0

1. Introduction

Let us start with two examples.

A company has a machine which drills holes into printed circuit boards. Since it produces many of these boards it wants the machine to complete one board as fast as possible. We cannot optimize the drilling time but we can try to minimize the time the machine needs to move from one point to another. Usually drilling machines can move in two directions: the table moves horizontally while the drilling arm moves vertically. Since both movements can be done simultaneously, the time needed to adjust the machine from one position to another is proportional to the maximum of the horizontal and the vertical distance. This is often called the L_∞-distance. (Older machines can only move either horizontally or vertically at a time; in this case the adjusting time is proportional to the L_1-distance, the sum of the horizontal and the vertical distance.)

An optimum drilling path is given by an ordering of the hole positions $p_1, \ldots, p_n$ such that $\sum_{i=1}^{n-1} d(p_i, p_{i+1})$ is minimum, where d is the L_∞-distance: for two points $p = (x, y)$ and $p' = (x', y')$ in the plane we write $d(p, p') := \max\{|x - x'|, |y - y'|\}$. An order of the holes can be represented by a permutation, i.e. a bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.

Which permutation is best of course depends on the hole positions; for each list of hole positions we have a different problem instance. We say that one instance of our problem is a list of points in the plane, i.e. the coordinates of the holes to be drilled. Then the problem can be stated formally as follows:

DRILLING PROBLEM

Instance: A set of points $p_1, \ldots, p_n \in \mathbb{R}^2$.

Task: Find a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $\sum_{i=1}^{n-1} d(p_{\pi(i)}, p_{\pi(i+1)})$ is minimum.

We now explain our second example. We have a set of jobs to be done, each having a specified processing time. Each job can be done by a subset of the employees, and we assume that all employees who can do a job are equally efficient. Several employess can contribute to the same job at the same time, and one employee can contribute to several jobs (but not at the same time). The objective is to get all jobs done as early as possible.

In this model it suffices to prescribe for each employee how long he or she should work on which job. The order in which the employees carry out their jobs is not important, since the time when all jobs are done obviously depends only on the maximum total working time we have assigned to one employee. Hence we have to solve the following problem:

JOB ASSIGNMENT PROBLEM	
Instance:	A set of numbers $t_1, \ldots, t_n \in \mathbb{R}_+$ (the processing times for n jobs), a number $m \in \mathbb{N}$ of employees, and a nonempty subset $S_i \subseteq \{1, \ldots, m\}$ of employees for each job $i \in \{1, \ldots, n\}$.
Task:	Find numbers $x_{ij} \in \mathbb{R}_+$ for all $i = 1, \ldots, n$ and $j \in S_i$ such that $\sum_{j \in S_i} x_{ij} = t_i$ for $i = 1, \ldots, n$ and $\max_{j \in \{1,\ldots,m\}} \sum_{i: j \in S_i} x_{ij}$ is minimum.

These are two typical problems arising in combinatorial optimization. How to model a practical problem as an abstract combinatorial optimization problem is not described in this book; indeed there is no general recipe for this task. Besides giving a precise formulation of the input and the desired output it is often important to ignore irrelevant components (e.g. the drilling time which cannot be optimized or the order in which the employees carry out their jobs).

Of course we are not interested in a solution to a particular drilling problem or job assignment problem in some company, but rather we are looking for a way how to solve all problems of these types. We first consider the DRILLING PROBLEM.

1.1 Enumeration

How can a solution to the DRILLING PROBLEM look like? There are infinitely many instances (finite sets of points in the plane), so we cannot list an optimum permutation for each instance. Instead, what we look for is an algorithm which, given an instance, computes an optimum solution. Such an algorithm exists: Given a set of n points, just try all possible $n!$ orders, and for each compute the L_∞-length of the corresponding path.

There are different ways of formulating an algorithm, differing mostly in the level of detail and the formal language they use. We certainly would not accept the following as an algorithm: "Given a set of n points, find an optimum path and output it." It is not specified at all how to find the optimum solution. The above suggestion to enumerate all possible $n!$ orders is more useful, but still it is not clear how to enumerate all the orders. Here is one possible way:

We enumerate all n-tuples of numbers $1, \ldots, n$, i.e. all n^n vectors of $\{1, \ldots, n\}^n$. This can be done similarly to counting: we start with $(1, \ldots, 1, 1)$, $(1, \ldots, 1, 2)$ up to $(1, \ldots, 1, n)$ then switch to $(1, \ldots, 1, 2, 1)$, and so on. At each step we increment the last entry unless it is already n, in which case we go back to the last entry that is smaller than n, increment it and set all subsequent entries to 1.

This technique is sometimes called backtracking. The order in which the vectors of $\{1, \ldots, n\}^n$ are enumerated is called the lexicographical order:

Definition 1.1. *Let $x, y \in \mathbb{R}^n$ be two vectors. We say that a vector x is* **lexicographically smaller** *than y if there exists an index $j \in \{1, \ldots, n\}$ such that $x_i = y_i$ for $i = 1, \ldots, j-1$ and $x_j < y_j$.*

Knowing how to enumerate all vectors of $\{1, \ldots, n\}^n$ we can simply check for each vector whether its entries are pairwise distinct and, if so, whether the path represented by this vector is shorter than the best path encountered so far.

Since this algorithm enumerates n^n vectors it will take at least n^n steps (in fact, even more). This is not best possible. There are only $n!$ permutations of $\{1, \ldots, n\}$, and $n!$ is significantly smaller than n^n. (By Stirling's formula $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$.) We shall show how to enumerate all paths in approximately $n^2 \cdot n!$ steps. Consider the following algorithm which enumerates all permutations in lexicographical order:

PATH ENUMERATION ALGORITHM

Input: A natural number $n \geq 3$. A set $\{p_1, \ldots, p_n\}$ of points in the plane.

Output: A permutation $\pi^* : \{1, \ldots, n\} \to \{1, \ldots, n\}$ with $cost(\pi^*) := \sum_{i=1}^{n-1} d(p_{\pi^*(i)}, p_{\pi^*(i+1)})$ minimum.

① Set $\pi(i) := i$ and $\pi^*(i) := i$ for $i = 1, \ldots, n$. Set $i := n - 1$.

② Let $k := \min(\{\pi(i) + 1, \ldots, n + 1\} \setminus \{\pi(1), \ldots, \pi(i-1)\})$.

③ **If** $k \leq n$ **then:**
Set $\pi(i) := k$.
If $i = n$ and $cost(\pi) < cost(\pi^*)$ **then** set $\pi^* := \pi$.
If $i < n$ **then** set $\pi(i+1) := 0$ and $i := i + 1$.
If $k = n + 1$ **then** set $i := i - 1$.
If $i \geq 1$ **then go to** ②.

Initialization

k= Next available number for position i

Permutation finished. Compute cost

Permutation not finished. Go forward

No more available numbers. Go one position backwards

Starting with $(\pi(i))_{i=1,\ldots,n} = (1, 2, 3, \ldots, n-1, n)$ and $i = n-1$, the algorithm finds at each step the next possible value of $\pi(i)$ (not using $\pi(1), \ldots, \pi(i-1)$). If there is no more possibility for $\pi(i)$ (i.e. $k = n + 1$), then the algorithm decrements i (backtracking). Otherwise it sets $\pi(i)$ to the new value. If $i = n$, the new permutation is evaluated, otherwise the algorithm will try all possible values for $\pi(i+1), \ldots, \pi(n)$ and starts by setting $\pi(i+1) := 0$ and incrementing i.

So all permutation vectors $(\pi(1), \ldots, \pi(n))$ are generated in lexicographical order. For example, the first iterations in the case $n = 6$ are shown below:

	$\pi := (1,2,3,4,5,6)$,	$i := 5$	
$k := 6$,	$\pi := (1,2,3,4,6,0)$,	$i := 6$	
$k := 5$,	$\pi := (1,2,3,4,6,5)$,		$cost(\pi) < cost(\pi^*)$?
$k := 7$,		$i := 5$	
$k := 7$,		$i := 4$	
$k := 5$,	$\pi := (1,2,3,5,0,5)$,	$i := 5$	
$k := 4$,	$\pi := (1,2,3,5,4,0)$,	$i := 6$	
$k := 6$,	$\pi := (1,2,3,5,4,6)$,		$cost(\pi) < cost(\pi^*)$?

Since the algorithm compares the cost of each path to π^*, the best path encountered so far, it indeed outputs the optimum path. But how many steps will this algorithm perform? Of course, the answer depends on what we call a single step. Since we do not want the number of steps to depend on the actual implementation we ignore constant factors. In any reasonable computer, ① will take at least $2n+1$ steps (this many variable assignments are done) and at most cn steps for some constant c. The following common notation is useful for ignoring constant factors:

Definition 1.2. *Let $f, g : D \to \mathbb{R}_+$ be two functions. We say that f is $O(g)$ (and sometimes write $f = O(g)$) if there exist constants $\alpha, \beta > 0$ such that $f(x) \leq \alpha g(x) + \beta$ for all $x \in D$. If $f = O(g)$ and $g = O(f)$ we also say that $f = \Theta(g)$ (and of course $g = \Theta(f)$). In this case, f and g have the same* **rate of growth**.

Note that the use of the equation sign in this notation is not symmetric. For example, let $D = \mathbb{N}$, and let $f(n)$ be the number of elementary steps in ① and $g(n) = n$ ($n \in \mathbb{N}$). Clearly we have $f = O(g)$ (in fact $f = \Theta(g)$) in this case; we say that ① takes $O(n)$ time (or linear time). A single execution of ③ takes a constant number of steps (we speak of $O(1)$ time or constant time) except in the case $k \leq n$ and $i = n$; in this case the cost of two paths have to be compared, which takes $O(n)$ time.

What about ②? A naive implementation, checking for each $j \in \{\pi(i)+1, \ldots, n\}$ and each $h \in \{1, \ldots, i-1\}$ whether $j = \pi(h)$, takes $O((n - \pi(i))i)$ steps, which can be as big as $\Theta(n^2)$. A better implementation of ② uses an auxiliary array indexed by $1, \ldots, n$:

② **For** $j := 1$ **to** n **do** $aux(j) := 0$.
For $j := 1$ **to** $i-1$ **do** $aux(\pi(j)) := 1$.
Set $k := \pi(i) + 1$.
While $k \leq n$ and $aux(k) = 1$ **do** $k := k + 1$.

Obviously with this implementation a single execution of ② takes only $O(n)$ time. Simple techniques like this are usually not elaborated in this book; we assume that the reader can find such implementations himself.

Having computed the running time for each single step we now estimate the total amount of work. Since the number of permutations is $n!$ we only have to estimate the amount of work which is done between two permutations. The counter i might move back from n to some index i' where a new value $\pi(i') \leq n$ is found. Then it moves forward again up to $i = n$. While the counter i is constant each of ② and ③ is performed once. So the total amount of work between two permutations

consists of at most $2n$ times ② and ③, i.e. $O(n^2)$. So the overall running time of the PATH ENUMERATION ALGORITHM is $O(n^2 n!)$.

One can do slightly better; a more careful analysis shows that the running time is only $O(n \cdot n!)$ (Exercise 3).

Still the algorithm is too time-consuming if n is large. The problem with the enumeration of all paths is that the number of paths grows exponentially with the number of points; already for 20 points there are $20! = 2432902008176640000 \approx 2.4 \cdot 10^{18}$ different paths and even the fastest computer needs several years to evaluate all of them. So complete enumeration is impossible even for instances of moderate size.

The main subject of combinatorial optimization is to find better algorithms for problems like this. Often one has to find the best element of some finite set of feasible solutions (in our example: drilling paths or permutations). This set is not listed explicitly but implicitly depends on the structure of the problem. Therefore an algorithm must exploit this structure.

In the case of the DRILLING PROBLEM all information of an instance with n points is given by $2n$ coordinates. While the naive algorithm enumerates all $n!$ paths it might be possible that there is an algorithm which finds the optimum path much faster, say in n^2 computation steps. It is not known whether such an algorithm exists (though results of Chapter 15 suggest that it is unlikely). Nevertheless there are much better algorithms than the naive one.

1.2 Running Time of Algorithms

One can give a formal definition of an algorithm, and we shall in fact give one in Section 15.1. However, such formal models lead to very long and tedious descriptions as soon as algorithms are a bit more complicated. This is quite similar to mathematical proofs: Although the concept of a proof can be formalized nobody uses such a formalism for writing down proofs since they would become very long and almost unreadable.

Therefore all algorithms in this book are written in an informal language. Still the level of detail should allow a reader with a little experience to implement the algorithms on any computer without too much additional effort.

Since we are not interested in constant factors when measuring running times we do not have to fix a concrete computing model. We count elementary steps, but we are not really interested in how elementary steps look like. Examples of elementary steps are variable assignments, random access to a variable whose index is stored in another variable, conditional jumps (if – then – go to), and simple arithmetic operations like addition, subtraction, multiplication, division and comparison of numbers.

An algorithm consists of a set of valid inputs and a sequence of instructions each of which can be composed of elementary steps, such that for each valid input the computation of the algorithm is a uniquely defined finite series of elementary steps which produces a certain output. Usually we are not satisfied with finite

computation but rather want a good upper bound on the number of elementary steps performed:

Definition 1.3. *Let A be an algorithm which accepts inputs from a set X, and let $f : X \to \mathbb{R}_+$. If there exists a constant $\alpha > 0$ such that A terminates its computation after at most $\alpha f(x)$ elementary steps (including arithmetic operations) for each input $x \in X$, then we say that A* **runs in $O(f)$ time**. *We also say that the* **running time** *(or the* **time complexity***) of A is $O(f)$.*

The input to an algorithm usually consists of a list of numbers. If all these numbers are integers, we can code them in binary representation, using $O(\log(|a|+2))$ bits for storing an integer a. Rational numbers can be stored by coding the numerator and the denominator separately. The **input size** of an instance with rational data is the total number of bits needed for the binary representation.

Definition 1.4. *An algorithm with rational input is said to run in* **polynomial time** *if there is an integer k such that it runs in $O(n^k)$ time, where n is the input size, and all numbers in intermediate computations can be stored with $O(n^k)$ bits.*

An algorithm with arbitrary input is said to run in **strongly polynomial time** *if there is an integer k such that it runs in $O(n^k)$ time for any input consisting of n numbers and it runs in polynomial time for rational input. In the case $k = 1$ we have a* **linear-time algorithm**.

Note that the running time might be different for several instances of the same size (this was not the case with the PATH ENUMERATION ALGORITHM). We consider the worst-case running time, i.e. the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum running time of an instance with input size n, and say that the running time of such an algorithm is $O(f(n))$. For some algorithms we do not know the rate of growth of f but only have an upper bound.

The worst-case running time might be a pessimistic measure if the worst case occurs rarely. In some cases an average-case running time with some probabilistic model might be appropriate, but we shall not consider this.

If A is an algorithm which for each input $x \in X$ computes the output $f(x) \in Y$, then we say that A **computes** $f : X \to Y$. If a function is computed by some polynomial-time algorithm, it is said to be **computable in polynomial time**.

Polynomial-time algorithms are sometimes called "good" or "efficient". This concept was introduced by Cobham [1964] and Edmonds [1965]. Table 1.1 motivates this by showing hypothetical running times of algorithms with various time complexities. For various input sizes n we show the running time of algorithms that take $100n \log n$, $10n^2$, $n^{3.5}$, $n^{\log n}$, 2^n, and $n!$ elementary steps; we assume that one elementary step takes one nanosecond. As always in this book, log denotes the logarithm with basis 2.

As Table 1.1 shows, polynomial-time algorithms are faster for large enough instances. The table also illustrates that constant factors of moderate size are not very important when considering the asymptotic growth of the running time.

Table 1.2 shows the maximum input sizes solvable within one hour with the above six hypothetical algorithms. In (a) we again assume that one elementary step

Table 1.1.

n	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
10	3 μs	1 μs	3 μs	2 μs	1 μs	4 ms
20	9 μs	4 μs	36 μs	420 μs	1 ms	76 years
30	15 μs	9 μs	148 μs	20 ms	1 s	$8 \cdot 10^{15}$ y.
40	21 μs	16 μs	404 μs	340 ms	1100 s	
50	28 μs	25 μs	884 μs	4 s	13 days	
60	35 μs	36 μs	2 ms	32 s	37 years	
80	50 μs	64 μs	5 ms	1075 s	$4 \cdot 10^7$ y.	
100	66 μs	100 μs	10 ms	5 hours	$4 \cdot 10^{13}$ y.	
200	153 μs	400 μs	113 ms	12 years		
500	448 μs	2.5 ms	3 s	$5 \cdot 10^5$ y.		
1000	1 ms	10 ms	32 s	$3 \cdot 10^{13}$ y.		
10^4	13 ms	1 s	28 hours			
10^5	166 ms	100 s	10 years			
10^6	2 s	3 hours	3169 y.			
10^7	23 s	12 days	10^7 y.			
10^8	266 s	3 years	$3 \cdot 10^{10}$ y.			
10^{10}	9 hours	$3 \cdot 10^4$ y.				
10^{12}	46 days	$3 \cdot 10^8$ y.				

takes one nanosecond, (b) shows the corresponding figures for a ten times faster machine. Polynomial-time algorithms can handle larger instances in reasonable time. Moreover, even a speedup by a factor of 10 of the computers does not increase the size of solvable instances significantly for exponential-time algorithms, but it does for polynomial-time algorithms.

Table 1.2.

	$100n \log n$	$10n^2$	$n^{3.5}$	$n^{\log n}$	2^n	$n!$
(a)	$1.19 \cdot 10^9$	60000	3868	87	41	15
(b)	$10.8 \cdot 10^9$	189737	7468	104	45	16

(Strongly) polynomial-time algorithms, if possible linear-time algorithms, are what we look for. There are some problems where it is known that no polynomial-time algorithm exists, and there are problems for which no algorithm exists at all. (For example, a problem which can be solved in finite time but not in polynomial time is to decide whether a so-called regular expression defines the empty set; see Aho, Hopcroft and Ullman [1974]. A problem for which there exists no algorithm at all, the HALTING PROBLEM, is discussed in Exercise 1 of Chapter 15.)

However, almost all problems considered in this book belong to the following two classes. For the problems of the first class we have a polynomial-time

algorithm. For each problem of the second class it is an open question whether a polynomial-time algorithm exists. However, we know that if one of these problems has a polynomial-time algorithm, then all problems of this class do. A precise formulation and a proof of this statement will be given in Chapter 15.

The JOB ASSIGNMENT PROBLEM belongs to the first class, the DRILLING PROBLEM belongs to the second class.

These two classes of problems divide this book roughly into two parts. We first deal with tractable problems for which polynomial-time algorithms are known. Then, starting with Chapter 15, we discuss hard problems. Although no polynomial-time algorithms are known, there are often much better methods than complete enumeration. Moreover, for many problems (including the DRILLING PROBLEM), one can find approximate solutions within a certain percentage of the optimum in polynomial time.

1.3 Linear Optimization Problems

We now consider our second example given initially, the JOB ASSIGNMENT PROBLEM, and briefly address some central topics which will be discussed in later chapters.

The JOB ASSIGNMENT PROBLEM is quite different to the DRILLING PROBLEM since there are infinitely many feasible solutions for each instance (except for trivial cases). We can reformulate the problem by introducing a variable T for the time when all jobs are done:

$$\begin{array}{lrcll} \min & T & & & \\ \text{s.t.} & \sum_{j \in S_i} x_{ij} & = & t_i & (i \in \{1, \ldots, n\} \\ & x_{ij} & \geq & 0 & (i \in \{1, \ldots, n\},\ j \in S_i) \\ & \sum_{i : j \in S_i} x_{ij} & \leq & T & (j \in \{1, \ldots, m\}) \end{array} \tag{1.1}$$

The numbers t_i and the sets S_i $(i = 1, \ldots, n)$ are given, the variables x_{ij} and T are what we look for. Such an optimization problem with a linear objective function and linear constraints is called a **linear program**. The set of feasible solutions of (1.1), a so-called **polyhedron**, is easily seen to be convex, and one can prove that there always exists an optimum solution which is one of the finitely many extreme points of this set. Therefore a linear program can, theoretically, also be solved by complete enumeration. But there are much better ways as we shall see later.

Although there are several algorithms for solving linear programs in general, such general techniques are usually less efficient than special algorithms exploiting the structure of the problem. In our case it is convenient to model the sets S_i, $i = 1, \ldots, n$, by a **graph**. For each job i and for each employee j we have a

point (called vertex), and we connect employee j with job i by an edge if he or she can contribute to this job (i.e. if $j \in S_i$). Graphs are a fundamental combinatorial structure; many combinatorial optimization problems are described most naturally in terms of graph theory.

Suppose for a moment that the processing time of each job is one hour, and we ask whether we can finish all jobs within one hour. So we look for numbers x_{ij} ($i \in \{1, \dots, n\}$, $j \in S_i$) such that $0 \leq x_{ij} \leq 1$ for all i and j and $\sum_{i:j \in S_i} x_{ij} = 1$ for $i = 1, \dots, n$. One can show that if such a solution exists, then in fact an integral solution exists, i.e. all x_{ij} are either 0 or 1. This is equivalent to assigning each job to one employee, such that no employee has to do more than one job. In the language of graph theory we then look for a **matching** covering all jobs. The problem of finding optimal matchings is one of the best known combinatorial optimization problems.

We review the basics of graph theory and linear programming in Chapters 2 and 3. In Chapter 4 we prove that linear programs can be solved in polynomial time, and in Chapter 5 we discuss integral polyhedra. In the subsequent chapters we discuss some classical combinatorial optimization problems in detail.

But not by the Simplex Method

1.4 Sorting

Let us conclude this chapter by considering a special case of the DRILLING PROBLEM where all holes to be drilled are on one horizontal line. So we are given just one coordinate for each point p_i, $i = 1, \dots, n$. Then a solution to the drilling problem is easy, all we have to do is sort the points by their coordinates: the drill will just move from left to right. Although there are still $n!$ permutations, it is clear that we do not have to consider all of them to find the optimum drilling path, i.e. the sorted list. It is very easy to sort n numbers in nondecreasing order in $O(n^2)$ time.

To sort n numbers in $O(n \log n)$ time requires a little more skill. There are several algorithms accomplishing this; we present the well-known MERGE-SORT ALGORITHM. It proceeds as follows. First the list is divided into two sublists of approximately equal size. Then each sublist is sorted (this is done recursively by the same algorithm). Finally the two sorted sublists are merged together. This general strategy, often called "divide and conquer", can be used quite often. See e.g. Section 17.1 for another example.

We did not discuss recursive algorithms so far. In fact, it is not necessary to discuss them, since any recursive algorithm can be transformed into a sequential algorithm without increasing the running time. But some algorithms are easier to formulate (and implement) using recursion, so we shall use recursion when it is convenient.

MERGE-SORT ALGORITHM

Input: A list $a_1, \ldots, a_n$ of real numbers.

Output: A permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $a_{\pi(i)} \leq a_{\pi(i+1)}$ for all $i = 1, \ldots, n-1$.

① **If** $n = 1$ **then** set $\pi(1) := 1$ and **stop** (**return** π).

② Set $m := \lfloor \frac{n}{2} \rfloor$.
Let $\rho :=$MERGE-SORT$(a_1, \ldots, a_m)$.
Let $\sigma :=$MERGE-SORT$(a_{m+1}, \ldots, a_n)$.

③ Set $k := 1$, $l := 1$.
While $k \leq m$ and $l \leq n - m$ **do**:
If $a_{\rho(k)} \leq a_{m+\sigma(l)}$ **then** set $\pi(k+l-1) := \rho(k)$ and $k := k+1$
else set $\pi(k+l-1) := m + \sigma(l)$ and $l := l+1$.
While $k \leq m$ **do**: Set $\pi(k+l-1) := \rho(k)$ and $k := k+1$.
While $l \leq n - m$ **do**: Set $\pi(k+l-1) := m + \sigma(l)$ and $l := l+1$.

As an example, consider the list "69,32,56,75,43,99,28". The algorithm first splits this list into two, "69,32,56" and "75,43,99,28" and recursively sorts each of the two sublists. We get the permutations $\rho = (2, 3, 1)$ and $\sigma = (4, 2, 1, 3)$ corresponding to the sorted lists "32,56,69" and "28,43,75,99". Now these lists are merged as shown below:

					$k := 1$,	$l := 1$
$\rho(1) = 2$,	$\sigma(1) = 4$,	$a_{\rho(1)} = 32$,	$a_{\sigma(1)} = 28$,	$\pi(1) := 7$,		$l := 2$
$\rho(1) = 2$,	$\sigma(2) = 2$,	$a_{\rho(1)} = 32$,	$a_{\sigma(2)} = 43$,	$\pi(2) := 2$,	$k := 2$	
$\rho(2) = 3$,	$\sigma(2) = 2$,	$a_{\rho(2)} = 56$,	$a_{\sigma(2)} = 43$,	$\pi(3) := 5$,		$l := 3$
$\rho(2) = 3$,	$\sigma(3) = 1$,	$a_{\rho(2)} = 56$,	$a_{\sigma(3)} = 75$,	$\pi(4) := 3$,	$k := 3$	
$\rho(3) = 1$,	$\sigma(3) = 1$,	$a_{\rho(3)} = 69$,	$a_{\sigma(3)} = 75$,	$\pi(5) := 1$,	$k := 4$	
	$\sigma(3) = 1$,		$a_{\sigma(3)} = 75$,	$\pi(6) := 4$,		$l := 4$
	$\sigma(4) = 3$,		$a_{\sigma(4)} = 99$,	$\pi(7) := 6$,		$l := 5$

Theorem 1.5. *The* MERGE-SORT ALGORITHM *works correctly and runs in* $O(n \log n)$ *time.*

Proof: The correctness is obvious. We denote by $T(n)$ the running time (number of steps) needed for instances consisting of n numbers and observe that $T(1) = 1$ and $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 3n + 6$. (The constants in the term $3n + 6$ depend on how exactly a computation step is defined; but they do not really matter.)

We claim that this yields $T(n) \leq 12n \log n + 1$. Since this is trivial for $n = 1$ we proceed by induction. For $n \geq 2$, assuming that the inequality is true for $1, \ldots, n-1$, we get

$$T(n) \leq 12 \left\lfloor \frac{n}{2} \right\rfloor \log \left(\frac{2}{3} n \right) + 1 + 12 \left\lceil \frac{n}{2} \right\rceil \log \left(\frac{2}{3} n \right) + 1 + 3n + 6$$

$$\begin{aligned} &= 12n(\log n + 1 - \log 3) + 3n + 8 \\ &\leq 12n \log n - \frac{13}{2}n + 3n + 8 \leq 12n \log n + 1, \end{aligned}$$

because $\log 3 \geq \frac{37}{24}$. □

Of course the algorithm works for sorting the elements of any totally ordered set, assuming that we can compare any two elements in constant time. Can there be a faster, a linear-time algorithm? Suppose that the only way we can get information on the unknown order is to compare two elements. Then we can show that any algorithm needs at least $\Theta(n \log n)$ comparisons in the worst case. The outcome of a comparison can be regarded as a zero or one; the outcome of all comparisons an algorithm does is a 0-1-string (a sequence of zeros and ones). Note that two different orders in the input of the algorithm must lead to two different 0-1-strings (otherwise the algorithm could not distinguish between the two orders). For an input of n elements there are $n!$ possible orders, so there must be $n!$ different 0-1-strings corresponding to the computation. Since the number of 0-1-strings with length less than $\lfloor \frac{n}{2} \log \frac{n}{2} \rfloor$ is $2^{\lfloor \frac{n}{2} \log \frac{n}{2} \rfloor} - 1 < 2^{\frac{n}{2} \log \frac{n}{2}} = (\frac{n}{2})^{\frac{n}{2}} \leq n!$ we conclude that the maximum length of the 0-1-strings, and hence of the computation, must be at least $\frac{n}{2} \log \frac{n}{2} = \Theta(n \log n)$.

In the above sense, the running time of the MERGE-SORT ALGORITHM is optimal up to a constant factor. However, there is an algorithm for sorting integers (or sorting strings lexicographically) whose running time is linear in the input size; see Exercise 6.

Lower bounds like the one above are known only for very few problems (except trivial linear bounds). Often a restriction on the set of operations is necessary to derive a superlinear lower bound.

See Wikipedia: Sorting algorithm

Exercises

DO → 1. Prove that $\log(n!) = \Theta(n \log n)$.

DO → 2. Prove that $n \log n = O(n^{1+\epsilon})$ for any $\epsilon > 0$.

3. Show that the running time of the PATH ENUMERATION ALGORITHM is $O(n \cdot n!)$.

4. Suppose we have an algorithm whose running time is $\Theta(n(t + n^{1/t}))$, where n is the input length and t is a positive parameter we can choose arbitrarily. How should t be chosen (depending on n) such that the running time (as a function of n) has a minimum rate of growth?

5. Let s, t be binary strings, both of length m. We say that s is lexicographically smaller than t if there exists an index $j \in \{1, \ldots, m\}$ such that $s_i = t_i$ for $i = 1, \ldots, j-1$ and $s_j < t_j$. Now given n strings of length m, we want to sort them lexicographically. Prove that there is a linear-time algorithm for this problem (i.e. one with running time $O(nm)$).
Hint: Group the strings according to the first bit and sort each group.

Think 6. Describe an algorithm which sorts a list of natural numbers $a_1, \ldots, a_n$ in linear time; i.e. which finds a permutation π with $a_{\pi(i)} \leq a_{\pi(i+1)}$ $(i = 1, \ldots, n-1)$ and runs in $O(\log(a_1 + 1) + \cdots + \log(a_n + 1))$ time.
Hint: First sort the strings encoding the numbers according to their length. Then apply the algorithm of Exercise 5.
Note: The algorithm discussed in this and the previous exercise is often called radix sorting.

References

General Literature:

Knuth, D.E. [1968]: The Art of Computer Programming; Vol. 1. Addison-Wesley, Reading 1968 (3rd edition: 1997)

Cited References:

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. [1974]: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading 1974

Cobham, A. [1964]: The intrinsic computational difficulty of functions. Proceedings of the 1964 Congress for Logic Methodology and Philosophy of Science (Y. Bar-Hillel, ed.), North-Holland, Amsterdam 1964, pp. 24-30

Edmonds, J. [1965]: Paths, trees, and flowers. Canadian Journal of Mathematics 17 (1965), 449-467