Chapter 2

Complexity, optimisation and debugging

2.1 Complexity

The *complexity*, or *efficiency* of an algorithm is a measure of how much resources (*i.e.*, time and memory) an algorithm takes to produce an answer. This is typically dependent on the problem size N. For example, if N is large, it will take longer to determine if the integer N is prime or to sort an array of N integers. However, the complexity depends not only on the problem size, but also at the exact inital conditions. For example, the sorting algorithm might be very quick if the initial array is already sorted, but will take a long time for randomized arrays. Equally, a prime-detection algorithm should take longer to show that 523 is prime (it has to do many checks), than to show that 10^{18} is not prime (as it is divisible by 2). Therefore, the complexity is typically a worst-case analysis (how long can it possibly take to sort N integers, regardless of the initial ordering). Further, since efficiency of an algorithm matters most for large problem sizes, we restrict the analysis to large $N \gg 1$. Thus, we want to answer the following questions:

1. Time complexity: how many elementary operations/ how much computing time does the algorithm take in the worst case to solve the problem as a function of the problem size $N \gg 1$?

2. Memory complexity: how much memory does your algorithm take …?

Since the exact computing time is very difficult to determine and hardware dependent, we only measure the *order of complexity*; this only takes into account the largest contribution as a function of the problem size N: *E.g.*, if the computing time of an algorithm equals $c = 2N^2 + 100$, the order of complexity will be $\mathcal{O}(N^2)$. If the computing time of a second algorithm equals $c = 14000N + 1000\log(N)$, the order of complexity will be $\mathcal{O}(N)$.

This, however, is only useful to estimate the computing costs for very large N: Due to the large coefficients, the second algorithm will only be quicker than the first algorithm if $N > 7000$!

2.1.1 Example: Finding the greatest common denominator

As a first example, we calculate the complexity of Algorithm 1, which determines the greatest common divisor (GCD) of two integers m and n.

Algorithm 1 A simple algorithm for finding the GCD

```
Input: $n, m \in \mathbb{N}$
Output: largest $z \in \mathbb{N}$ s.t. $\mathrm{mod}(n, z) = 0$ and $\mathrm{mod}(m, z) = 0$
  for $k = 1$ to $\min(m, n)$ do
    if $\mathrm{mod}(n, k) = 0$ and $\mathrm{mod}(m, k) = 0$ then
      $z \leftarrow k$
    end if
  end for
```

Here, $\mathrm{mod}(n, z)$ denotes the remainder of the long integer division, *e.g.* $\mathrm{mod}(13,\ 5) = 3$ since $13 = 2 \cdot 5 + 3$. The algorithm takes all numbers k from 1 to $\min(m, n)$ and checks if k divides both m and n. If it finds such a common divisor, it assigns its value to z. As it loops over k in increasing order, the value of z will be the largest common divisor, the GCD. Note that, since 1 is a common divisor for any pair (m, n), the GCD is always defined.

To calculate the complexity, note that each loop consists of four operations, $\mathrm{mod}(n, k) = 0$, $\mathrm{mod}(m, k) = 0$, the *and* operator and $z \leftarrow k$.[1] The loop is repeated $\min(m, n)$ times, thus the time complexity of the algorithm is $4N$, with $N = \min(m, n)$ representing the problem size. As only four variables are used (m, n, k, z), the memory complexity is 4. Thus, *the complexity of Algorithm 1 is of order $\mathcal{O}(N)$ in time and $\mathcal{O}(1)$ in memory.*

The speed of this algorithm can be improved: For example, since we are looking for the greatest common divisor, one can start to search through all numbers 1 to $\min(m, n)$ *backwards* and stop once the first common divisor (and therefore the greatest one) is found. This is shown in Algorithm 2, which is faster, as the loop is repeated N times only in the worst case, when the GCD is 1. However, the order of complexity of Algorithm 2 is not better than for Algorithm 1, as it is determined by the worst case.

---

[1] The exact number of elementary operations is different for a complex function such as $\mathrm{mod}(n, k) = 0$ and a simple assignment such as $z \leftarrow k$. However, since we are only interested in the order of complexity, it is sufficient to know that the amount of operations is independent of N.

Algorithm 2 An improved algorithm for finding the GCD

```
Input: $n, m \in \mathbb{N}$
Output: largest $z \geq 1$ s.t. $\mathrm{mod}(n, z) = 0$ and $\mathrm{mod}(m, z) = 0$
  for $k = \min(m, n)$ downto 1 do
    if $\mathrm{mod}(n, k) = 0$ and $\mathrm{mod}(m, k) = 0$ then
      return $z \leftarrow k$
    end if
  end for
```

Using a very different approach, we will now show that even the order of complexity of an algorithm can be improved. Algorithm 3, also known as Euclid's algorithm,[2] takes into account that each common divisor (therefore also the greatest one) of m and n is also a common divisor of $mod(m, n)$ and n.[3] Therefore, we can reduce the problem size by replacing the pair (m, n) by $(n, \mathrm{mod}(m, n))$ recursively until we reach a problem that is easy to solve. This operation only reduces the problem size if $m > n$. Note, however, that the first iteration will switch the two values if the initial data is not in increasing order. The algorithm can be ended once $\mathrm{mod}(m, n) = 0$ as this means that n is a common divisor (and therefore the greatest one). For example, starting the algorithm with $(m, n) = (105, 252)$ yields $(105, 252) \to (252, 105) \to (105, 42) \to (42, 21) \to (21, 0)$, therefore $GCD(252, 105) = 21$.

Algorithm 3 Euclid's algorithm

```
Input: $n, m \in \mathbb{N}$
Output: largest $z \geq 1$ s.t. $\mathrm{mod}(n, z) = 0$ and $\mathrm{mod}(m, z) = 0$
  while $n \neq 0$ do
    $k \leftarrow mod(m, n)$
    $m \leftarrow n$
    $n \leftarrow k$
  end while
  return $z \leftarrow n$
```

The complexity of Algorithm 3 is of order $\mathcal{O}(logN)$ in time and $\mathcal{O}(1)$ in memory, and therefore much faster than the previous two algorithms. To prove this, set $k = \mathrm{mod}(m, n)$ and $m > n$. Then $m \geq n > k \geq 0$ and $m \geq n + k > 2k$. Therefore, $(m \cdot n)/2 > n \cdot k$, thus the product of the two numbers will more than half with each iteration. After at most two iterations, the product of the two numbers is less than $N^2 = \min(m, n)^2$. Therefore, it will be smaller than 1 in at most $\log_2(N^2) + 3 = 2\log_2(N) + 3$ iterations. Each iteration consists of elementary operations, therefore the algorithm takes $\mathcal{O}(\log N)$ operations.

---

[2] T. L. Heath, *The Thirteen Books of Euclid's Elements*, Cambridge Univ. Press, 1925

[3] If $m = a \cdot k$, $n = b \cdot k$, $m = c \cdot n + mod(m, n)$, then $mod(m, n) = (b - a \cdot c)k$

Table 2.1: Complexity of an algorithm. For which N can we solve a problem which requires $f(N)\mu$s computing time?

$f(N)$	1 second	1 month	1 century
$\ln N$	10^{434294}	$10^{1125691290709}$	$10^{1369591078130094}$
N	10^{6}	10^{12}	10^{15}
$N \ln N$	10^{5}	10^{11}	10^{14}
N^2	10^{3}	10^{6}	10^{7}
2^N	19	41	51
$N!$	9	14	17

2.1.2 Interpreting the order of complexity

While the order of complexity cannot predict the actual speed of your algorithm, it is a good measure to estimate the problem size one can expect to solve in a reasonable time. Table 2.1 shows the time it takes to solve a problem of given complexity. While an algorithm of complexity $\mathcal{O}(\log N)$ can be used for almost arbitrary problem sizes, an algorithm of polynomial order, $\mathcal{O}(N^\alpha)$, can be used for moderate problem sizes. An algorithm of higher complexity such as $\mathcal{O}(N!)$ or $\mathcal{O}(2^N)$, can only be solved in a reasonable time for very small problem sizes. The goal of algorithm design therefore is to make the complexity as small as possible. Here are a few examples of the complexity of the well-known algorithms for basic mathematical problems:

- Find all prime numbers from 1 to N, using the Sieve of Erathostenes: Time complexity $\mathcal{O}(N \log(\log N))$, memory complexity $\mathcal{O}(N)$.[4]
- Find all permutations of the numbers $1, \ldots, N$ using the MATLAB function `perms`: Time and memory complexity $\mathcal{O}(N!N)$.

You can test the time complexity of an algorithm using the MATLAB command `cputime`. This is illustrated in Algorithm 4. There, a set of problem sizes N is chosen and the computing time t is calculated for each value of N. Because the computing time cannot be accurately measured for small N and we want to finish the test in a reasonable time, we pick the range of problem sizes N such that the computing time is not too small and not too large (between approx. $0.01s$ and $60s$). If we do not know the time complexity beforehand, we can also plot t against N in log-log scale; if the result is a straight line, its slope corresponds to the polynomial order of complexity. If the curve is convex (e.g. curved like a parabola), the algorithm has a higher than polynomial order and will be impractical for large problem sizes.

---

[4] `http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

Algorithm 4 script to test the time complexity of the `primes` algorithm

```
% chose a set of medium—to—large problem sizes N
N=logspace(5.5,8,30);
for i=1:length(N)
    % calculate the computing time for each value of N
    tinit = cputime;
    p = primes(N(i));
    t(i)=cputime—tinit;
end
% plot the computing time t scaled by the expected order
% of complexity (should be constant for large N)
loglog(N,t./(N.*log(log(N))),'.—')
```

2.1.3 Example: Finding the running average

For many basic problems, algorithms of varying complexity exist. For example, Algorithms 5 and 6 both compute the running average $A_k = \sum_{i=1}^{k} x_i$, $k = 1, \dots, N$ from the input values x_k. However, Algorithm 5 requires a pair of nested loops, yielding a time complexity of $c \approx \sum_{k=1}^{N} k = \frac{1}{2}N(N-1) = O(N^2)$. Algorithm 6 is much simpler, with a time complexity of $c = O(N)$.

Algorithm 5 Calculating the running average

```
Input: $x_k \in \mathbb{R}$, $k = 1 \dots, N$
Output: $A_k \in \mathbb{R}$, $k = 1 \dots, N$, denoting the average value of $x_1, \dots, x_k$
  for $k = 1$ to $N$ do
    $A_k \leftarrow \text{Average}(x_1, \dots, x_k)$
  end for

Function Average
Input: $x_k \in \mathbb{R}$, $k = 1 \dots, N$
Output: $A \in \mathbb{R}$ denoting the average value of $x_1, \dots, x_k$
  $a \leftarrow 0$
  for $m = 1$ to $k$ do
    $a \leftarrow a + x_m$
  end for
  $A \leftarrow a/k$
```

Algorithm 6 An improved algorithm for the running average

```
Input: $\vec{x} \in \mathbb{R}^N$
Output: $\vec{A} \in \mathbb{R}^N$, where $A_k$ denotes the average value of $x_1, \dots, x_k$
  $s \leftarrow 0$
  for $k = 1$ to $N$ do
    $s \leftarrow s + x_k$
    $A_k = s/k$
  end for
```

2.1.4 Compute complexity faster

We have determined the time complexity of the nested loop in Algorithm 5 using the summing formula $c = \sum_{k=1}^{N} k = \frac{1}{2}N(N-1) \approx \frac{1}{2}N^2 = O(N^2)$.

While the sum required here is well-known, it is sometimes more difficult to determine sums such as $\sum_{k=1}^{N} f(k)$. Note, however, that

$$\sum_{k=1}^{N} f(k) \approx \int\limits_{0}^{N} f(x)\,\mathrm{d}x.$$

This is illustrated in Figure 2.1. Therefore, we can estimate the time complexity with an integral:

$$c \approx \int\limits_{0}^{N} x\,\mathrm{d}x = \frac{1}{2}N^2 = O(N^2).$$

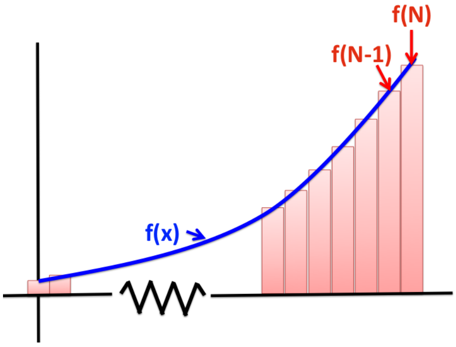

Figure 2.1: Computing complexity faster using an integral

2.1.5 Example: Sorting

A common problem in many programs is how to store sorted data. This can be done using different types of data structures:

- arrays $a_1, a_2, \ldots, a_N$
- trees (discussed later)
- linked list (discussed in §4.3)

Next, we will discuss two algorithms, one using arrays and one using a tree.

Sorting with arrays: list sort

The basic algorithm for sorting a list is given in Algorithm 7. A sorted list is created by starting with the first element, and then adding iteratively all other elements by a) finding the correct position for an element and b) inserting it. Since the `findposition` and `insert` sub-algorithms are used iteratively, their complexity will ultimately determine the complexity of the sorting algorithm.

Algorithm 7 Sorting algorithm using arrays

```
Input: list $\{a\}$ of length $N \in \mathbb{N}$
Output: list $\{b\}$ but now sorted
  $b_1 \leftarrow a_1$
  for $q = 2$ to $N$ do
    $p$ = findposition($a_q$, $b(1:q)$)
    insert($a_q$, $p$, $b(1:q)$)
  end for
```

An exemplary algorithm for the `findposition` and `insert` routines are given in Algorithms 8 and 9. Here, we assume that the first $q-1$ values have already been sorted and the q-th value has to be inserted. The position is found using a bisection algorithm where the position is found by iteratively splitting the already sorted array in half and deciding in which half the insertion has to take place. Then, all values after the insertion position are shifted one index further back, leaving a gap into which the new element is inserted.

The complexities of the algorithms are $\mathcal{O}(\log q)$ for `findposition` and $\mathcal{O}(q)$ for `insert` (considering the worst case), thus the insert algorithm is the more expensive part and determines the complexity of the search algorithm, which is therefore $\mathcal{O}(\sum_{q=1}^{N} q) = \mathcal{O}(N^2)$.

Algorithm 8 bisection algorithm for `findposition`

```
Input: list $\{b\}$ in the range $l..r$
Output: position $k$ of $x$ in that list
  $m = (l + r)/2$
  if $x < a_m$ then
    $k =$ findposition($x$, $b(l..m)$)
  else
    $k =$ findposition($x$, $b(m..r)$)
  end if
```

Algorithm 9 Algorithm for `insert`

```
Input: list $\{b\}$ in the range $1 : q$
Output: add $x$ at position $k$
  for $k = q$ downto $p + 1$ do
    $\_(k) = b(k - 1)$
  end for
  $b(p) = x$
```

Sorting with trees: heap sort

Next, we show that the complexity of searching algorithms can be improved using trees. An example of a tree structure is shown in Figure 2.2. A tree has nodes, which contain the data and which are connected by branches to other nodes. Each node is connected to a varaible number of child nodes and one parent node; only the root node has no parent. The distance from the root (i.e. the amount of branches between the root and the node) is called the depth, or level, of the node. Examples of tree structures are books, which are structures into chapters, sections, subsections and paragraphs, or the directory structure on the harddrive of your computer.

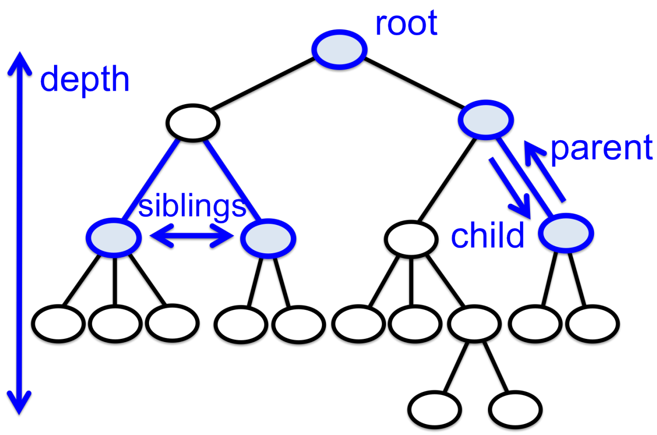

Figure 2.2: Structure and nomenclature of a tree

A tree-based sorting algorithm will not create a sorted array of values, but will store the array in a tree which is designed such that it is cheap to insert new elements and to extract the elements in a sorted way, *i.e.*, to extract the smallest element. This is illustrated in Algorithm 10. We will show that each insertion and extraction into a tree of q elements can be done in $\mathcal{O}(\log q)$ time, and hence the total algorithm is $O(N \ln N)$.

Algorithm 10 structure of the `heapsort` algorithm

```
Input: list $\{a\}$ of length $N \in \mathbb{N}$
Output: list $\{b\}$ but now sorted
  put all elements of $\{a\}$ in a (special) tree
  extract elements out of this tree in a sorted way $\rightarrow \{b\}$
```

We will use a *full binary tree*, as shown in Figure 2.3. In a *binary* tree, each node has either two children or none. In a *full* binary tree, each node has two children, except the rightmost nodes on the second-lowest depth and the nodes on the lowest depth, which have no children. This creates a tree which has a unique mapping onto a linear array and thus can be stored without the use of pointers to represent the branches.

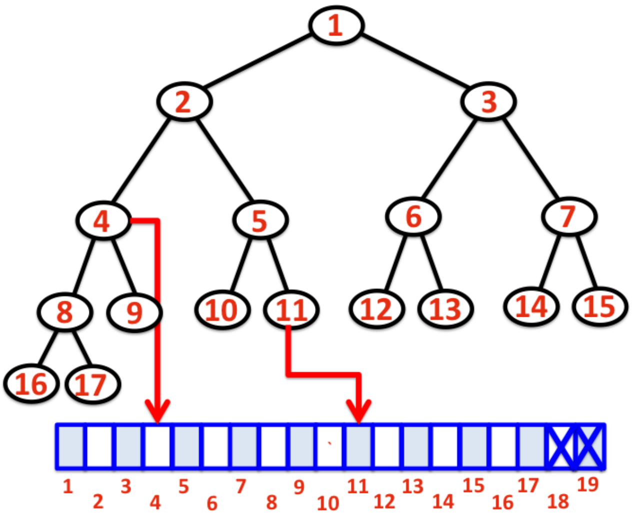

Figure 2.3: A full binary tree and the mapping to a linear array

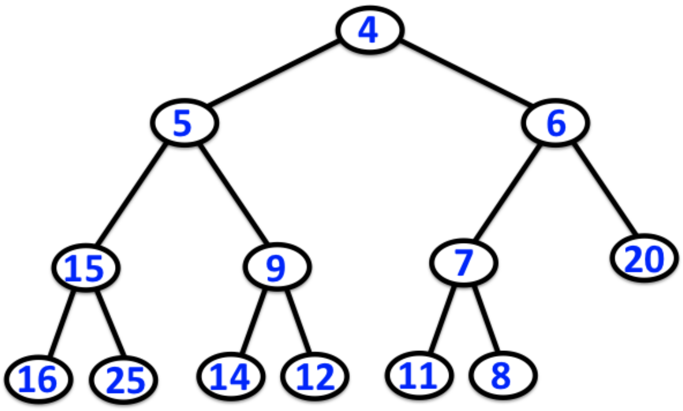

Figure 2.4: Data stored as a heap in a full binary tree

The data will be stored as a *heap*, as shown in Figure 2.4. In a heap tree, the parent is smaller than both children; hence the root must contain the smallest item. We have to ensure that the tree remains a heap tree after each insertion and extraction. The insertion process is shown in Figure 2.5: a number is inserted at the end of the tree. Then the tree is reordered by iteratively comparing the new element with its parent and switching the two nodes if the parent is larger. The extraction process is shown in Figure 2.6: the smallest number is extracted at the root of the tree and the last element is moved to the root. Then the tree is reordered by iteratively comparing the new root element with the smaller of the children and switching the two nodes if the child is smaller. Both these processes will ensure that the tree remains a heap. Since the depth of a tree with q nodes is $\lceil \log q \rceil$, there are at most $\sum_{q=1}^{N} \lceil \log q \rceil \approx \int_0^N \log x \, dx$ switches and comparisons required to insert and extract all elements. Therefore, the time complexity of the sorting algorithm is $O(N \ln N)$.

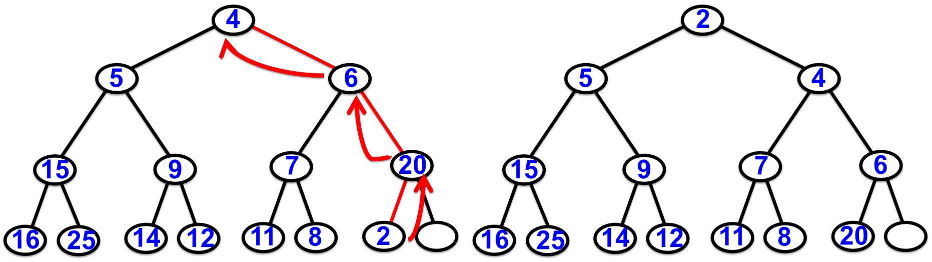

Figure 2.5: Inserting the number 2 into a heap tree and reordering; process (left) and final state (right)

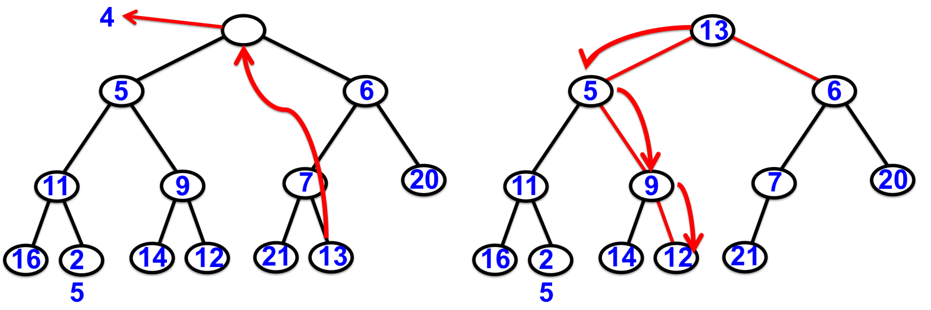

Figure 2.6: Extracting the smallest number (4) from a heap tree and reordering; extraction (left) and reordering (right)

While the complexity of this sorting algorithm is very low, the disadvantage is that this method *always* takes $c = O(N \ln N)$ even if the original data is sorted. Other sorting algorithms, like bubblesort, require $O(N^2)$ operations in the worst case, but can run in only $O(N)$ operations if the data is "almost" sorted. Hence, in practice, you should choose a sorting algoithm depending on your data.