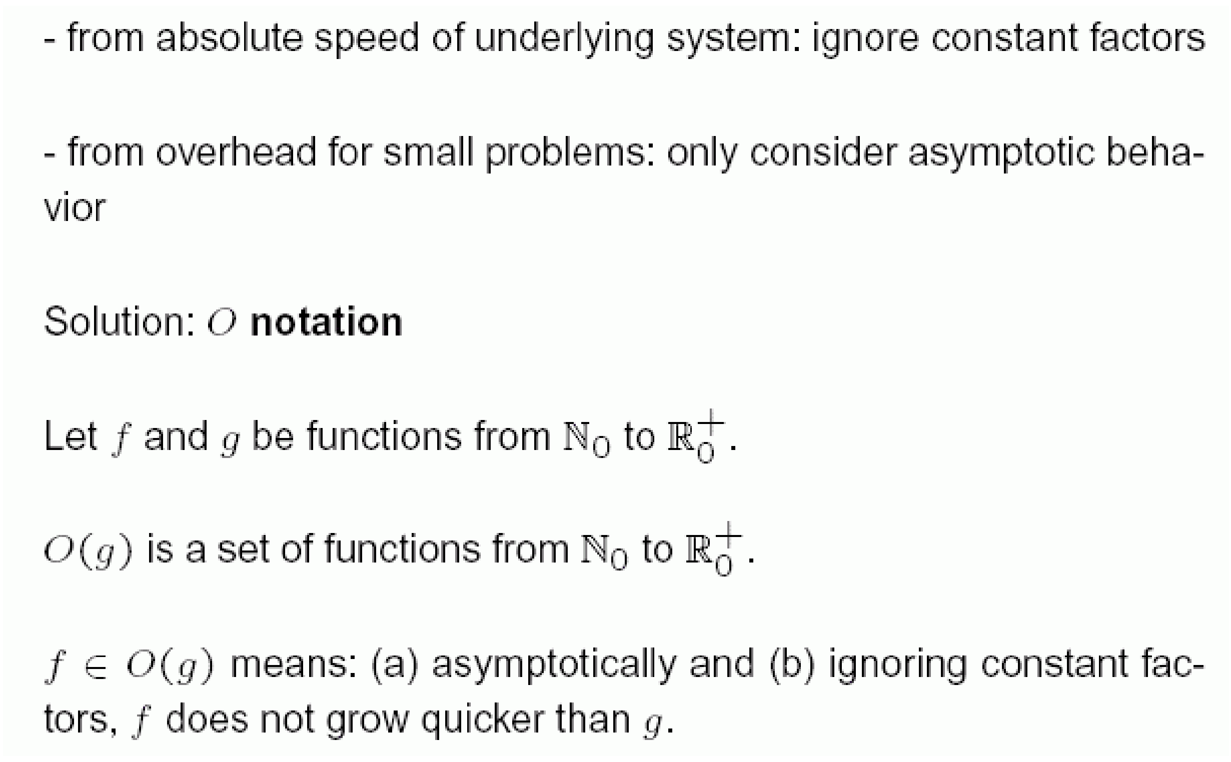
In general: Easy to compute, and provides a useful upper bound for the running time.

Average case: In the example: Depends on inputs (array and element searched for), and on the statistical distributions of these inputs.

In general: Difficult to compute, but would often be most useful.

How to describe different "growth forms" of functions?

$\rightarrow$ Bachmann-Landau "*O* Notation" (uppercase "O", not a zero!) — for short: *O* calculus



- from absolute speed of underlying system: ignore constant factors

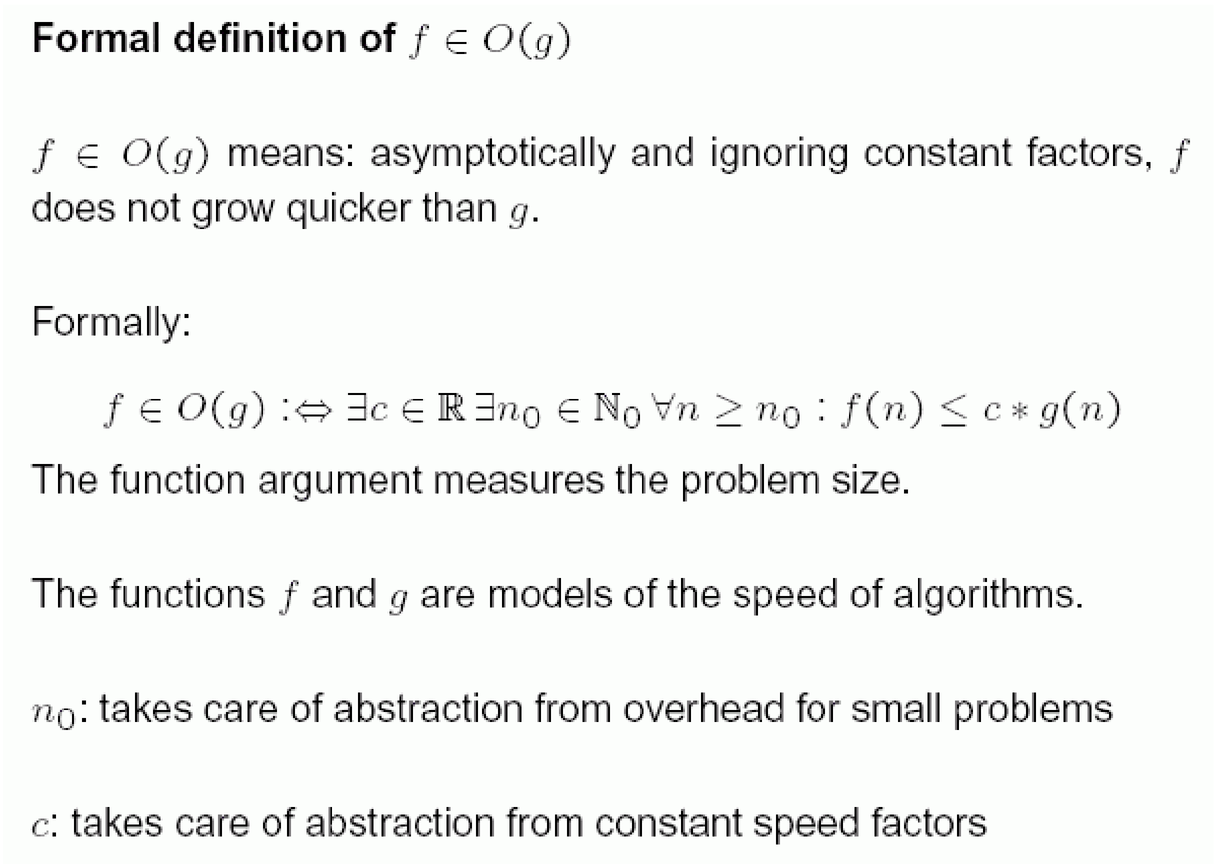
- from overhead for small problems: only consider asymptotic behavior

Solution: O **notation**

Let f and g be functions from $\mathbb{N}_0$ to $\mathbb{R}_0^+$.

$O(g)$ is a set of functions from $\mathbb{N}_0$ to $\mathbb{R}_0^+$.

$f \in O(g)$ means: (a) asymptotically and (b) ignoring constant factors, f does not grow quicker than g.



Formal definition of $f \in O(g)$

$f \in O(g)$ means: asymptotically and ignoring constant factors, f does not grow quicker than g.

Formally:

$$f \in O(g) :\Leftrightarrow \exists c \in \mathbb{R}\, \exists n_0 \in \mathbb{N}_0\, \forall n \geq n_0 : f(n) \leq c * g(n)$$

The function argument measures the problem size.

The functions f and g are models of the speed of algorithms.

n_0: takes care of abstraction from overhead for small problems

c: takes care of abstraction from constant speed factors

Features of the O Notation

Often: n used as implicit argument.

Example: **Linear functions:** $O(n)$ is the set of functions whose values, in the long run, increase at most linearly with the argument.

Constant factors are irrelevant:

$$O(n) = O(1000 \times n) = O(n \times 1E - 2000)$$

Function values up to some finite value are irrelevant:

$$O(n) = O(\text{if } n < 10000 \text{ then } 1000000 \text{ else } n \text{ endif})$$

Appropriateness of O Abstraction

Run times of algorithms are often given by O-sets.

Consider inputs of size n.

- The run times for small arguments are irrelevant.
- Constant differences between run times are ignored: $O(f(n)+10000) = O(f(n))$.
- Constant factors between between run times are ignored: $O(c \cdot f(n)) = O(f(n))$.

These abstractions are not always appropriate! When are they?

Important classes of algorithms, according to their runtime behaviour:

Programs using a number of steps in $O(n)$ (for an input of size n) are said to need **linear** time.

If the number of steps is in $O(1)$, then the run time is called **constant**. We have $\forall c \in \mathbb{N} : O(c) = O(1)$.

If the number of steps is in $O(n^2)$, then the run time is called **quadratic**; $O(n^3)$ means: **cubic**.

If the number of steps is in $O(\log n)$, then the run time is called **logarithmic**.

A function f is called **polynomially increasing** if there is some polynomial $p(n)$ such that $f \in O(p)$.

Thus: The polynomial is an **upper** bound for f.

A function g is called **exponentially increasing** if $2^n \in O(g)$.

Thus: The exponential function is a **lower** bound for g.

Computation rules for the *O* notation

Shorter notation: using operators on sets for element-wise application.

- $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
- $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
- $O(\log_c n) = O(\log n)$, because of ($\log_c n = \frac{\log n}{\log c}$)

Example:

Analysis of a search algorithm

Given:

- an unsorted integer array, **`pArray`**.
- a single integer, **`pElem`**.

Wanted:
the first index **`i`** where **`pElem`** is found as an element of the array (i.e., **`pArray[i] == pElem`**).
If it cannot be found: result –1.

Algorithm: *Sequential (or linear) search*.

Goes step by step through the array, starting with **`i=0`**.

```
// Check if (pElem) occurs in (pArray).  If it does, return first
//   index such that (pArray[i]==pElem), if it does not, return -1.
static int searchForElement_linear(int pElem, int[] pArray)
{
  // Run through the array, looking for (pElem).
  for(int i=0; i<pArray.length; ++i)
  {
    // If we found the element: return corresponding index.
    if(pArray[i] == pElem) return i;
  }

  // Element not found: Return -1.
  return -1;
}
```

Analysis of this algorithm:

Worst case run time (number of steps) of searching in an unsorted list of length n (size of input).

Counting steps for worst case:

- Initialization: `i=0`: Constant

- For each array element: comparison `i<pArray.length`, comparison `pArray[i] == pElem`, increment `++i`: $3n$, if each such statement needs one step.

- At the end: comparison `i<pArray.length`, return: Constant

In sum: $c_1 + 3n + c_2 \in O(n)$: **Linear** running time.

Average case: Which assumptions are appropriate?

Let us now assume that the array pArray is already sorted, i.e., the elements have ascending order:
`pArray[0] <= pArray[1] <= pArray[2] <=` ...

Then we can use the same algorithm as above

- but we can do better:

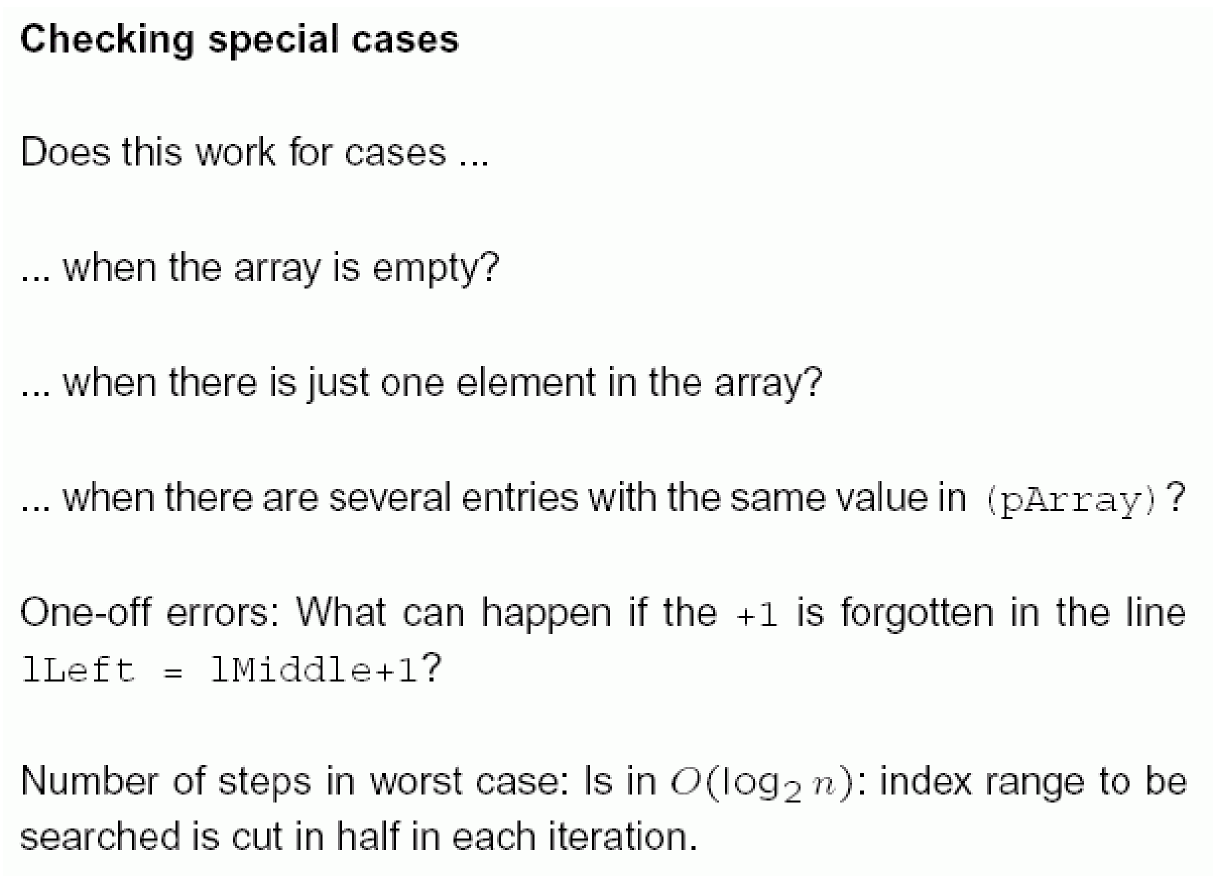
Algorithm: *Binary search*

```
// Check if (pElem) occurs in (pArray).  If it does, return an
//   index such that (pArray[i]==pElem).  If it does not, return -1.
//   You may assume that (pArray) is sorted in ascending orders.
static int searchForElement_sorted(int pElem, int[] pArray)
{
  int lLeft = 0, lRight = pArray.length;

  // If the element is in the array, it has an index larger than or
  //   equal to (lLeft), and strictly smaller than (lRight).
  while(lRight-lLeft > 0)
  {
    // Find the index halfway between (lLeft) and (lRight), rounding
    //   downwards, if necessary.
    int lMiddle = (lLeft+lRight)/2;

    if(pArray[lMiddle] == pElem)     return lMiddle;
    else if(pArray[lMiddle] < pElem) lLeft = lMiddle+1;
    else                             lRight = lMiddle;
  }

  // We did not find the element.
  return -1;
}
```

Example: we look for the number 11 in the following array:

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
pArray[i]	**1**	**2**	**4**	**6**	**7**	**8**	**10**	**11**	**13**	**17**	**19**

go first to the central index, **`lMiddle`**
(0+10)/2 = 5,   there we have 8 < 11 $\Rightarrow$ go to the right
there look again at the central index:
((5+1)+10)/2 = 8, there 13 > 11 $\Rightarrow$ go to the left,
split again the array... until 11 is found



Checking special cases

Does this work for cases ...

... when the array is empty?

... when there is just one element in the array?

... when there are several entries with the same value in `(pArray)`?

One-off errors: What can happen if the `+1` is forgotten in the line `lLeft = lMiddle+1`?

Number of steps in worst case: Is in $O(\log_2 n)$: index range to be searched is cut in half in each iteration.

Further example: Two sorting algorithms

Task: To bring the elements of an unsorted integer array in ascending order

(sorting into *descending* order can then be done in analogous way!)

How to solve it?

Idea: bring the largest element to the leftmost position by pairwise exchanging neighbouring elements
repeat this in the still unsorted first part of the array

$\rightarrow$ effect: the large elements go like "bubbles" through the array to their appropriate places

Algorithm "Bubble sort"

```
// Sort (pArray) with the bubble sort algorithm.
static void bubbleSort(int[] pArray)
{
  for(int i=pArray.length; i>1; --i)
  {
    for(int j=0; j<i-1; ++j)
    {
      if(pArray[j]>pArray[j+1])
      {
        // Exchange elements at indices (j) and (j+1).
        int k = pArray[j+1];
        pArray[j+1] = pArray[j];
        pArray[j] = k;
      }
    }
  }
}
```

Analysis of Bubble Sort Algorithm

What happens if `(pArray)` is already sorted?

What must be changed in the program to recognize early if the array is sorted?

Input: Length of array which is searched.

Worst case: How is this computed?

Size of input: n is length of input array to be sorted.

Assumptions: Comparisons, index access, incrementation/decrementation constant time `i`: n different values, all from n down to `1`.

For each `i`, `j` assumes all values from `0` to $i-2$.

For each such pair of `i, j`, a comparison is done, and possibly, values are exchanged: constant number of steps: $O(1)$.

How many pairs are there maximally used?

$$\sum_{i=2}^{n} \sum_{j=0}^{i-2} O(1)$$

This equals $O(0+1+2+...+(n-2)) = O(\frac{(n-1)(n-2)}{2}) = O(\frac{n^2-3n+2}{2}) = O(n^2)$

Algorithms with better asymptotic running time?

Idea: Use binary splitting of the array, like in the case of binary search

Quick Sort

Idea for sorting a partial array A from index l to index u. Lets us call the input size $n := u - l + 1$. (inclusive):

- If $u - l \leq 1$, then stop: we are ready. Otherwise continue:

- **Select a pivot element** e in A in the range from l to u (inclusive). Try to select an element which has a value such that half of the other elements are greater, and half of them are smaller.

  Methods: Select some random element; or select three random elements, and take one of them with a medium value.

  Typical methods take $O(1)$ time: Negligible.

- **Split array:** Reorder array w.r.t. e such that e gets index j (i.e. $A[j] = e$), such that $l \leq i < j \Rightarrow A[i] \leq e$, and such that $j < i \leq u \Rightarrow e \leq A[i]$.

  Method: Start with two pointers l' and u' from l and u, and start with j with $A[j] = e$.

  Repeat the following until $l' \geq u'$:

  - Increment l' until an element is reached with $A[l'] > e$ or $l' \geq u'$.
  - Decrement u' until an element is reached with $A[u'] \leq e$ or $l' \geq u'$.
  - If $A[l'] > e \wedge A[u'] \leq e$: Exchange values at $A[l']$ and $A[u']$ and increment l'.

  Finally, exchange elements at $A[j]$ and $A[u']$ and set $j := u'$.

  This needs about $u - l + 1$ steps: $O(n)$.

- **Recursive call:** Sort A in range $l, \ldots, j-1$, and in the range $j+1, u$.

Idea: If the pivot is chosen well, the number of splits of the initial array is logarithmic, since the array is split in half in each split.

This times complexity lies in $O(n \log n)$.

Worst case: After reordering, the pivot is at one of the ends of the partial array: $O(n^2)$

Average case: depends on selection of pivot.

Remark: There is still another sorting algorithm which needs only *O*(*n* log *n*) time even in the worst case (Heap sort).