CHAPTER 5

Fundamental Algorithms

The big idea in Computer Science is the concept of "mechanizable activities" which we call algorithm. The origin of the name goes back to the persian-arabic scholar *Abu Ja'far Mohammed ibn Musa al-Khwarizmi* where his home town "al-Khwarizmi" was romanized to "algorismus" with the meaning of "calculation method".

We may extend the meaning of an algorithm to consist of a finite ordered sequence of precise instruction that yield a correct result under all possible inputs. With this definition we do not limit ourself to compute numbers only but are able to give precise instructions to treat any problem that is solvable in a finite sequence of steps. We may find a recipe to be a cooking algorithm or the music notes an algorithm how to play a song. The manual for the DVD-Player provides algorithms how to operate it.

This stepwise process may be shown as flow chart (see Figure **??** in Chapter 2).

Programs "explain" algorithms to the computer in a formal way and without ambiguity. Algorithms are the conceptual substance of programs that give live to the computer hardware.

CHAPTER OBJECTIVES

- To be able to classify different algorithms
- To know the basic searching and sorting algorithms
- To understand the costs in terms of space and time complexity.

5.1 Classification of Algorithms

We have already seen some algorithms in previous chapters for sorting or summing up a set of numbers. There exist a multitude of algorithms even for the same problem. When two algorithms produce the same result, their

resource consumption, implementation, or the design paradigm may still differ fundamentally.

To get a better overview it may be useful to classify algorithms. However there are may criteria to do so. We can divide the algorithms by the problem domain they address, by the design paradigm, by the implementation, by its capability, or by the resources needed. As these criteria are independent of each other, classification is a multidimensional problem.

Every field of science has its own problems to solve. Specific algorithms are needed for each problem domain, but related problems are often studied together.

The following classification by **problem domain** is widely common:

numerical algorithms help to solve (differential) equations, to numerically integrate functions (find its area under the function), or just calculate the mean value of a set of numbers.

search algorithms find entries in a list, tree, graph, or other structures. These include the shortest path problem or heuristic and fuzzy searching.

sort/merge algorithms deal with the (re)ordering of data elements according to a *sort order criteria* (see Section 5.4).

graph algorithms are used to find neighbors or routes in network. They are used to draw and do graph coloring, or calculate a minimum spanning tree for a graph. That is a subgraph which forms a tree and connects all the nodes of the graph.

string algorithms find substrings (called pattern) in a string or text. For text approximate searching with a certain similarity like phonetic matching is useful.

cryptographic algorithms are used to encipher and decipher information to ensure privacy and prevent any spoofing.

data compression algorithms help to reduce the amount of data to be transferred. It is used not only for data communication purpose but to save storage and computer resources. For multimedia data compression algorithms with loss in quality are widely used (see Section 10.6).

combinatorial algorithms compute the possible orderings or selection of objects.

Some domains in our list are missing and others overlap with each other as for example the search and graph algorithms do. Other may be further subdivided. We will not try to disentangle the situation as ongoing research will link domains or split up fields further. Computer users use this classification to search for an algorithm to solve an application problem.

Another possibility to classify algorithms is by their design method. These are programming concepts and paradigms that we will present in Chapter 7. The classification by **design method** include:

divide and conquer principle that is used to solve complex problems by cutting down the problem into manageable pieces.

dynamic programming constructs an optimal overall solution by putting optimal partial solutions together.

greedy algorithms are similar to dynamic programming except that the solution for a subproblem may not be known. Instead it chooses the possibly best solution at the moment (a local solution). In many cases it turns out to be a globally good solution, but it may be far from good, too.

approximation algorithms only try to find an inexact but efficient solution to a problem. It differs from greedy algorithms that one can chose how close the approximation comes to the true result.

probabilistic and heuristic algorithms include some randomness or heuristic into the algorithm to make it faster at the cost of correctness. Examples are the random mutation in genetic algorithms or a random starting point for search algorithms.

On the **implementation level** an algorithm could be characterized as serial or parallel, as recursive or iterative, as deterministic or random, as exact or approximative.

A serial algorithm means that each step is executed in sequence while a parallel implementation allows to execute two or more steps in parallel. This requires a multi-processor computer or a network of computers.

Within an iterative algorithm part of the instruction sequence is repeated several times. In a recursion the algorithm refers to itself, i. e. build up a recursion stack. The classical example is the factorial function.

A deterministic algorithm always returns the same result for a certain input. In contrast, a non deterministic algorithms may return different results when executed again. This is the case for an algorithm with a “memory”. Suppose the algorithm adds the input to the last value it has computed. So calling the algorithm twice with an input value 1 will increment the last value and return $n + 1$ for the first call and $n + 2$ for the second call.

The meaning of exact and approximative applies here to the implementation or representation of data values. An algorithm for summing up a set of numbers will return the exact result only if the numbers are represented exactly. Coding numbers as floating point may result in approximate results depending on the precision of the coding. (see Chapter 10).

Our next classification deals with the efficiency of algorithms in terms of time and space requirements.

5.1.1 Time and Space Complexity

When we execute an algorithm we want to know how many instruction steps and how much storage cells are needed. Usually not the absolute values for a particular situation are relevant, but how will these values change when the input changes. In other words: we want to abstract from a particular case or hardware and see how the algorithm behaves relatively if we say double the amount of input data.

As example lets return to the sort algorithm from Chapter 2. How does the number of instructions grow with the numbers to sort? If for a certain set with n numbers we need r instructions, we want to know the instructions necessary for a set with the double size $2n$.

Look at the algorithm sketched as flow chart in Figure **??** of Chapter 2. We can identify two nested loops. If we fix a certain value k ($0 \leq k < n$) for the outer loop we have the inner loop to run n times at the worst case. As the value n does not depend on the value k the loops are independent of each other. We can estimate the total number of cycles y_n we need to run through. For both sets with n and $2n$ numbers the upper bounds are:

$$y_n = n \cdot n = n^2; \qquad y_{2n} = 2n \cdot 2n = 4n^2 \tag{5.1}$$

Let c be the number of instructions of one cycle run. This value is bound by a constant for every possible situation. It does not depend on the cardinality of number set to sort, i.e. this constant c remains unchanged for any set to sort. If we denote with r the total number of instructions to sort a set of n numbers, then

$$r = c \cdot y_n = c \cdot n^2 \tag{5.2}$$

For a set with cardinality $2n$ this leads to $z = 4r$ instructions as the following calculation shows:

$$\frac{z}{r} = \frac{c \cdot y_{2n}}{c \cdot y_n} = \frac{c \cdot (2n)^2}{c \cdot n^2} = 4 \tag{5.3}$$

From the equation 5.3 we conclude for our sorting algorithm that if we double the numbers to sort we quadruple the number of instructions to execute. This is only an upper bound because we have situations where less instructions are needed, e.g. if we don't have to exchange the numbers in a cycle or when we move a number from the original set to the sorted list.

The summation problem from Chapter 2 is quite different to the sorting algorithm. The algorithm to sum up the integers from 1 to N as in Figure **??** contains only one loop that determines the grows of instructions (corresponds with the time needed) with the number N. It is left to the reader to show that this is a linear growth.

In our examples we have made some simplifications: Only the upper bound of instructions was considered. Special input situations like an already (or partially) sorted list was not taken into account. In fact we were very cautious and considered the *worst case*. We did not really care if we need 5 or 9 instructions in one cycle as long it was limited by a fixed number c.

To generalize the situation take z_A as the time or instructions needed to process an input of n values with the algorithm A, then the formula looks like this:

$$z_A \leq const \cdot f_A(n) \quad \forall n \in \mathbf{N} \tag{5.4}$$

where f_A is a function specific for an algorithm. This function f_A is called the *complexity* of the algorithm A.

Usually we do not care about the constant $const$ and are only interested in large n. Instead to know the exact function $f_A(n)$ we prefer a well known estimate like n^2, $n \cdot \log n$ etc.

This abstraction leads us to the terms *asymptotic complexity* and *O-notation* (spoken: big-Oh notation). The term *asymptotic complexity* emphasizes that the formula 5.4 need only to be true for large values of n. The *O-notation* subsumes the constant $const$ and takes as argument well known functions like a linear,

Figure 5.1 Growth of some O-function classes

polynomial, logarithmic or exponential function. An O-function represents a *complexity class* where all functions f_A from that class are dominated by the O-function for large values n. Figure 5.1 shows a graph of how the complexity is growing for some O-functions.

Typical complexity classes are:

$O(1)$	constant function
$O(\log n)$	function with logarithmic growth
$O(n)$	function with linear growth
$O(n \log n)$	function with linear-logarithmic growth
$O(n^2)$	function with quadratic growth
$O(n^3)$	function with cubic growth
$O(k^n)$	function with exponential growth ($k > 1$)

We have so far only considered the cost in number of instruction which is roughly proportional to the time needed for an algorithm to execute. Another resource is the storage space required by an algorithm as function of the input data. This is known as the *space complexity*.

Let's return to our sorting algorithm from Chapter 2. Analyzing Figure **??** we need to sort n numbers the unsorted list *orig[n]* and the sorted list *final[n]*, plus 4 variables for controlling the loops and the number "on the side" which results in $2n+4$ storage elements. We easily see that our sorting algorithm is of O(n) space complexity because the memory required grows linearly with the numbers to sort.

Please note that the space complexity in our sorting example is less than the time complexity. This is true in general since it takes time to use space, so a problem's space complexity never grows faster than its time complexity. This is why we are mostly interested in time complexity.

It is an obvious goal to find algorithms of low complexity for both time and space. If you have lets say an algorithm of exponential complexity you can easily overstrain any computer (even future generations) if you increase the input data.

Unfortunately there are many important problems of exponential complexity. A famous one is the "traveling salesman" who wants to visit n cities with a minimum of traveling and returning to his starting point. In fact if you check every possibility this is a kind of permutation problem that results into n factorial ($n! = 1 \cdot 2 \cdot 3 \cdot \ldots n$) possibilities.

Lets give an impression of the rapidity of growth with the following table where we assume that our computer can process an instruction in one nanosecond:

n	n! / time
1	1 1 ns
2	2 2 ns
3	6 6 ns
4	24 24 ns
5	125 125 ns
6	720 0.72µs
10	3 628 800 3.6 ms
15	1 307 674 368 000 > 21 sec
20	2 432 902 008 176 640 000 > 77 years
40	815 915 283 247 897 734 345 611 269 596 115 894 272 $*10^9$ $2.58 * 10^{31}$ years
41	33 452 526 613 163 807 108 170 062 053 440 751 665 152 $*10^9$ $1.06 * 10^{33}$ years

This little demonstration makes it crystal clear that we cannot solve the traveling salesman problem that way if he has 20 or more cities to visit. In our calculation we have already assumed that our supercomputer could calculate the distance and compare the results in one nanosecond. Even next computer generations will fail to reach that speed. So there is absolutely no chance to solve 40 destinations even in the far future.

5.2 Searching

One of the most important class of algorithms deals with the searching for data. Depending on the data structure and the ordering of the data different

```
function linSearch(searchValue)
  for i := 1 to n step 1 do
    if (array[i] = searchValue) return true
  end for
  return false
end function
```

Figure 5.2 Linear search algorithm for searchData in an unsorted array

algorithms have been developed. Starting with a simple unsorted linear structure we proceed to tree structures with different search strategies.

5.2.1 Linear Search

A linear search is used to find a data item in an array or a list. To search for a particular data item we may check each element in sequence. In the case of an array we simply iterate over the array index and look at each index position for the data element. For a variable length list we may use a *do-while* loop and check the data elements one by one until we find it or reach the end of the list. Because the data elements are not sorted there is not much to do to speed up the search. We are lucky if the desired value is at the beginning of the array and in the worst case we hit the value on the end of the array. On average we have to read about half of the elements until we find the desired one. If the searched data is absent, we do not know it until all elements are read as Figure 5.2.1 shows.

As a variant of this algorithm we could return the index of the data item if we have a match instead of "true" and return -1 in the case of "value not found". This makes it easier to refer later to the data element.

As time complexity we have for each data item mainly one compare and one index increment to do, that is a constant time cost. In the best case we hit the data right at array index 1, in the worst case at index n and on average we have to read $(n + 1)/2$ entries. We conclude that the linear search algorithm yields $O(1)$ for the best case, and $O(n)$ for the average and worst cases.

If the values in the array are sorted more efficient searching is possible. No one will read the telephone book from the beginning to look for the phone number of **Peter Naur**. We could jump to the middle of the book and check whether the name there (which is **Knuth** in the example of Figure 5.3) is before or after our target name. Then we continue before resp. after the middle of the book. We continue as in Figure 5.3 until we hit the name or we find that it is not present. With this algorithm a telephone number is found much faster as with a linear search. We will generalize this concept by using tree structures in the next section.

5.3 Tree Traversal Algorithms

We recall from Chapter 4 that trees are hierarchical structures with a top level element called *root node*.

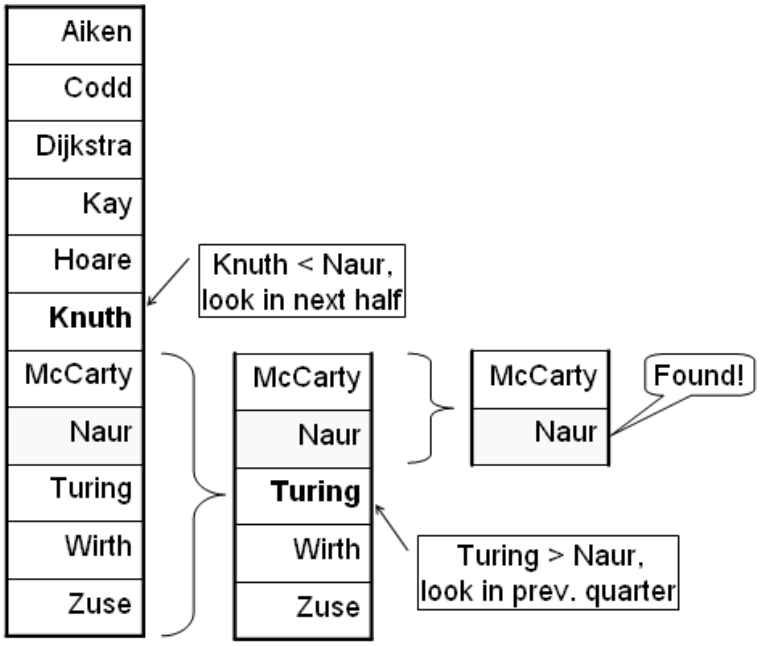

Figure 5.3 Half step searching in a sorted array (telephone searching)

Traversing a tree always starts at the root node. To reach a particular node exactly one navigation path exists. Searching for a certain node we have to traverse all branches of the tree until we find the desired node. This might be done by systematically traversing in a *pre-order* or *post-order* manner (see Section 4.4.2).

This is no benefit compared to a linear structure. But, if the nodes are ordered, very efficient algorithms exist that work similar to our telephone book search.

5.3.1 Binary Search

For this algorithm a special tree is required where each node has at most two child nodes and the nodes need to be ordered as introduced in Section 4.4.1 and 4.4.3. The two subtrees of a node represent the lower and bigger half of an ordered collection and the node itself represents the data element in the "middle".

We will now present the algorithm for the telephone search (Figure 5.3.1) in pseudo code.

Figure 5.3.1 uses the dot-notation *node.name* meaning that the person's name is extracted as string from the node. Likewise with *node.left* and *node.right* we denote the left resp. right child node.

We recall from Section 4.4.3 that the depth of a balanced binary search tree is bound by $\log_2(\#nodes)$. Apparently the depth of a tree limits the necessary steps to find a node. As the number of nodes double with each increment in depth the average number of search steps is $\log_2(\#nodes) - 1$.

```
searchName = “Turing”
node := root
function BinTreeSearchRecursive(node, searchName)
  if (node = empty) then
    return "name not found"
  else
    if (searchName = node.name) then
      return node
    else
      if (searchName < node.name) then
        return BinTreeSearchRecursive(node.left, searchName)
      else
        return BinTreeSearchRecursive(node.right, searchName)
      end if
    end if
  end if
end function
```

Figure 5.4 Searching for “Turing” in a sorted binary tree

The minimum number of nodes to visit is 1. This is the case if the root node is the one we search for.

From the previous reasoning result the complexity estimations for best, average, and worst case as $O(1)$, $\log_2(n) - 1$, and $\log_2(n)$ respectively.

The binary search algorithm traverses a tree in a unique path from the root to the destination node. In other cases like printing all node values it may be necessary to visit all nodes of a tree. This can be done systematically in two ways.

5.3.2 Breadth-First Search

Breath-first is a traversal concept that visits the nodes in order of its distance to the root node. The distance of a node is measured in number of nodes (*hops*) from the root node. First the root node itself is visited (0 hops), then all child nodes of the root are visited (1 hop). Following, all nodes with 2 hops distance to the root node are visited, i.e. the child nodes of the root’s children. This continues until no more descendants are found.

The concept works not only for trees but for any connected graph. The root node has to be replaced by a starting node. Instead of the child nodes all directly connected nodes are chosen. If a node is hit that has been already visited, we ignore this node.

Figure 5.5 shows the working steps of an example graph with 6 nodes (vertices) and 6 edges (adopted from **?**). In the example we show the nodes that have been processed completely in black, nodes visited but not investigated for continuing links with an X, and nodes not yet visited in white. The arrows indicate the direction for the next nodes to be investigated. If a node is new it is marked with an *X*. Please note that already visited nodes like the starting node in the second step are ignored.

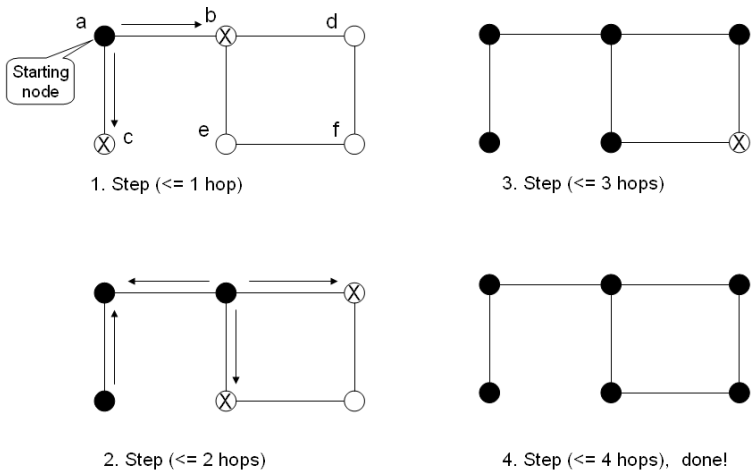

Figure 5.5 Breadth-first search example with 4 processing steps

```
function depthFirst(node)
  node.state := visited
  for n ϵ Neighbors(node) do
    if (n.state = new) then call depthFirst(n)
  end for
  node.state := processed
  return node
end function
```

Figure 5.6 Depth-first search algorithm

The breadth-first search processes all nodes only once because we ignore already visited nodes. In addition, for all edges in the graph we have to follow the links to reach the next nodes to process. The time complexity for this algorithm results in $O(|nodes| + |edges|)$

5.3.3 Depth-First Search

While the breath-first approach proceeds hop-by-hop from the starting node the depth-first search tries to follow a path as long as possible. The algorithm works recursively trough the nodes until a "dead end" is reached or there are no more unvisited nodes.

We present the algorithm in pseudo code in Figure 5.3.3.

The function *depthFirst* takes the starting node as parameter. This node is marked as visited. Then for all neighbor nodes, these are the directly connected nodes, if they have not been visited the function is called again. This is a recursive call as now all of these nodes are marked as visited and their neighbors will be used for the next call of *depthFirst*. The recursion stops if there are no more unvisited nodes and the actual node is returned. If we

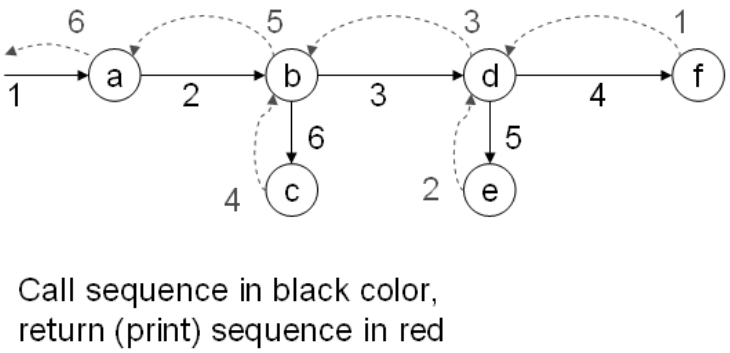

Figure 5.7 Call sequence in a depth-first search

replace the "return node" by "print node" the following list of nodes is printed for the scenario in Figure 5.3.3:

f, e, d, c, b, a

In the result sequence we assume that the *Neighbors* collection is always alphabetically in descending order. This is why the node e is returned before node d. In other words, a node is only returned if all its neighbors have been already processed and returned. The depth-first sequence is illustrated in Figure 5.7.

The time complexity of the depth-first search is again $O(|nodes| + |edges|)$. This follows from the fact that the processing of a node starts with changing its state to *visited*. This ensures that each node is only processed once. For each node the function is called for all edges going off the current nodes. This is equivalent to all directly connected nodes. In total we have processing time to be proportional to the sum of all nodes and all edges.

5.4 Sorting

Sorting is one of the most common computer processing. Take the numbers as example, they can be naturally sorted in ascending (<) or descending (>) order. This is not only necessary to present sorted lists to the user but it can also speedup the searching.

It doesn't astonish that much research effort has gone and is still going into sorting algorithms. Many books have been written that deal with this topic. The most prominent is Donald E. Knuth's work on the *Art of Computer Programming, Volume 3: Sorting and Searching* **?**.

To state more precisely what we mean by sorting a *sort order criteria* has to be chosen. There exist many possibilities to order things. If a set of persons have to be ordered, we could sort them by birthday, alphabetical by name, by weight, by height, or any other criteria. The property that is used to order objects is called a **sort order criteria**. In general, we give the sort order criteria a name or unique symbol (e.g. $\angle$) to distinguish different orderings.

Given a set of items S and an sort order criteria $\angle$. The **total order** defined by $\angle$ has the following properties:

Let $a, b, c \in S$ then
(1) a $\angle$ b or b $\angle$ a (totality)
(2) if a $\angle$ b and b $\angle$ c then a $\angle$ c (transitivity)
(3) if a $\angle$ b and b $\angle$ a then a = b (antisymmetry)

In the case of persons let's use the birthday as order criteria. Then we have the oldest person first and the youngest last.

Tom(11/21/1955) $\angle$ Pat(07/07/1975) $\angle$ Jeff(07/07/1975) $\angle$ Eve(02/22/1987)

If the sort criteria allows equality and two or more nodes are equal according to the criteria, these nodes are placed next to each other in any order.

For any sort algorithm there are two problems to solve:

- comparison according to the sort order criteria
- building the sorted collection of items

The first problem requires that any two items v_1 and v_2 are always comparable by criteria $\angle$. The total order properties above imply exactly one of three outcomes

$$v_1 \angle v_2 \quad \text{or} \quad v_1 = v_2 \quad \text{or} \quad v_2 \angle v_1$$

The second problem requires that a ordered collection is built up where all items are in $\angle$-order. This means the items have to be moved into the desired order.

Recall the *selection sort* from Chapter 2 where a list of numbers was sorted by exchanging the smallest number with the first number, then exchange the second smallest number in the list with the second number, then exchange the third smallest number in the list with the third number, etc. Finally all numbers are in ascending order.

Looking for the smallest number in a set of n numbers requires $n - 1$ comparisons. Hence, the *selection sort* algorithm requires $n - 1$ times to find the minimum in a set of numbers that decreases its cardinality by one with every iteration. The number of comparisons can be calculated as

$$(n-1) + (n-2) + (n-3) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

To bring the numbers into sorted order we have to swap numbers by setting the first number aside, moving the second number to the first's place, and finally storing the number set aside to second's place. This are three number movement operations. In sum we have $3(n-1) = O(n)$ movement operations.

Putting the cost of comparison and number movement together we get a square order complexity for the *selection sort*:

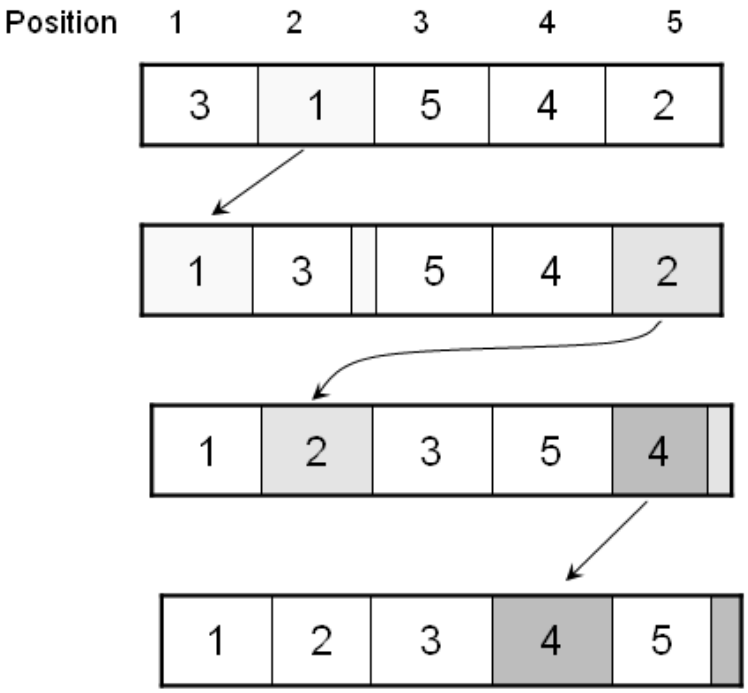

Figure 5.8 Insertion sort example

$$O(n^2) + O(n) = O(n^2)$$

We see from Figure **??** that the algorithm executes both loops with the comparisons no matter if the list of numbers is already sorted or not. This is why the complexity remains the same for best, average, and worst case.

5.4.1 Insertion Sort

It is easy to improve the selection sort algorithm that it gets a better performance for average and best cases. The basic idea for the **insertion sort** is the same as with card games where the players sort their cards by removing some cards and inserting them at the correct place until the cards on-hand are fully sorted.

Before presenting the general algorithm we show the mechanism with a little example in Figure 5.8.

The smallest number 1 is taken from the second position and inserted before number 3. Number 1 takes now position 1 and number 3 sits in position 2. The former position 2 that number 1 previously occupied is now empty and indicated in Figure 5.8 by a little gap between the new positions 2 and 3. With the next scan the next smallest number 2 is inserted behind the first position. In the third scan only the number 4 at the fifth position is out of order and it is merged between numbers 3 and 5. You may have already noticed that if the list is nearly in order, then only a few numbers need to be moved to an other place. This reduces the complexity considerably for nearly sorted lists. For a completely preordered list there is only one compare-scan needed just to

```
function InsertionSort(List)
  for i := 2 to List.size do
    j := i
    side := List(i); /* remember */
    while List(j-1) > side do
      /* move List(j-1) one position to the right */
      List(j) := List (j-1)
      j := j - 1
    end while
    List(j) := side /* remembered */
  end for
  return List
end function
```

Figure 5.9 Pseudo code for insertion sort algorithm

prove that the list is sorted. The $n - 1$ compares for the scan result in a $O(n)$ complexity. However if the list is in reverse order there are as many compares as with the selection sort resulting in a complexity of $O(n^2)$.

The general *insertion sort* algorithm in pseudo code in Figure 5.4.1 shows that for inserting a number two numbers are stored or moved at the code lines marked with /* remember(ed) */.

To make free space for inserting number x at the correct position all numbers from that position up to the position where the number x originates have to move one position to the right.

From the pseudo code we see that a list with n numbers we have two numbers stored within the for loop that runs $n - 1$) times. For sorting the list we have at least $2(n - 1) = O(n)$ store or move operations even when the list is completely pre-sorted. Our next sort algorithm will be much better in this case.

5.4.2 Bubble Sort

The **bubble sort** algorithm only swaps adjacent numbers if they are not in order. So the first run through a list L of n entries begins with comparing elements $L(n)$ and $L(n - 1)$. If $L(n) > L(n - 1)$ the two numbers are interchanged. Then elements $L(n - 1)$ and $L(n - 2)$ are compared and possibly interchanged. Continuing towards the head of the list will move the smallest number to the first position. With the second run the second smallest number goes to the second position. With each following run the sorted part of the list grows at least by one element. The algorithm terminates when no more elements are swapped and hence the list is sorted.

If you think of the list of numbers running top-down. Then with each compare scan it looks like the smallest number bubbles to the top of the list as in Figure 5.10. From this observation the algorithm got its name.

The pseudo code for doing a bubble sort is very compact. It consists of two nested loops. The inner loop runs over the list to sort so that each adjacent

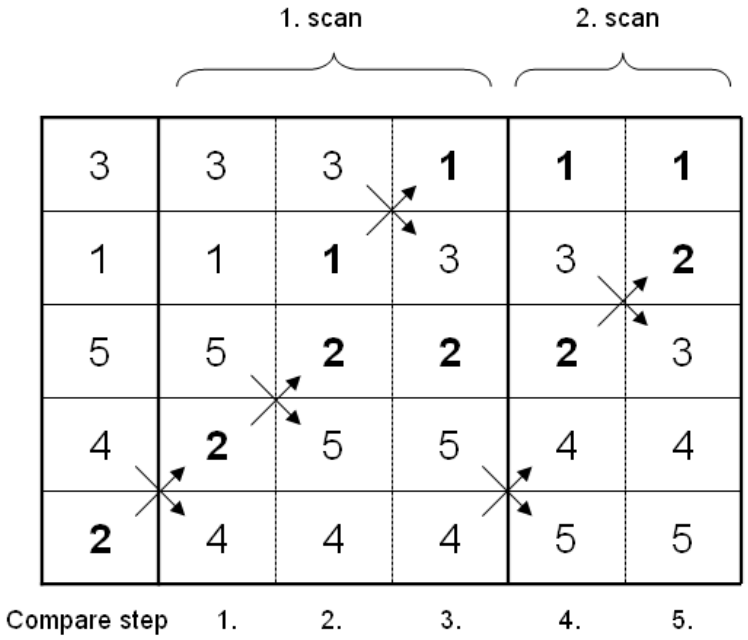

Figure 5.10 Bubble sort example

```
function BubbleSort(List)
  do
    for i := 1 to n-1 do
      if List(i) > List(i+1) then
        /* swap List(i) and List(i+1) */
        temp := List(i)
        List(i) := List(i+1)
        List(i+1) := temp
      end if
    end for
  until no more swaps occur
  return List
end function
```

Figure 5.11 Pseudo code for bubble sort algorithm

number pair is compared and swapped if they are not in order. The outer loop ensures that this is done until no more number swaps occur in the list.

Using the pseudo code from Figure 5.4.2 to estimate the complexity for best and worst case is simple. We mainly have to consider the cost for comparison and swapping of numbers. If the numbers are already in sorted order, there is only one compare scan to do and no number is moved. Hence we have for the best case $n - 1$ compare and no (0) move operations which leads to

$$(n-1)+0 = O(n)$$

In the worst case when the list in sorted in reverse order for each comparison the numbers must be swapped. The cost for swapping two numbers are 3 store operations. Initially in a reverse ordered list the smallest number is at the tail. Then with each compare scan over the list (this is the *for-loop*) the number moves one position towards the head. After n compare scans it reaches its final position at the head of the list. The cost for this bubble movement from the tail to the head is $n-1$ compare and $3n$ store operations. Now the bubble movement has to be repeated for the second smallest number which is now at the tail of the list. But now only $n-2$ number swaps are necessary because the smallest number is already at its place and the second smallest goes behind it.

To move all the remaining numbers to their final position the *do-until* loop has to be repeated n times. With the i^{th} run only $n-i$ swaps are necessary. The total cost for the worst case is n^2 compare and $3(n-i)$ store operations for each run of the *for-loop*.

$$n^2 + \sum_{i=1}^{n} 3(n-i) = n^2 + n(n-1)/2 = O(n^2)$$

5.4.3 Quick Sort

quick sort is based on the divide and conquer principle (see Section 7.1). A pivot element p is chosen from the list L to divide the list into two partitions, the lower part T and the upper part U. The pivot element may be chosen arbitrarily but there exist a lot of proposals for good choices in order to improve the performance of the algorithm.

Now numbers in the lower partition T that are bigger than the pivot element p are swapped with numbers in the upper partition U that are smaller than the pivot element. After this all elements in T are less or equal than p and all elements in U are greater or equal p.

The same partitioning and swapping of element is now repeated for each partition until the partitions contain only one element. The list is now sorted as illustrated in Figure 5.12.

More formally the algorithm is described in Figure 5.4.3 as pseudo code consisting of two functions, *QuickSort()* and *Partition()*. Partition takes the list to sort and lower and upper bound for the partitioning. It does not only choose a pivot element but exchanges numbers so that the following relations hold:

$$t \leq p \leq u \text{ for all } t \in T \text{ and } u \in U$$

The function *QuickSort()* makes sure that function *Partition()* is called as long as the upper bound is greater than the lower bound. The first call has as lower bound the value 1 and as upper bound the value n for a list L of n elements.

The partition algorithm shown in 5.4.3 is only one of many possible realizations. Many ideas and heuristics have been proposed to choose the pivot element. Other issues have been the behavior when an element is found that is equal to the pivot, or iterative variants. The direction to swap numbers may alternate from left to right.

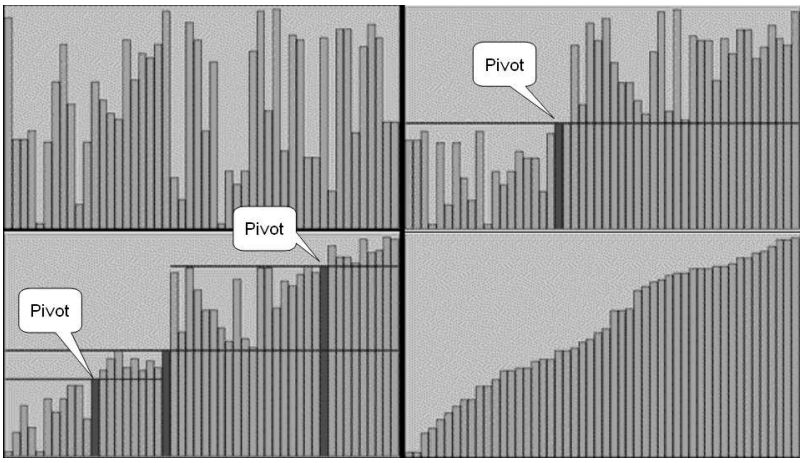

Figure 5.12 Quick sort example (graph taken from Wikipedia ?)

```
function QuickSort(List, t, u)
  if u > t then
    i := Partition(List, t, u)
    QuickSort(List, t, i-1)
    QuickSort(List, i+1, u)
  end if
  return List
end function

function Partition(List, t, u)
  p := (t+u)/2
  while t ≤ u do
    l ∈ [t ... p − 1]withList(l) ≥ List(p) /* c */
    r ∈ [p + 1 ... u]withList(r) < List(p) /* c */
    if l ≤ r then
      swap L(l) and L(r)
      t := l + 1;
      u := r - 1;
    end if
  end while
  return u
end function
```

Figure 5.13 Pseudo code for quick sort algorithm

Now let's turn to the complexity of the quick sort algorithm. For the worst case the number of comparison and move or store operations will be counted.

To move the numbers to the correct partition all elements in the list are compared with the pivot element, i.e $n - 1$ comparisons. This is done in lines commented with /* c */ where we raise the lower index and reduce the upper

index for the elements to compare. The next call to *QuickSort* contains not more than $n-1$ elements that are compared to the new pivot element, i.e. $n-2$ comparisons. Each subsequent call reduces the partition by at least one element. In the worst case we have $\sum_{i=1}^{n-1} i$ comparisons and the same number of swap operations leading to

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

In the best case two equal size partitions are created by the pivot element. This happens when the pivot element is also the median of the list. In this case the recursive call to *QuickSort* halves the number of elements to partition. For a list with n elements after $\log(n)$ calls the upper and lower bounds are the same and the sorting is completed. We know already that for n elements $n-1$ comparisons are needed to partition the list. We get as best case complexity

$$(n-1) + (n-1)/2 + (n-1)/4 + \ldots + 1 < n\log(n) = O(n\log(n))$$

After laborious computations and estimations it turns out that on average the run time for the quick sort has the same complexity as for the best case. This means that for most practical situation this algorithm work a lot better than the insertion or bubble sort. In a worst case scenario the cost reaches square order complexity.

It can be shown that a general sorting algorithm has at least $O(n\log(n))$ complexity.

5.4.4 Heap Sort

The **heap sort** algorithm is an example for such an optimal algorithm where the complexity reaches the theoretical limit of $O(n\log(n))$

The algorithm follows the selection principle but does this in a very smart way. It uses a *heap data structure* (see Section 4.4.4 to find the maximum of n values with a single operation. To have a set sorted, build a heap of it and remove successively the root element until the heap is empty.

The heap sort algorithm is trivial if we rely on a heap with a built-in "heapify" function. The tough work is in rebuilding the heap as we have shown in the previous chapter.

We give a pseudo code in Figure 5.4.4 for the heap sort assuming that the heap supports at least the *sift_down* procedure. The list to sort is reused for building up the heap structure by taking element by element and let it sift down in the heap. When the heap is completed it will be returned in sorted order.

The computing costs for the heap sort consist mainly of building the heap and retrieving the nodes from the root of the heap. In a binary tree structure the cost for sifting a value in the heap is proportional to the height of the tree. So, the cost for building and reading a heap of n nodes requires

$$const \cdot 2n\log(n) = O(n\log(n))$$

instructions.

```
function HeapSort(List)
  l := n /* List has n elements */
  while l > 1 do /* transform List to Heap
    swap List(1) and List(l)
    sift_down(List(1) in List[1. . . l-1]) /* is Heap now*/
    l := l-1
  end while
  return List/Heap in sorted order
end function
```

Figure 5.14 Pseudo code for heap sort algorithm

5.5 Summary

In this chapter we have classified algorithms in different dimensions, namely in terms of its computing time. We abstracted from real computer models by calculating only the relative time needed when the input size changes. Basically we make performance statement like these: If the input is doubled the processing time doubles or quadruples. The prediction is only on the processing time behavior in magnitude for large data sets, called time complexity.

The basic searching and sorting algorithms have been explained in pseudo code. For each algorithm the concept was analyzed with regard to complexity and a small example showed how it works.

Exercises

5.1 Point out the differences of the selection and insertion sort.

5.2 When is an algorithm of exponential complexity tolerable?

5.3 Can you rewrite the pseudo code for the bubble sort if we want to sort in descending order?

5.4 Try to find an animated visualization of the Quick-sort on the WWW.

Bibliographical Notes

There is a rich choice of books on algorithms. Nice visualizations of algorithms are found in the book *Introduction to Algorithms* of **?**.

A lot of descriptive illustrations and animations of the quick sort are found in the WWW. A good point to start with is the Wikipedia article on QuickSort **?**. There are links to reference literature, implementations, variants, and nicely illustrated examples and animation on how the algorithm works.