Chapter 2

Divide-and-conquer algorithms

A *computational problem* $\mathcal{P}$ takes an input ("instance") x from a set $\mathcal{I}$ of inputs and transforms x into a result r from a set $\mathcal{O}$. Formally, one can describe such a problem as a function $f\colon \mathcal{I} \to \mathcal{O}$. A *divide-and-conquer algorithm* $\mathcal{A}$ solves a problem $\mathcal{P}$ on an instance x of size $n = |x|$ either directly (if n is small enough) or by using recursive calls for smaller instances (if n is larger).

The general structure of a divide-and-conquer algorithm is as follows. We want to solve a computational problem $\mathcal{P}$ on inputs x.

(0) Triviality test: If $|x| \leq n_0$, then solve problem $\mathcal{P}$ for x by a "direct" or "simple" method; return the result r.
(Here n_0 is some constant. It takes constant time to solve the problem on a "trivial" instance of size $\leq n_0$.)

(1) Divide/Split: From x calculate a many smaller problem instances $x_1, \ldots, x_a$.

(2) Recursion: Make a **recursive calls** (to the algorithm) to solve $\mathcal{P}$ on the instances $x_1, \ldots, x_a$. Results: $r_1, \ldots, r_a$.

(3) Combine: From $x, x_1, \ldots, x_a, r_1, \ldots, r_a$ calculate solution r that solves $\mathcal{P}$ for x.

In this chapter, we will study several examples of algorithms that are based on this idea. The first of these algorithms shows that we can indeed multiply two n-bit integers faster than in time $O(n^2)$. (Compare the school method in Section 1.1.2.)

2.1 Faster integer multiplication: Karatsuba's algorithm

Assume we want to multiply two n-bit integers x and y. This means that the input is the *pair* (x, y). As its size $|(x, y)|$ we take n. For simplicity assume n is a power of 2.

(0) Triviality test: If $n = 1$, then the bits x and y can be multiplied by one boolean operation[1]: $r := x \wedge y$.

Now assume $n > 1$.

(1) Divide/Split: Split the binary representation of x into the left half x_L and the right half x_R:

$$x = \boxed{x_L}\boxed{x_R} = x_L \cdot 2^{n/2} + x_R.$$

Split the binary representation of y into left half y_L and right half y_R:

$$y = \boxed{y_L}\boxed{y_R} = y_L \cdot 2^{n/2} + y_R.$$

Example: For $x = 10101100_2$ and $y = 10110101_2$ (binary notation for 172 and 181) we get

$$x = 1010_2 \cdot 2^4 + 1100_2 = 10 \cdot 16 + 12 \quad \text{and} \quad y = 1011_2 \cdot 2^4 + 0101_2 = 11 \cdot 16 + 5.$$

Then

$$x \cdot y = \boxed{x_L \cdot y_L} \cdot 2^n + \left(\boxed{x_L \cdot y_R} + \boxed{x_R \cdot y_L}\right) \cdot 2^{n/2} + \boxed{x_R \cdot y_R}. \tag{2.1}$$

We see four instances of multiplications of $n/2$-bit integers (in the boxes).

(2) Recursion: The four instances from (1) are solved recursively. From the four recursive calls we obtain the products

$$r_1 = x_L \cdot y_L, \quad r_2 = x_L \cdot y_R, \quad r_3 = x_R \cdot y_L, \quad r_4 = x_R \cdot y_R.$$

(3) Combine: Now we can calculate

$$x \cdot y = r_1 \cdot 2^n + (r_2 + r_3) \cdot 2^{n/2} + r_4,$$

which involves only adding four $2n$-bit numbers, hence this costs time $O(n)$. (Note that multiplication by powers of 2 is for free or costs at most linear time, since it amounts to appending zeroes, i.e., a shift.)

Let $T(n)$ be the cost (running time or number of binary operations) for this algorithm when applied to two n-bit numbers. (To be precise, $T(n)$ is the maximum time needed for any instance of size n.) We get, for some constants g and c:

$$T(n) \leq \begin{cases} g & \text{for } n = 1 \\ 4 \cdot T(n/2) + c \cdot n & \text{for } n > 1. \end{cases} \tag{2.2}$$

This is because the cost for $n = 1$ is constant and for $n > 1$ the cost for multiplying two n-bit numbers is the sum of the cost of triviality test, splitting and combining

[1] When implementing the algorithm, one takes as n_0 a bigger number, like 16 or 32. Such small numbers are multiplied by the hardware. However, also see Remark (c) at the end of this section.

(which is $O(n)$ since splitting is almost free and combining costs time $O(n)$) and of four recursive calls with numbers half the size. Each of these costs $T(n/2)$.

An equality like (2.2) is called a *recurrence relation* for $T(n)$. We will see below a general method for solving such recurrence relations, i.e., for finding a closed formula that is an upper bound on $T(n)$. This general method will give $T(n) = O(n^2)$.

Now this is the same as the time we get from the school method for multiplication. How disappointing!

In the example one sees that the "divide-and-conquer" idea in itself is not necessarily enough to get a better running time than by the obvious and straightforward algorithms. For multiplication one needs another idea. The purpose of this idea must be to reduce the number of recursive calls in an instance from four to three. The book describes "Gauss' trick" for multiplying two complex numbers by carrying out only three multiplications of reals instead of the obvious four. We head directly to Karatsuba's trick for integer multiplication, which is similar to Gauss' trick.

Trick: Write

$$(x_L - x_R) \cdot (y_L - y_R) = x_L \cdot y_L + x_R \cdot y_R - (x_L \cdot y_R + x_R \cdot y_L).$$

This implies

$$x_L \cdot y_R + x_R \cdot y_L = x_L \cdot y_L + x_R \cdot y_R - (x_L - x_R) \cdot (y_L - y_R).$$

So if we calculate the *three* products

$$P_1 = \boxed{x_L \cdot y_L}, P_2 = \boxed{x_R \cdot y_R}, P_3' = \boxed{(x_L - x_R) \cdot (y_L - y_R)},$$

we can calculate the number $x_L \cdot y_R + x_R \cdot y_L$ needed in (2.1) as $P_1 + P_2 - P_3'$ just by additions and subtractions.

With this trick we finally get *Karatsuba's algorithm*.

Since the algorithm can process only nonnegative numbers, we have to be a little careful with the signs of the factors[2] $x_L - x_R$ and $y_L - y_R$. We let s_x be the sign (1 for nonnegative and -1 for negative) of $x_L - x_R$ and s_y the sign of $y_L - y_R$, and let

$$P_3 = |x_L - x_R| \cdot |y_L - y_R|.$$

Then $P_3' = s_x s_y \cdot P_3$, hence $x_L \cdot y_R + x_R \cdot y_L = P_1 + P_2 - s_x s_y \cdot P_3$, and we get

$$x \cdot y = P_1 \cdot 2^n + (P_1 + P_2 - s_x s_y \cdot P_3) \cdot 2^{n/2} + P_2.$$

For pseudocode see Algorithm 1.

[2] In many books, in particular in our book, one does not *subtract*, but *add* x_L and x_R, and y_L and y_R. Then there are no sign problems, but there are overflow problems, since $x_L + x_R$ may have $n/2 + 1$ bits. We avoid this by subtracting: The absolute values $|x_L - x_R|$ and $|y_L - y_R|$ always can be written with $n/2$ bits.

Algorithm 1: Karatsuba multiplication

function multiplyK(x, y)
INPUT: Two n-bit integers $x, y \geq 0$, where n is a power of 2
OUTPUT: The product $x \cdot y$ in binary.
METHOD:
// Triviality test:
  if $n = 1$ **then return** $x \wedge y$
// Divide/Split: Prepare smaller inputs:
  $x_L \leftarrow$ leftmost $\frac{n}{2}$ bits of x; $x_R \leftarrow$ rightmost $\frac{n}{2}$ bits of x;
  $y_L \leftarrow$ leftmost $\frac{n}{2}$ bits of y; $y_R \leftarrow$ rightmost $\frac{n}{2}$ bits of y;
// 3 recursive calls:
  $P_1 \leftarrow$ `multiplyK`(x_L, y_L);
  $P_2 \leftarrow$ `multiplyK`(x_R, y_R);
  $P_3 \leftarrow$ `multiplyK`$(|x_L - x_R|, |y_L - y_R|)$;
// Combine:
  $s_x \leftarrow \text{sign}(x_L - x_R)$; $s_y \leftarrow \text{sign}(y_L - y_R)$;
  return $P_1 \cdot 2^n + (P_1 + P_2 - s_x \cdot s_y \cdot P_3) \cdot 2^{n/2} + P_2$

Example: $x = 10101100_2$, $y = 10110101_2$. The length is $n = 8$.

$x_L = 1010_2$, $x_R = 1100_2$, $y_L = 1011_2$, $y_R = 0101_2$, binary numbers of length 4.

$x = 1010_2 \cdot 2^4 + 1100_2 = 10 \cdot 16 + 12$, $y = 1011_2 \cdot 2^4 + 0101_2 = 11 \cdot 16 + 5$.

Differences: $x_L - x_R = 1010_2 - 1100_2 = -2 = -0010_2$ and $y_L - y_R = 1011_2 - 0101_2 = 6 = 0110_2$. Hence $s_x = -1$ and $s_y = 1$.

Three subinstances, with results (provided by the recursive calls) in binary:

$$P_1 = 1010_2 \cdot 1011_2 = 10 \cdot 11 = 110 = 01101110_2,$$
$$P_2 = 1100_2 \cdot 0101_2 = 12 \cdot 5 = 60 = 00111100_2,$$
$$P_3 = 0010_2 \cdot 0110_2 = 2 \cdot 6 = 12 = 00001100_2.$$

It follows the combination step:

$$\begin{aligned} x \cdot y &= 01101110_2 \cdot 2^8 + (01101110_2 + 00111100_2 - (-1) \cdot 1 \cdot 00001100_2) \cdot 2^4 + 00111100_2 \\ &= 01101110_2 \cdot 2^8 + (01101110_2 + 00111100_2 + 00001100_2) \cdot 2^4 + 00111100_2. \end{aligned}$$

This can be calculated by shifting and adding[3] five binary numbers:

[3] It is a mere coincidence that there is no subtraction. We have $(-1)s_x s_y = 1$.

$$\begin{array}{rr} & 01101110 \\ & \quad 01101110 \\ & \quad 00111100 \\ & \quad 00001100 \\ + & \qquad 00111100 \\ \hline & 0111100110011100 \end{array}$$

The result is $0111100110011100_2 = 31132 = 172 \cdot 181$, which is the correct product.

Analysis of the algorithm: Now we estimate the running time $T_{\mathrm{K}}(n)$ of Karatsuba's algorithm. Since there are only three recursive calls, the recurrence relation (2.2) turns into

$$T_{\mathrm{K}}(n) \leq \begin{cases} g & \text{for } n = 1 \\ 3 \cdot T_{\mathrm{K}}(n/2) + c \cdot n & \text{for } n > 1. \end{cases} \tag{2.3}$$

(The combination cost $c \cdot n$ is a little higher than in the naive algorithm with four subinstances.) To find a closed expression for $T_{\mathrm{K}}(n)$, we proceed as follows. Let $n = 2^L$, hence $L = \log_2 n$. We keep substituting (2.2).

$$\begin{aligned} T_{\mathrm{K}}(n) &\leq 3T_{\mathrm{K}}(n/2) + cn \\ &\leq 3(3T_{\mathrm{K}}(n/4) + c(n/2)) + cn \\ &= 3^2 T_{\mathrm{K}}(n/2^2) + 3cn/2 + cn \\ &\leq 3^2(3T_{\mathrm{K}}(n/2^3) + c(n/2^2)) + 3cn/2 + cn \\ &= 3^3 T_{\mathrm{K}}(n/2^3) + cn((3/2)^2 + 3/2 + 1) \\ &\vdots \\ &\leq 3^i T_{\mathrm{K}}(n/2^i) + cn \cdot \sum_{0 \leq j < i} \left(\frac{3}{2}\right)^j \\ &\vdots \\ &\leq 3^L T_{\mathrm{K}}(n/2^L) + cn \cdot \sum_{0 \leq j < L} \left(\frac{3}{2}\right)^j . \end{aligned} \tag{2.4}$$

Regarding the first term in (2.4) we see that $T_{\mathrm{K}}(n/2^L) = T_{\mathrm{K}}(n/n) = T_{\mathrm{K}}(1) \leq g$. The second term is determined by the geometric series

$$\sum_{0 \leq j < L} \left(\frac{3}{2}\right)^j = \frac{(\frac{3}{2})^L - 1}{\frac{3}{2} - 1} < 2\left(\frac{3}{2}\right)^L = \frac{2 \cdot 3^L}{2^L} = \frac{2 \cdot 3^L}{n}.$$

So we can estimate:

$$T_{\mathrm{K}}(n) \leq 3^L g + cn \cdot \frac{2 \cdot 3^L}{n} = 3^L(g + 2c).$$

And what is 3^L? We calculate

$$3^L = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{(\log_2 3)(\log_2 n)} = 2^{(\log_2 n)(\log_2 3)} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3}.$$

Hence

$$T_{\mathrm{K}}(n) = O(n^{\log_2 3}).$$

Note that $\log_2 3 = 1.5849\cdots < 2$, so the running time of Karatsuba's algorithm is much smaller than that of the school method, at least for large n.

Theorem 2.1.1
Karatsuba's algorithm multiplies two n-bit numbers in time $O(n^{1.59})$.

Remarks: (a) Multiplication algorithms with running times of $O(n^{1+\varepsilon})$ for arbitrary $\varepsilon > 0$ have been developed after Karatsuba's algorithm was published in 1962, and the algorithm by Schönhage and Strassen with running time $O(n \log n \log\log n)$ appeared in 1971. Some improvements were made in the 2000s, and only in 2019 a multiplication algorithm with running time $O(n \log n)$ was presented by David Harvey and Joris van der Hoeven. Unfortunately, the latter one is not (yet?) practical.
(b) One can use Karatsuba's algorithm also for n that is not a power of 2. For the program see Fig. 2.1 in the book. The running time bound is the same.
(c) When using Karatsuba's algorithm, the recursion should be stopped at some larger n_0, where one should switch to the school method, i.e., Algorithm 2 in Section 1.1.2. The resulting "hybrid" algorithm is faster than the school method already for numbers with length n around 1000.

2.2 Recurrence relations and the master theorem

Read again the introductory segment of Chapter 2, where divide-and-conquer algorithms are described in general. In the present section we want to develop a method for estimating the running time of a big class of algorithms that follow this paradigm (not all, unfortunately).

Consider some divide-and-conquer algorithm $\mathcal{A}$. Let $T(n) := T_{\mathcal{A}}(n)$ be the worst case running time of $\mathcal{A}$ on inputs of size n. Clearly, $T(1) = O(1)$. If $n > 1$, then $T(n)$ is composed of the time required for steps (0) through (3) in the scheme. We assume that the triviality test (find out if $|x| \le n_0$ or not), the splitting step and the combining step together take polynomial time, meaning time $O(n^d)$ for some constant d. Further, we assume that the number a of subinstances formed in the dividing step does not depend on n but is always the same, and that the size of the subinstances is at most $\lceil n/b \rceil$ for a constant $b > 1$. (Karatsuba's algorithm is an example for an algorithm that satisfies these assumptions with $a = 3$, $b = 2$, $d = 1$.) The time in the recursion step can then be estimated as follows: We have a recursive calls on

instances of size $\lceil n/b \rceil$. The time for all these taken together is $a \cdot T(\lceil n/b \rceil)$. Taking all contributions together, we arrive at the following system of inequalities:

$$T(n) \leq \begin{cases} g, & \text{if } n = 1, \text{ for some constant } g, \\ a \cdot T\left(\lceil \frac{n}{b} \rceil\right) + c \cdot n^d, & \text{if } n > 1, \text{ for some constant } c. \end{cases} \tag{2.5}$$

Such a system is called a *recurrence relation*. We always assume[4] that $a > 0$, $b > 1$, and $d \geq 0$.

2.2.1 Visualizing the running time

Figure 2.1 depicts the time spent by a divide-and-conquer algorithm on a (large) instance x of size n, as described in the previous section. For simplicity we assume that n is a power of b, so we can omit rounding. Let $n = b^L$, or $L = \log_b n$. Every node in the tree corresponds to a problem instance for which there is a (direct or indirect) recursive call when algorithm $\mathcal{A}$ is run on instance x. The node label is the running time of a particular subinstance, ignoring recursion. (In fact that is the $c \cdot n^d$-running time in (2.5) which accounts for dividing and combining.) For example, on level 0 the size of the instance is n, and we use time $c \cdot n^d$ (ignoring recursion), on level 1 the size of an instance is n/b, and we use time $c \cdot (n/b)^d$ per node (ignoring recursion) and so on. The time per node on level i, for $0 \leq i \leq L$ is $c \cdot (n/b^i)^d$, since by (2.5) the instance size shrinks by a factor of b from level to level. How many nodes do we have on the different levels? Level 0 just consists of a single node, which corresponds to the actual instance to be solved. The instance on level 0 is split into a subinstances, which make up the a nodes on level 1. The a^i instances on level i, for $0 \leq i < L$, are split into a subinstances each, hence there are $a \cdot a^i = a^{i+1}$ subinstances on level $i+1$. This continues until the Case "$n = 1$" in (2.5) is reached, with a^L nodes on level L.

2.2.2 The master theorem

In this section we will estimate $T(n)$, which is given by (2.5), in O-notation.

[4] Although our illustration uses that a is an integer and that $n = b^L$ for an integer b, these are inessential assumptions. By more careful considerations one can show that the result we are about to prove holds for arbitrary real numbers $a > 0$, $b > 1$, and $d \geq 0$.

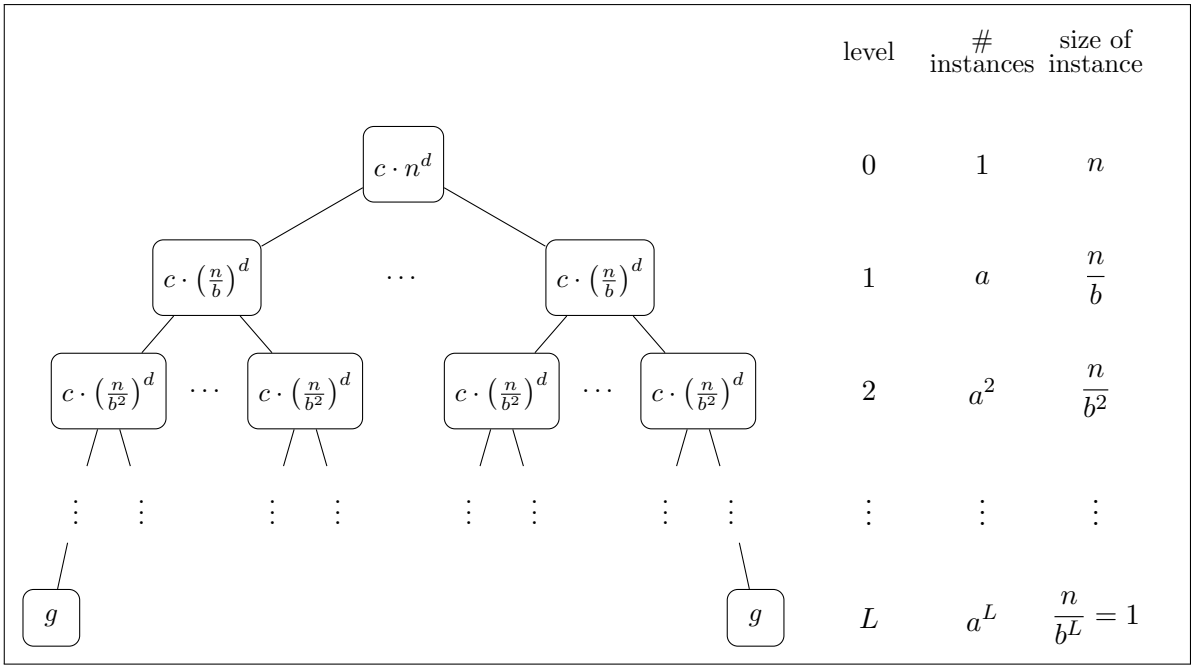

Figure 2.1: The recursion tree for a DaC-Algorithm whose running time satisfies (2.5).

Theorem 2.2.1 ***Master theorem***
Assume that for arbitrary fixed integers $a > 0$, $b > 1$ and real $d \geq 0$ the function $T(n)$ obeys the recurrence relation

$$T(n) \leq \begin{cases} g, & \text{if } n = 1, \text{ for some constant } g, \\ a \cdot T\left(\lceil \frac{n}{b} \rceil\right) + c \cdot n^d, & \text{if } n > 1, \text{ for some constant } c. \end{cases}$$

Then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a, \\ O(n^d \log n), & \text{if } d = \log_b a, \\ O(n^{\log_b a}), & \text{if } d < \log_b a. \end{cases} \tag{2.6}$$

Proof of Theorem 2.2.1. We will see shortly that the proof mainly consists in evaluating a geometric series. So recall the formula:

$$\sum_{0 \leq i < L} q^i = \begin{cases} \frac{1-q^L}{1-q} < \frac{1}{1-q}, & \text{if } 0 < q < 1, \\ L, & \text{if } q = 1, \\ \frac{q^L - 1}{q-1}, & \text{if } q > 1. \end{cases} \tag{2.7}$$

We only consider the case that n is a power of b. (The theorem is also true for arbitrary values of n, but the proof is a little more technical.)

Consider the recursion tree in Figure 2.1. The size of every problem instance on level i is n/b^i (we can omit $\lceil\cdot\rceil$, because n is a power of b). The recursion stops on level $L = \log_b n$. On level i there are a^i sub-instances of size n/b^i each. By Case "$n > 1$" in (2.5), the time for all these subinstances on level i, $0 \le i < L$, taken together, ignoring recursion, is

$$a^i \cdot c \cdot (n/b^i)^d = c \cdot n^d \cdot (a/b^d)^i = c \cdot n^d \cdot q^i, \tag{2.8}$$

where $q := a/b^d$. Note that q is a constant. The time for level L is at most $a^L \cdot g$ (by Case "$n = 1$" in the recurrence). We obtain the following upper bound on the total time by summing all these contributions over all levels $0 \le i \le L$:

$$T(n) \le c \cdot n^d \cdot \sum_{0 \le i < L} q^i + a^L \cdot g. \tag{2.9}$$

We first calculate a^L, using simple exponentiation rules:

$$a^L = (b^{\log_b a})^{\log_b n} = b^{(\log_b a)(\log_b n)} = b^{(\log_b n)(\log_b a)} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}. \tag{2.10}$$

(Note how natural it is that the exponent $\log_b a$ appears when one transforms a^L, the number of leaves, to an expression to the base n.)

Now we use (2.7) to bound the geometric series $S_L := \sum_{0 \le i < L} q^i$ in (2.9) from above. There are three cases, corresponding to the three cases for the evaluation of the geometric series.

Case 1: $q < 1$. – By the definition of q this means $a < b^d$, or $\log_b a < d$. By (2.7) we have that $S_L \le \frac{1}{1-q}$, and (2.9) turns into

$$T(n) \le \frac{c \cdot n^d}{1-q} + O(n^{\log_b a}) = O(n^d) + O(n^{\log_b a}) = O(n^d).$$

In this case the split/combine cost of the original instance of size n dominates the time for the whole recursion.

Case 2: $q = 1$. – By the definition of q this means $a = b^d$, or $\log_b a = d$. By (2.7) we have $S_L = L = \log_b n$, and (2.9) turns into

$$T(n) \le cn^d \cdot \log_b n + O(n^{\log_b a}) = O(n^d \log n).$$

In this case each recursion level contributes the same to the total cost, since the variable parts in the number a^i of instances and the costs $cn^d/(b^d)^i$ of the instances cancel each other.

Case 3: $q > 1$. – By the definition of q this means $a > b^d$, or $\log_b a > d$. By (2.7) we have $S_L \leq \frac{q^L}{q-1}$. Now $q^L = a^L/b^{dL} = a^L/(b^L)^d = a^L/n^d$, and so (2.9) turns into

$$T(n) \leq c \cdot n^d \cdot \frac{a^L}{n^d} + O(n^{\log_b a}) = O(a^L) + O(n^{\log_b a}) = O(n^{\log_b a}).$$

In this case the recursion levels become more expensive as they become deeper; actually, the bottom level $i = L$ dominates the running time, up to a constant factor. □

Let us return for a moment to the two algorithms for multiplying integers that we considered in the previous section. The first one (without Gauss' trick) led to the recurrence (2.2). Thus we have $a = 4$, $b = 2$, $d = 1$. Since $\log_2 4 = 2 > 1 = d$, this gets us into Case 3, and the running time of this algorithm is $O(n^2)$, as claimed in Section 2.1. Karatsuba's algorithm led to the recurrence (2.3). Now we have $a = 3$, $b = 2$, $d = 1$. Since $\log_2 3 = 1.5849\cdots > 1 = d$, we are in Case 3 as well, and the running time is $O(n^{\log_2 3})$. This is the same result that we obtained by a direct calculation in Section 2.1.

Example: Binary search. Binary search is a fast search procedure that assumes that we have an array `A[1..n]` filled with keys `A[i]`, $1 \leq i \leq n$, which are *sorted*, i.e., we have `A[i]` $<$ `A[j]` if $i < j$. Given a search key x, we are to determine if x occurs in the array. Assuming that $n \geq 1$, we compare x with an element in the middle of the array, namely we let $m = \lceil \frac{n}{2} \rceil$ and compare x with `A[m]`. If $x <$ `A[m]`, we recursively search in `A[1..m − 1]`. (This is o.k., since x is not in position m, and it cannot be in positions $m+1, \ldots, n$ because of the ordering.) If $x =$ `A[m]`, the search ends successfully. If $x >$ `A[m]`, we recursively search in `A[m + 1..n]`.

If one wants to implement this idea, one quickly notices that one needs a recursive function **rsearch**(ℓ, r) that checks whether x is contained in `A[`ℓ`..r]`, for $1 \leq \ell \leq n+1$ and $0 \leq r \leq n$. This function works as follows: (0) Triviality test: If $r < \ell$, the array in question is empty and the answer is "no". (1) Splitting, (2) Recursion, (3) Combining: Otherwise we calculate $m = \ell + \lfloor \frac{1}{2}(r - \ell) \rfloor$, and consider three cases. If $x <$ `A[m]`, we return the result of the recursive call **rsearch**$(\ell, m-1)$. If $x =$ `A[m]`, we return "yes". If $x >$ `A[m]`, we return the result of the recursive call **rsearch**$(m+1, r)$. To search for x in `A[1..n]` we call **rsearch**$(1, n)$. (It is an easy exercise to write up pseudocode for this.)

Clearly, Binary search is a divide-and-conquer algorithm. The triviality test costs time $O(1)$. Splitting consists of the computation of m and the comparison of x with `A[m]`. The length of the array for which the recursive call happens is at most half that of `A[`ℓ`..r]`. Combining consists just in passing on the result, and it takes time $O(1)$.

If $T(n)$ denotes the maximum time for carrying out **rsearch**(`A[`ℓ`..r]`) for a subarray

of length $n = r - \ell + 1$, we get the following recurrence:

$$T(n) \leq \begin{cases} g, & \text{if } n \leq 1, \text{ for some constant } g, \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + c, & \text{if } n > 1, \text{ for some constant } c. \end{cases}$$

At the first glance this does not look like what we expect in the master theorem. Where is a? Where is d? But note that $T\left(\lfloor \frac{n}{2} \rfloor\right) = 1 \cdot T\left(\lfloor \frac{n}{2} \rfloor\right)$ and $c = c \cdot n^0$. So $a = 1$, $b = 2$, and $d = 0$ are the right choices. We compare:

$$\log_b a = \log_2 1 = 0 = d.$$

This means we are in Case 2 in the master theorem, and our recurrence equation implies $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$.

Summing up: Binary search in an array of length n has running time $O(\log n)$.

2.3 Mergesort

The *sorting problem* is the following: Given is a sequence $(a_1, \ldots, a_n)$ of objects, each equipped with a *key* from an ordered set (numbers, strings in lexicographic order, or the like). The task is to rearrange the objects so that the keys are in increasing (or, more generally, nondecreasing order). If the sequence of objects is $((5, \texttt{e}), (17, \texttt{e}), (12, \texttt{g}), (2, \texttt{m}), (28, \texttt{o}), (8, \texttt{r}), (35, \texttt{r}), (22, \texttt{s}), (42, \texttt{t}))$, where the first component in a pair is the key, the result of sorting is

$$((2, \texttt{m}), (5, \texttt{e}), (8, \texttt{r}), (12, \texttt{g}), (17, \texttt{e}), (22, \texttt{s}), (28, \texttt{o}), (35, \texttt{r}), (42, \texttt{t})).$$

Note that the nonkey parts of the objects are just dragged along with the keys. We omit these parts in examples, so that the previous example would just look like this: If we sort $(5, 17, 12, 2, 28, 8, 35, 22, 42)$, the result is $(2, 5, 8, 12, 17, 22, 28, 35, 42)$.

Mergesort is a sorting algorithm that follows the divide-and-conquer paradigm. We follow our scheme: Input: S, a sequence of n objects.

Triviality test: If $n \leq 1$, then just return the input.[5]

Split/divide: Split S into two disjoint sequences S_1, S_2, as evenly as possible.

Recursion: Sort S_1 by recursion, result R_1; sort S_2 by recursion, result R_2.

Combine: "Merge" the two sorted sequences R_1 and R_2 into one sorted sequence R; return R as the result.

[5] In real implementations, one would use a simple direct sorting method if $n \leq n_0$ for some constant n_0 like $n_0 = 8$.

How do we split? This depends on the way the sequence S is given. If S is written in a subarray $A[\ell, r]$, with $\ell < r$, we calculate $m = \lfloor(\ell + r)/2\rfloor = \ell + \lfloor(r - \ell)/2\rfloor$ and let S_1 be $A[\ell, m]$ and S_2 be $A[m+1, r]$. If S is given as a linked list L, we split L into two sublists L_1 and L_2 of (almost) equal lengths by running through L and alternatingly placing entries into L_1 and L_2.

Recursion takes care of itself, we need not discuss it at all. (That is the elegance of thinking about divide-and-conquer algorithms!)

How do we combine? The idea is to take objects one after the other from the two sorted sequences R_1 and R_2 in an interleaved fashion, so that always the smallest key still available is chosen, which then is transported to the end of sequence R we are building up. This idea can be expressed as follows as a recursive procedure: If R_1 is empty, return R_2. If R_2 is empty, return R_1. Otherwise look at the first keys x_1 in R_1 and x_2 in R_2. If $x_1 \leq x_2$ then let R_1' be R_1 without x_1; recursively merge R_1' and R_2; prepend x_1 to the result. If $x_1 > x_2$ then let R_2' be R_2 without x_2; recursively merge R_1 and R_2'; prepend x_2 to the result. This procedure is expressed in pseudocode in the function **merge** in the book (page 51). The big advantage of this formulation is that it is obviously correct, since the chosen key (x_1 or x_2) must be the smallest of all.

We describe the algorithm more concretely, namely as the task of merging two increasing sequences stored in nonempty subarrays `A[`$\ell..m$`]` and `A[`$m+1..r$`]`, where the result is written in the subarray $B[\ell..r]$. There we work with three pointers: `i` in `A[`$\ell..m$`]`, `j` in `A[`$m+1..r$`]`, and `k` in `B[`$\ell..r$`]`, which mark the beginning of the rest R_1, the beginning of the rest R_2, and the end of the output sequence R. So we keep comparing `A[i]` with `A[j]`, copying the smaller one to `B[k]`, and increasing the "used" pointer values, until we reach the end of one of the two input sequences. (For an illustration see Figure 2.2). Then the rest of the other sequence is just copied.

Algorithm 2: procedure merge(`A[`$\ell..r$`]`, m, `B[`$\ell..r$`]`)

```
1  INPUT: 1 ≤ ℓ ≤ m < r ≤ n, A[ℓ..m] and A[m+1..r] are sorted subarrays.
2  OUTPUT: B[ℓ..r]: output (sub)array, must be disjoint from A[ℓ..m].
3    i ← ℓ; j ← m+1; k ← ℓ;  // pointers to the beginning of the arrays
4    while i ≤ m and j ≤ r do
5      if A[i] ≤ A[j] then B[k] ← A[i]; i++; k++
6                     else B[k] ← A[j]; j++; k++
7  // i = m+1 or j = r+1: one of the input subarrays is exhausted
8  // the rest is just copying
9    while i ≤ m do B[k] ← A[i]; i++; k++
10   while j ≤ r do B[k] ← A[j]; j++; k++
11   return
```

The process of transportation is illustrated in Figure 2.2.

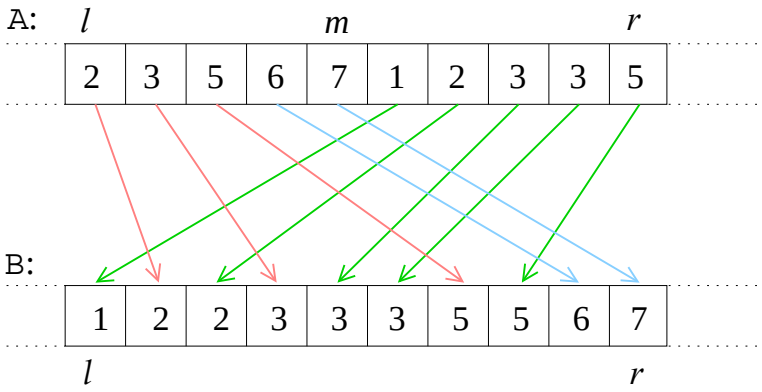

Figure 2.2: **merge**: Two sorted subsequences `A[`$\ell..m$`]` and `A[`$m+1..r$`]` are combined into one sorted sequence `B[`$\ell..r$`]`. The order of copying is from left to right in `B[`$\ell..r$`]`. Blue arrows show where objects are copied without comparison.

For correctness, one might be tempted to say that it is "clear" that it works. For students that would like to see a more precise argument, here is a sketch of a real proof. Following the recursive idea of the book, one can argue by induction on the total length of the *remaining* sequences `A[`$i..m$`]` and `A[`$j..r$`]`, where `i` contains i and `j` contains j. The induction claim is that if one starts in this situation, with $m-i+r-j+2$ entries still to process, and `k` containing $k = i+j-m-1$, these elements will be written in sorted order to the subarray `B[`$k..r$`]`. This is shown by induction. If $i = m+1$, the rest `A[`$j..r$`]` will just be copied (line **10**). If $j = r+1$, the rest `A[`$i..m$`]` will be copied (line **9**). Otherwise the minimum of `A[`i`]` and `A[`j`]` is copied to `B[`k`]`, which is the smallest since the two input sequences are sorted, and the two "used" indices are increased (lines **5** and **6**). The procedure continues with rests that are shorter by one than before. The induction hypothesis says that the elements in these rests are written to `B[`$k+1..r$`]` in increasing order.

Now assume an array `A[`$1..n$`]` is given as a global variable. Similarly given is an auxiliary array `B[`$1..n$`]`. We write up a procedure that recursively sorts the subarray `A[`$\ell..r$`]` of `A[`$1..n$`]`. This yields Algorithm 3.

The correctness of this algorithm is easily proved by induction on $r-\ell+1$, the number of elements in the subarray. What about running time? It is clear that **merge**(ℓ, r) has running time $O(r-\ell+1)$ (length of the subarray) and makes at most $r-\ell$ key comparisons. Let $T_{\text{MS}}(n)$ be the maximum time needed for any call **procedure rmsort**(ℓ, r), where $n = r-\ell+1$. We get the following recurrence:

$$T_{\text{MS}}(n) \leq \begin{cases} g, & \text{if } n = 1, \text{ for some constant } g, \\ 2 \cdot T_{\text{MS}}\left(\lceil \frac{n}{2} \rceil\right) + c \cdot n, & \text{if } n > 1, \text{ for some constant } c. \end{cases}$$

The constant g is for the cost of checking that $\ell = r$ and doing nothing. The additive term cn is for the cost of merging. We have two recursive calls with subarrays of length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Looking to the master theorem, we see that we have $a = 2$,

Algorithm 3: procedure rmsort(ℓ, r)

```
1 INPUT: A[ℓ..r] with 1 ≤ ℓ ≤ r ≤ n is a nonempty subarray of A[1..n]
2 OUTPUT: A[ℓ..r] contains the same objects as before, sorted in increasing
                                                                    order
3   if r > ℓ then                    // r ≤ ℓ is the trivial case, nothing to do
4     m ← ℓ + ⌊(r − ℓ)/2⌋
5     rmsort(ℓ, m)  // 1st recursive call
6     rmsort(m + 1, r)  // 2nd recursive call
7     merge(A[ℓ..r], m, B[ℓ..r])  // combine step, result in B[ℓ..r]
8     copy B[ℓ..r] into A[ℓ..r]
9   return
```

$b = 2$, and $d = 1$. Since $d = 1 = \log_2 2 = \log_b a$, we are in Case 2, and the closed solution is $T_{\text{MS}}(n) \leq O(n^d \log n) = O(n \log n)$.

In order to sort the whole array, call the recursive procedure for input $(1, n)$. This yields Algorithm 4.

Algorithm 4: procedure mergesort$(\texttt{A[1..}n\texttt{]})$

```
1 INPUT: A[1..n]
2 OUTPUT: A[1..n] contains the same objects as before, sorted in increasing
    order
3 DATA STRUCTURE: B[1..n], auxiliary array
4   if n > 1 then rmsort(1, n)
5   return
```

Remark: It is possible to show that on an input of length n, **mergesort** never makes more than $n \log n$ comparisons (note the absence of any multiplicative constants).

We sum up our observations regarding mergesort.

Theorem 2.3.1
Algorithm **mergesort** sorts a sequence of n elements in time $O(n \log n)$. The number of comparisons is smaller than $n \log n$. The algorithm needs extra space $O(n)$.

2.4 Medians (or: The selection problem)

Make sure you read pages 53–55 in the book!

Examples:
(a) Given is the sequence $[7, 15, 11, 9, 3]$. Find the fourth smallest entry ("the entry

with rank $k = 4$")! Entries 7, 9, 3 are smaller than 11, entry 15 is bigger than 11, so 11 is the fourth smallest entry.
(b) Given is the sequence $[7, 15, 11, 9, 7, 3, 7]$. Find the third smallest entry ("the entry with rank $k = 3$"). Entries 7, 7, 3, 7 are ≤ 7, and 15, 11, 9 are bigger than 7. So 7 is the answer.

The ***selection problem*** is the following: Given is an array `A[1..n]` with entries $a_1, \ldots, a_n$ from some ordered set (repetitions are allowed; in the examples, the entries will be numbers), as well as a number k from $\{1, \ldots, n\}$. We want to find the kth smallest entry in the array, or, technically, ***the element of rank k***. This is an entry a_i such that $a_j \leq a_i$ for *at least* k many indices j and $a_j > a_i$ for *at most* $n - k$ indices j.

Using the concept of sorting, we could alternatively say that if $b_1 \leq \cdots \leq b_n$ is $a_1, \ldots, a_n$ in sorted order, then b_k is the element of rank k. But for selection one does not need to sort.

Special cases: With $k = 1$ we are looking for the ***minimum***. With $k = n$ we are looking for the ***maximum***. With $k = \lceil n/2 \rceil$ we are looking for the ***median*** of the sequence $a_1, \ldots, a_n$, the element "in the middle".[6] The role of the median as an important statistical parameter in a sequence of measurements is explained in the book. (It can be considered a "typical" measurement, which is more robust with respect to outliers than the average.)

The obvious method to calculate the entry of rank k is to sort the input sequence $a_1, \ldots, a_n$. This takes time $\Theta(n \log n)$ with Mergesort (worst case). We aim at a procedure that takes linear time ("in expectation").

The idea, based on divide-and-conquer, is the following. It has great similarities with Quicksort (see Section 2.4x to follow), but also some differences, in particular we must use three-way-partitioning.

(0) Triviality test: If $n \leq n_0$, just sort the array `A[1..n]` by some simple procedure like **mergesort** and read off the result `A[`k`]`.

Assume now that $n > n_0$.

(1) Split/Divide: We choose a position s in $\{1, \ldots, n\}$ at random, and call the entry $x :=$ `A[`s`]` the ***splitter*** or ***pivot***. Then we compare all other entries of the array with x (at the cost of $n-1$ comparisons and $O(n)$ running time), and find out which entries are smaller than x, equal to x, and larger than x. The smaller entries go to subarray A_L, the larger entries to subarray A_R, the entries equal to x to subarray A_x. Actually, we rearrange the elements in A so that A_L, A_x, A_R just sit next to each other:

$$A_L = \texttt{A[1..}p_1 - 1\texttt{]}, A_x = \texttt{A[}p_1..p_2\texttt{]}, A_R = \texttt{A[}p_2 + 1..n\texttt{]}.$$

[6] In statistics, one often has a (long) sequence $a_1, \ldots, a_n$ of measurements in $\mathbb{R}$, and one needs the "α-quantile", which is the element of rank $\lceil \alpha n \rceil$, for $0 < \alpha \leq 1$.

Example: Assume $\texttt{A[1..}n\texttt{]} = [7, 15, 6, 11, 8, 9, 2, 7, 10, 3, 6]$ and $s = 3$, hence $x = \texttt{A[3]} = 6$. Then (e.g.)

$$A_L = \texttt{A[1..2]} = [3, 2], A_x = \texttt{A[3..4]} = [6, 6], A_R = \texttt{A[5..1]} = [7, 9, 15, 11, 10, 8, 7].$$

Note that the order inside the subarrays is arbitrary; it depends on the details of the procedure we use to split.

(2) Recursion: Now we need recursive calls. What are the subproblems? Actually, there is at most one subproblem. If $k < p_1$, we get the solution by calling the algorithm recursively on $A_L = \texttt{A[1..}p_1 - 1\texttt{]}$, with k. If $p_1 \le k \le p_2$, we are done and can output $\texttt{A[}k\texttt{]}$. If $p_2 < k$, we recursively look for the element of rank $k' = k - p_2$ in the subarray $A_R = \texttt{A[}p_2 + 1..n\texttt{]}$.

In the example: If $k = 2$, we use recursion for $[3, 2]$ and $k = 2$; if $k = 4$, we output $\texttt{A[}k\texttt{]} = 6$; if $k = 9$, we use recursion for $[7, 9, 15, 11, 10, 8, 7]$ and the new value $k' = 9 - 4 = 5$.

(3) Combine: There is nothing to do.

To simplify the program text, we do not generate new arrays for recursive calls, but rather pass the limits of the "current subarray" as arguments. This means the input for a recursive call consists of two indices ℓ and r with $1 \le \ell \le r \le n$ such that $\ell \le k \le r$. (This is what we did in **mergesort**.) In the procedure a splitter $x = \texttt{A[}s\texttt{]}$ is chosen, by randomly choosing s in $[\ell, r] = \{\ell, \ldots, r\}$. Then the subarray $\texttt{A[}\ell..r\texttt{]}$ is split into three parts

$$A_L = \texttt{A[}\ell..p_1 - 1\texttt{]}, A_x = \texttt{A[}p_1..p_2\texttt{]}, A_R = \texttt{A[}p_2 + 1..r\texttt{]}.$$

If $k < p_1$, we call the procedure recursively on A_L, if $p_1 \le k \le p_2$, we output $\texttt{A[}k\texttt{]}$, if $p_2 < k$, we call the procedure recursively on A_R. The method is given in pseudocode in Algorithm 5.

The details of the "partitioning" done in lines **9**–**13** of **rqselect** are not important. We just have to know that it takes $r - \ell$ comparisons and time $O(r - \ell)$. It can even be arranged that this procedure takes only constant extra space.

In order to solve the selection problem for $\texttt{A[1..}n\texttt{]}$ and k we call **rqselect**$(1, n, k)$, see Algorithm 6. It is almost obvious that the procedure gives the correct output.[7] The only question is how long it takes. (Only) For this analysis we make the assumption that the entries $a_1, \ldots, a_n$ are different. Note that this means that we always have $p_1 = p_2$, in all calls to **rqselect**.

It is possible to carry out an analysis similar to the one we will do for Quicksort, see Section 2.4x to follow. The result is that at most $4n$ comparisons are used and that

[7] If you want to prove it, use induction on recursive calls. The *induction hypothesis* is: All elements in $\texttt{A[1..}\ell - 1\texttt{]}$ are smaller than all elements in $\texttt{A[}\ell..r\texttt{]}$, and all elements in $\texttt{A[}\ell..r\texttt{]}$ are smaller than all elements in $\texttt{A[}r + 1..n\texttt{]}$, and $\ell \le k \le r$.

Algorithm 5: recursive Quickselect(ℓ, r, k)

```
1  function rqselect(ℓ, r, k)
2  INPUT: Numbers ℓ, r, k, 1 ≤ ℓ ≤ k ≤ r ≤ n, global array A[1..n]
3  OUTPUT: The element of rank k − ℓ + 1 in A[ℓ..r].
4  // Triviality check:
5    if r − ℓ ≤ n_0 then sort A[ℓ..r]; return A[k].
6  // Split:
7    Pick s at random from {ℓ, ..., r};
8    x ← A[s];
9    rearrange A[ℓ..r] so that
10     entries in A[ℓ..p_1 − 1] are smaller than x,
11     entries in A[p_1..p_2] are equal to x,
12     entries in A[p_2 + 1..r] are bigger than x,
13     for some ℓ ≤ p_1 ≤ p_2 ≤ r;
14 // possibly one recursive call:
15   Case k < p_1: return rqselect(ℓ, p_1 − 1, k);
16   Case p_1 ≤ k ≤ p_2: return A[k];
17   Case p_2 < k: return rqselect(p_2 + 1, r, k).
```

Algorithm 6: Quickselect$(\mathtt{A}[1..n], k)$

```
1  function quickselect(A[1..n], k)
2  INPUT: Array A[1..n] of objects on which an order is defined, number k,
                                                          1 ≤ k ≤ n
3  OUTPUT: The element of rank k in A[1..n].
4  METHOD: call the recursive procedure for the whole array:
5    return rqselect(1, n, k).
```

the expected running time is $O(n)$. We prefer to give a more direct analysis here, in which we can utilize the master theorem. The result is the following.

Theorem 2.4.1
The expected running time of **quickselect**(`A[1..n]`, k) is $O(n)$.

Remarks: Quickselect is very fast in practice, and a careful implementation of this algorithm should be (and is being) used for all reasonable statistical computations of quantiles of a sequence of measurements. Don't sort for this purpose! (Excepting if the number of elements is very small, or if many quantiles of the same sequence are needed.)

How does one *prove* the result stated in Thm. 2.4.1? To get a rough idea of what is going on, we first consider a totally unrealistic situation, namely that the randomly chosen element x happens to be the median of the current subarray all the time, i.e., that $p_1 = p_2 = \lceil (r - \ell + 1)/2 \rceil$ in all recursive calls. In this case we get the following recurrence for the running time $T(n)$ of **rqselect**(ℓ, r, k) with subarray length $n = r - \ell + 1$:

$$T(n) = \begin{cases} O(n \log n) & \text{for } n \le n_0 \\ T(\lfloor n/2 \rfloor) + O(n) & \text{for } n > n_0. \end{cases} \tag{2.11}$$

To make things simple, we assume $n_0 = 1$. (But don't implement **quickselect** with this choice!) Then we can apply the master theorem with $a = 1$, $b = 2$, $d = 1$. Since $\log_b a = \log_2 1 = 0 < 1 = d$, we are in Case 1, and get $T(n) = O(n^1) = O(n)$. This is a nice result, and it actually gives the best case for Quickselect. Unfortunately, we used the unrealistic assumption that the splitter always is the median, and this is by no means true.

Now we look at the real situation. Again we assume all entries in `A[1..n]` are different. We say the "partitioning" done for `A[`ℓ`..r]` in lines **9**–**13** is *bad* if the splitter is among the smallest $\lfloor \frac{1}{4}(r - \ell + 1) \rfloor$ entries or the largest $\lfloor \frac{1}{4}(r - \ell + 1 \rfloor)$ entries in `A[`ℓ`..r]`. Otherwise we call the partitioning "good".

Example: `A[`ℓ`..r]` $= [5, 3, 7, 6, 9, 2, 8, 1, 12, 4, 10]$. Then $r - \ell + 1 = 11$, and $\lfloor \frac{1}{4}(r - \ell + 1) \rfloor = \lfloor 11/4 \rfloor = 2$. Splitters 1, 2, 10, 12 give bad partitionings, splitter 2 for example leads to the very unbalanced partitioning $A_L = [1]$, $A_x = [2]$, $A_R = [5, 3, 7, 6, 9, 8, 12, 4, 10]$. The other elements as splitters lead to good partitionings.

If the partitioning is good, at least $\lfloor \frac{1}{4}(r - \ell + 1) \rfloor + 1 > \frac{1}{4}(r - \ell + 1)$ of the entries are in the subarray that is not considered in the recursive call, which means that the length of the subarray for which the procedure is called recursively is $\le \frac{3}{4}(r - \ell + 1)$. Since exactly $2\lfloor \frac{1}{4}(r - \ell + 1) \rfloor \le \frac{1}{2}(r - \ell + 1)$ elements as splitters out of $r - \ell + 1$ many lead to a bad partitioning, the probability that the partitioning is good is at least $\frac{1}{2}$.

Fact. Assume we toss a coin repeatedly until *heads* appears.
(a) If the coin is fair (probability $\frac{1}{2}$ for *heads* and for *tails*), the expected number of rounds until *heads* appears for the first time is 2.
(b) If the the coin is biased so that the probability for *heads* is at least $\frac{1}{2}$, the expected number of rounds until *heads* appears is ≤ 2.

This implies that if we start with $A[\ell..r]$, call **rqselect**(ℓ, r, k) and carry out recursive calls until a good partitioning happens, the expected number of recursive calls will be at most 2. Since the size of the subarrays dealt with here is $r - \ell + 1$ and smaller, the expected time for all this will be $O(r - \ell + 1)$.

Let $T(n)$ denote the maximum expected[8] time spent for **rqselect**(ℓ, r, k), if we start with a subarray $A[\ell..r]$ of length n or smaller. We get the following recurrence.

$$T(n) \leq \begin{cases} O(1) & \text{for } n = 1 \\ 1 \cdot T(\lfloor \frac{3}{4} n \rfloor) + O(n) & \text{for } n > 1. \end{cases} \tag{2.12}$$

Note: The "$O(n)$" in the case $n > 1$ is the expected computation time from the first call **rqselect**(ℓ, r, k) (with $n = r - \ell + 1$) up to and including to the first call **rqselect**(ℓ', r', k) where the partitioning is good.

As mentioned in Section 2.2, the master theorem also holds if $b > 1$ is not an integer. We read off parameters $a = 1$, $b = \frac{4}{3}$, $d = 1$, and with $\log_b 1 = 0 < 1 = d$ we see that we are in Case 1, which gives $T(n) = O(n^d) = O(n)$. This is Theorem 2.4.1. □

2.4x "The UNIX sort command" (Box on page 56)

Quicksort is another divide-and-conquer algorithm for sorting. Its main features are: It is fast in practice, in particular for sorting small items. It is randomized – the algorithm repeatedly chooses random numbers. Its *expected* running time is $O(n \log n)$. (Do not worry about its worst-case running time of $O(n^2)$. When implemented properly, this bad case has a probability so small that it will never occur.)

We want to sort an array `A[1..n]`. For simplicity we assume the entries are natural numbers, and that all input numbers are different. (See below for a comment about dealing with inputs with equal numbers.)

Quicksort is formulated as a recursive procedure **rqsort** ("recursive quicksort") with parameters ℓ and r, $1 \leq \ell, r \leq n$. The purpose of the call **rqsort**(ℓ, r) is to sort the subarray `A[`ℓ`..`r`]` (and never touch `A[1..`$\ell - 1$`]` and `A[`$r + 1$`..`n`]`).

It follows the divide-and-conquer paradigm.

[8] Of course this leads to an application of the master theorem this was not intended for. Fortunately the formulation of the master theorem is so general it can also be applied here.

(0) Triviality test: If $r \leq \ell$ then the subarray to be sorted is empty or has only one entry. We do nothing. (*Note*: This is obviously correct. Actually, in the implementation, we arrange things so that **rqsort**(ℓ, r) is not called at all if $r \leq \ell$.)

(1) Splitting: This is called "partitioning" in Quicksort. – Choose uniformly at random an index s from the set $[\ell, r] = \{\ell, \ell+1, \ldots, r\}$. Entry $x = \texttt{A[}s\texttt{]}$ is called the "splitter" or "partitioning element" or "pivot". Put entries smaller than x to the left, entries larger than x to the to the right, and x in between. More technical: The entries in $\texttt{A[}\ell..r\texttt{]}$ are rearranged so that for some $p \in [\ell, r]$ we have:

$$\texttt{A[}\ell\texttt{]}, \ldots, \texttt{A[}p-1\texttt{]} < x, \text{ and } \texttt{A[}p\texttt{]} = x, \text{ and } A\texttt{A[}p+1\texttt{]}, \ldots, \texttt{A[}r\texttt{]} > x.$$

Example: $\texttt{A[}4..10\texttt{]} = [5, 3, 8, 2, 10, 6, 5]$. The random experiment gives $s = 6$, so $x = \texttt{A[}6\texttt{]} = 8$ is the splitter. We rearrange ("partition"), and a possible result is $\texttt{A[}4..10\texttt{]} = [5, 3, 2, 6, 5, 8, 10]$ with $p = 9$ the new position of the splitter $x = 8$.

(2) Recursion: We call **rqsort** recursively on subarrays $\texttt{A[}\ell..p-1\texttt{]}$ and $\texttt{A[}p+1..r\texttt{]}$.

Remark: Actually, in the program, one checks whether $\ell < p-1$ and $\ell + 1 < r$, respectively, and carries out the respective recursive call only if it is necessary. This saves time since it avoids useless recursive calls.

(3) Combining: Do nothing.

Remark: We insert here the correctness proof, which is done by induction on the size $r - \ell + 1$ of the subarray to be sorted. We know that the algorithm works correctly in the base case $r \leq \ell$. If $\ell < r$ we have that by the induction hypothesis the recursive calls give the correct result. So $\texttt{A[}\ell..p-1\texttt{]}$ and $\texttt{A[}p+1..r\texttt{]}$ are sorted and all entries in $\texttt{A[}\ell..p-1\texttt{]}$ are smaller than $x = \texttt{A[}p\texttt{]}$ and all entries in $\texttt{A[}p+1..r\texttt{]}$ are greater than $x = \texttt{A[}p\texttt{]}$. But then the whole subarray $\texttt{A[}\ell..r\texttt{]}$ is sorted.

In order to sort $\texttt{A[}1..n\texttt{]}$ we check if $n \leq 1$ (in which case we do nothing), and in case $n > 1$ we call **rqsort**$(1, n)$.

We can now formulate the algorithm in pseudocode. We do not discuss the partitioning procedure in detail, since it is a little intricate, but we are not interested in the details here. One can formulate it in such a way that no superfluous key comparisons are made, only (1) extra space is used, time $O(r - \ell)$ and exactly $r - \ell$ comparisons are made. (All entries in the subarray excepting x are compared with x.)

Correctness is settled already. We analyze the running time.

The running time of a call **rqsort**(ℓ, r) is $\Theta(r - \ell)$, if one excludes the recursive calls; the number of comparisons in the partition procedure for $(\ell, r, \texttt{p})$ is exactly $r - \ell$.

Algorithm 7: partition$(\ell, r, \mathtt{p})$

```
// Can be called for 1 ≤ ℓ ≤ r ≤ n.
// Rearranges subarray A[ℓ..r], for "splitter" x := A[ℓ].
// Variable p is a return value, contains p at the end, ℓ ≤ p ≤ r.
// x in A[p] and A[ℓ], ..., A[p − 1] ≤ x and x < A[p + 1r], ..., A[r].
// Cost, if ℓ < r: exactly r − ℓ key comparisons; time Θ(r − ℓ).
```

Algorithm 8: rqsort(ℓ, r)

```
// "randomized quicksort", a recursive procedure.
// Can be called for 1 ≤ ℓ < r ≤ n, sorts A[ℓ..r].
  pick s at random from {ℓ, ℓ + 1, ..., r}
  interchange A[ℓ] and A[s]
  partition(ℓ, r, p)
  if ℓ < p − 1 then rqsort(ℓ, p − 1)
  if p + 1 < r then rqsort(p + 1, r)
```

The total running time is

$$\sum_{\substack{1\le\ell<r\le n\\ \textbf{rqsort}(\ell,r)\text{ is called}}} \Theta(r-\ell) = \Theta\left(\sum_{\substack{1\le\ell<r\le n\\ \textbf{rqsort}(\ell,r)\text{ is called}}} (r-\ell)\right) = \Theta(C),$$

where

$$C = \sum_{\substack{1\le\ell<r\le n\\ \textbf{rqsort}(\ell,r)\text{ is called}}} (r-\ell)$$

is the total number of comparisons carried out in the whole algorithm. (We used the summation rule for O notation.)

See Fig. 2.3 for an illustration of all the calls to **rqsort** that may happen. The total number of comparisons is the sum of the lengths (minus 1) of all appearing subarrays that have length > 1.

The first observation is the following. It may happen that in each call of **rqsort** the splitter is the smallest or the largest entry in the subarray. In this case there will be $n-1$ calls to **rqsort**, with $r-\ell = n-1$, then $n-2$, then $n-3$, ..., then 2, then 1.

Algorithm 9: rquicksort(A[1..n])

```
// "randomized Quicksort", sorts an array A[1..n]).
  if n > 1 then rqsort(1, n)
```

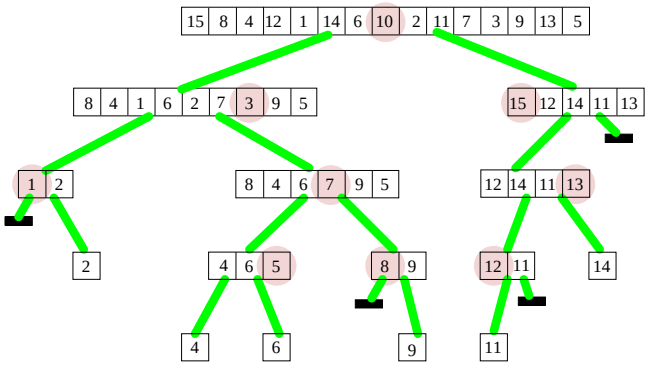

Figure 2.3: Quicksort action as a tree. Each array of length > 1 stands for a call to rqsort; the cost is the length of the array. Splitters are pink.

The overall number of comparisons is $\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$. This means that *in the worst case* the running time of Quicksort is $\Theta(n^2)$.

What is the *best case*? If in each situation the splitter splits the interval (of length $r - \ell + 1$) into two parts of length about $(r - \ell + 1)/2$, we get the same recurrence relation for the running time of **rqsort** on subarrays of length m as in Mergesort: $T(1)$ is a constant, and for $m > 1$ we have $T(m) \leq 2T(m/2) + O(m)$. The solution is $T(m) = O(m \log m)$ by the master theorem. Hence the call **rquicksort** on n entries will take time $\Theta(n \log n)$ in the best case.

We will now find out what the behavior of quicksort is *in expectation*. This means the following: The many random experiments done in choosing the splitters create a probability space, and the trees like in Fig. 2.3 vary according to the outcomes of these random experiments. The number C of comparisons then is a random variable, depending on the random choices of the algorithm. We set out to determine $\mathbf{E}(C)$, the *expectation* of C. (Since the running time of the algorithm is proportional to C, the *expected running time* is proportional to $\mathbf{E}(C)$.)

We analyze $\mathbf{E}(C)$ for the case where all entries are distinct. Let $b_1 < \cdots < b_n$ be the input elements in $A[1..n]$ in ascending order. (For the figures it is assumed that the entries in $A[1..15]$ actually are $1, \ldots, 15$. This is o.k., since the algorithm only compares numbers and does not care about the actual numbers. So in the figures we have $b_i = i$, for $i = 1, \ldots, 15$.)

Define random variables, for $1 \leq i < j \leq n$, of course also over the probability space created by the random choices of the algorithm:

$$X_{ij} = \begin{cases} 1, & \text{if } b_i, b_j \text{ are compared,} \\ 0, & \text{otherwise.} \end{cases} \tag{2.13}$$

Then $C = \sum_{1 \leq i < j \leq n} X_{ij}$. (Indeed: Each comparison that takes place is counted as 1.)

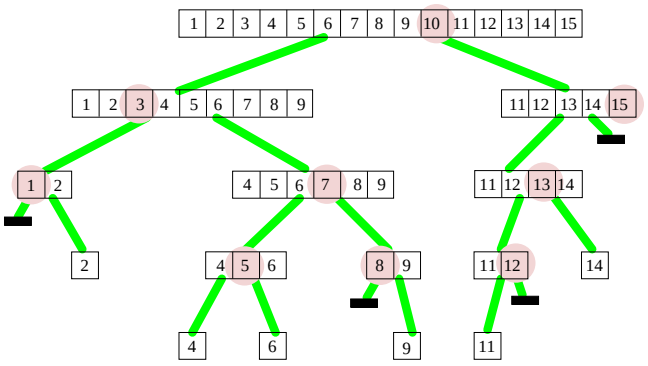

Figure 2.4: Quicksort action as a tree. Look at the ideal situation where the input is sorted (as $(b_1, \ldots, b_{15}) = (1, \ldots, 15)$). The probabilities are the same as in Fig. 2.3.

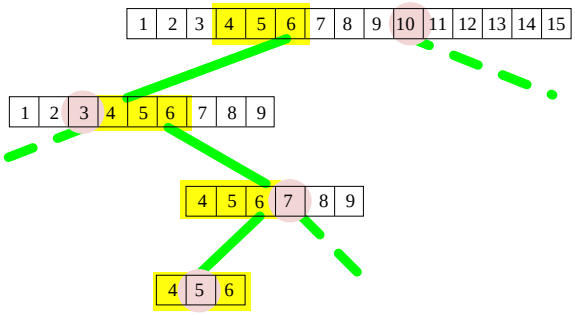

Figure 2.5: Quicksort action as a tree, idealized. $I_{4,6} = \{b_4, b_5, b_6\}$ (here $= \{4, 5, 6\}$). The probability that 4 and 6 are compared is $\frac{2}{|\{4,5,6\}|} = \frac{2}{3}$.

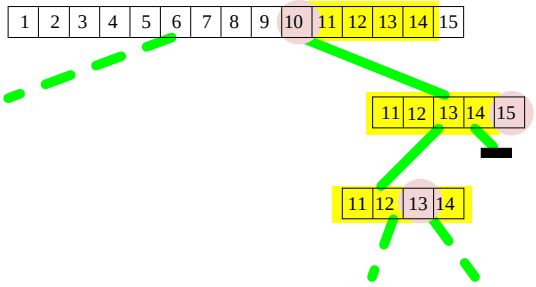

Figure 2.6: Quicksort action as a tree, idealized. $I_{11,14} = \{11, 12, 13, 14\}$. The probability that 11 and 14 are compared is $\frac{2}{|\{11,12,13,14\}|} = \frac{2}{4}$.

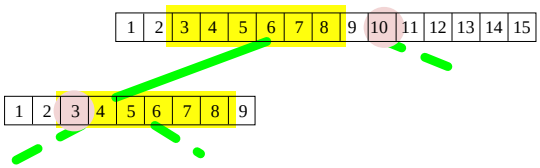

Figure 2.7: Quicksort action as a tree, idealized. $I_{3,8} = \{3, 4, 5, 6, 7, 8\}$. The probability that 3 and 8 are compared is $\frac{2}{|\{3,4,5,6,7,8\}|} = \frac{2}{6}$.

Hence (linearity of expectations):

$$\mathbf{E}(C) = \sum_{1 \le i < j \le n} \mathbf{E}(X_{ij}) = \sum_{1 \le i < j \le n} \mathbf{Pr}(X_{ij} = 1). \tag{2.14}$$

We need to find the value of $\mathbf{Pr}(X_{ij} = 1) = \mathbf{Pr}(b_i, b_j \text{ are compared})$, for $1 \le i < j \le n$.

For this, consider $I_{ij} = \{b_i, \ldots, b_j\}$. We observe this set in the course of the algorithm with its repeated partitioning into subarrays of $A[1..n]$. (For illustration, see Figures 2.4 to 2.7.[9]) At the beginning, all elements of I_{ij} are in the original big array $A[1..n]$. Now assume $A[\ell..r]$ contains all elements of I_{ij}, and **partition**$(\ell, r, \mathtt{p})$ is called. If the splitter x is smaller than b_i or larger than b_j, all elements of I_{ij} go to the same subinterval, and we keep on observing. Only when an element of I_{ij} is chosen as the splitter x, "something happens". If $x = b_i$ or $x = b_j$, the elements b_i and b_j are compared, if x is some element in between b_i and b_j, the partitioning by x places b_i in the left subarray and b_j in the right subarray, so they are never compared afterwards.

Since all elements of I_{ij} have the same chance of being picked as splitter (no matter what the current subarray looks like!):

$$\mathbf{Pr}(X_{ij=1}) = \mathbf{Pr}(b_i, b_j \text{ are compared}) = \frac{|\{b_i, b_j\}|}{|I_{ij}|} = \frac{2}{j - i + 1}.$$

[9] In the figures, the elements in the subarrays are drawn in increasing order. Since the splitter is chosen at random, the order of the elements in the subarray does not matter.

Plugging this into (2.14) and computing yields

$$\begin{aligned}\mathbf{E}(C) &= \sum_{1\le i<j\le n} \frac{2}{j-i+1} \\ &\le \sum_{1\le i<n} \sum_{i<j\le n} \frac{2}{j-i+1} \\ &\le \sum_{1\le i<n} \sum_{1\le k<n} \frac{2}{k+1} \\ &= 2(n-1)(H_n-1),\end{aligned}$$

where $H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ is the "n-th harmonic number". It is not hard to see that $\ln n < H_n < \ln n + 1$, for $n \ge 2$. (See Appendix.) Note that $2n\ln n = (2\ln 2)n\log_2 n$, where $2\ln 2 = 1.38629....$

Theorem 2.4.2
For randomized quicksort on n distinct input elements we have: The *expected* number of comparisons is smaller than $1.39n\log_2 n$; the *expected* running time is $O(n\log n)$.

Remarks:

- Quicksort is very fast in practice, hence it is heavily used, e.g. in the UNIX sort command or in standard algorithms libraries with C++ or Java.
- One should not worry about the worst case of $O(n^2)$. A more precise analysis shows that C is quite well concentrated around its mean and that with with probability $1-o(n)$ we will have no more than $cn\log n$ comparisons, as n gets large.
- Our algorithm formulation and the analysis assume that all entries are different. The possibility of equal entries requires special care. For example, if all entries are equal, and no care is taken, we could get quadratic running time. A standard method for dealing with this issue is that in partitioning one splits the subarray `A[`ℓ`..`r`]` into three parts, not two, with border points $p_1 \le p_2$ for the area taken by copies of the splitter, instead of one position p.

  $$\texttt{A[}\ell\texttt{]},\ldots\texttt{A[}p_1-1\texttt{]} < x, \text{ and } \texttt{A[}p_1\texttt{]} = \cdots = \texttt{A[}p_2\texttt{]} = x, \text{ and } \texttt{A[}p_2+1\texttt{]},\ldots,\texttt{A[}r\texttt{]} > x.$$

  Example: If `A[4..17]` $= [3,5,1,6,3,4,9,3,4,7,5,6,5,2]$, choosing $s=5$ (or $s=14$ or $s=16$) will give splitter $x=5$, and a possible result of partitioning is $[3,1,3,4,3,4,2,5,5,5,6,9,7,6]$ with $p_1=11$ and $p_2=13$.

  With this modification, the running time analysis also works well for the more general case with repeated elements.

- Sometimes one sees versions of Quicksort in which the splitters are not randomly chosen. Here one has to be careful with the (deterministic) choice of the position s of the splitter. For example, taking $s = \ell$ will always result in poor (quadratic) running times on inputs that are already sorted. The least one should do is to take $s = \lfloor(\ell + r)/2\rfloor$ as position of the splitter, or, better still, take the median of the three elements `A[`ℓ`]`, `A[`$\lfloor(\ell + r)/2\rfloor$`]`, and `A[`$r$`]` as splitter.

A lower bound for sorting

(Gray box on pages 52 and 53 in the book.)

We consider our standard question: Can we sort faster than in $O(n \log n)$ time, which is what we get with Mergesort (worst case) and Quicksort (expected case)? The answer is *no*, for a certain kind of sorting algorithms.

We say an algorithm that handles items with keys from an ordered domain (integers, real numbers, strings, etc.) is *comparison based* if the only operations it applies to items is comparing two keys (with $\overset{?}{<}$, $\overset{?}{>}$, $\overset{?}{=}$, and their negations $\overset{?}{\leq}$, $\overset{?}{\geq}$, $\overset{?}{\neq}$). Apart from that, items can be moved around and copied.[10]

Now let us assume we have a comparison-based algorithm $\mathcal{A}$ that sorts all n-tuples of objects. This algorithm can be transformed into a *comparison tree* that captures all the comparisons made by the algorithm, on all possible inputs. Actually, to get a clean picture, we choose as inputs sequences $(a_1, 1), (a_2, 2), \ldots, (a_n, n)$, where $a_1, \ldots, a_n$ are $1, \ldots, n$ in one of the $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ possible orders. This sequence of pairs is to be sorted according to the first components $a_1, \ldots, a_n$, the second component is just dragged along. The result is a sequence $(1, s_1), \ldots, (n, s_n)$.

Example: Input $(6, 1), (5, 2), (2, 3), (4, 4), (1, 5), (3, 6)$ is transformed into output $(1, 5)$, $(2, 3), (3, 6), (4, 4), (5, 2), (6, 1)$. Then $(s_1, \ldots, s_6) = (5, 3, 6, 4, 2, 1)$. Note that $a_i = j$ if and only if $s_j = i$, for $1 \leq i, j \leq 6$.

To get the tree, we let algorithm $\mathcal{A}$ run until the first key comparison occurs. It is $a_i \overset{?}{\diamond} a_j$ for some comparison operator $\diamond$. Since all keys in the input are different, we can ignore the possibility that $a_i = a_j$, which leaves us with comparators $<$ and $>$. By exchanging the cases if necessary, we can assume that the comparison is $a_i \overset{?}{<} a_j$ with $i < j$. We create a root with inscription $a_i \overset{?}{<} a_j$, and a right subtree that (recursively) constructs the tree from the rest of the algorithm under the assumption that $a_i < a_j$ is true and a left subtree constructed under the assumption that $a_i < a_j$ is wrong. When the algorithm stops and gives output $(1, s_1), \ldots, (n, s_n)$, we create a leaf and

[10] What is forbidden in comparison-based algorithms is e.g. assuming that the keys are numbers, so that they can be added or multiplied, or assuming keys are integers given as bitstrings, so that one can extract the most significant bit or use the key as an index in an array.

label it with $(s_1, \ldots, s_n)$.

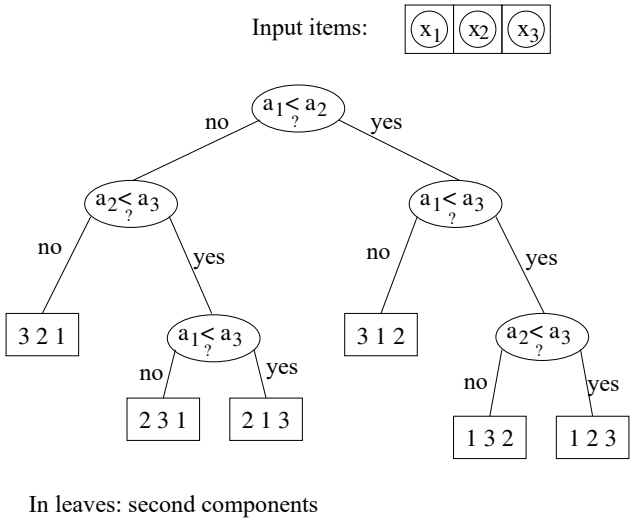

Figure 2.8: Sorting items $x_1 = (a_1, 1), x_2 = (a_2, 2), x_3 = (a_3, 3)$ according to the first component, with mergesort. We assume a_1, a_2, a_3 are different, so we do not worry about equal keys. All possible computations can be collected in a comparison tree or sorting tree. In the leaves we note the order of the second components when the algorithm stops.

In Figure 2.8 we have drawn the tree that results from running **mergesort** on three inputs $(x_1, x_2, x_3) = ((a_1, 1), (a_2, 2), (a_3, 3))$. (The input is split into segments (x_1, x_2) and x_3. First a_1 and a_2 are compared. If $a_1 < a_2$, sequences (x_1, x_2) and x_3 are merged, which leads to comparison $a_1 \stackrel{?}{<} a_3$ and, if necessary, $a_2 \stackrel{?}{<} a_3$ in the right subtree. If $a_1 > a_2$, then x_1 and x_2 are interchanged. Then (x_2, x_1) and x_3 are merged, which gives rise to the comparison $a_2 \stackrel{?}{<} a_3$ and, if necessary, $a_1 \stackrel{?}{<} a_3$, in the left subtree. We get $3! = 6$ leaves. Algorithm **mergesort** with 4 inputs will lead to a tree with $4! = 24$ leaves.

Note that every computation of the algorithm gives rise to a path in the tree from the root to a leaf. Different inputs induce paths to different leaves, since $s_i = j$ if and only if $a_j = i$, for $1 \leq i, j \leq n$. Thus there are exactly $n!$ leaves.

It is very easy to see that if a binary tree has *depth* (length of a longest path from the root to some leaf) d then it can have no more than 2^d leaves. From this we get that a tree with N leaves must have depth at least $\log_2 N$. With $N = n!$ we can conclude from what we had before that the tree for $\mathcal{A}$ has a path of length at least $\log_2(n!)$, or that on some input

$$\mathcal{A} \text{ makes at least } \log_2(n!) \text{ comparisons.}$$

Yes, fine, but what is $\log_2(n!)$? We note (for $n \geq 1$):

$$\frac{n^n}{n!} < \sum_{i \geq 0} \frac{n^i}{i!} = e^n,$$

hence $n! > (n/e)^n$. Taking logarithms (to the base 2), we get

$$\log(n!) > n \log n - n(\log e).$$

Note that $\log e$ is a constant of value ≈ 1.44269504.

Theorem 2.4.3
Let $\mathcal{A}$ be a comparison-based algorithm that sorts arbitrary n-tuples of objects. Then we have:

(i) There is an input on which $\mathcal{A}$ makes at least $\log(n!) > n \log(n) - 1.443n$ comparisons. Thus the running time of $\mathcal{A}$ is $\Omega(n \log n)$.

(ii) If all $n!$ permutations of $\{1, \ldots, n\}$ are equally likely, then the average number of comparisons made by $\mathcal{A}$ is at least $\log(n!)$.

(iii) If $\mathcal{A}$ uses randomization (like Quicksort in Section 2.4x), there is an input on which the expected number of comparisons made by $\mathcal{A}$ is at least $\log(n!)$.

Comparing this lower bound with the statement that Mergesort on n inputs never makes more than $n \log n$ comparisons, one sees that the number of comparisons of Mergesort is never more than $1.443n$ more than the minimum possible.

Remark: There are algorithms that can sort n (small) integers in $O(n)$ time. They use the integer keys as indices, thus are not comparison-based.

Appendix: Bounding H_n and $n!$

The harmonic numbers We defined $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$. In order to obtain an estimate for H_n in terms of more familiar quantities, we use an integration trick. Since $t \mapsto 1/t$ is monotonically decreasing in $[1, \infty)$, we have, for all $i \geq 2$:

$$\frac{1}{i} \leq \int_{i-1}^{i} \frac{dt}{t} \leq \frac{1}{i-1}.$$

We sum these inequalities over $2 \leq i \leq n$ and combine the integrals, to obtain:

$$H_n - 1 = \sum_{i=2}^{n} \frac{1}{i} \leq \int_1^n \frac{dt}{t} \leq \sum_{i=2}^{n} \frac{1}{i-1} = H_{n-1} = H_n - \frac{1}{n} < H_n.$$

The integral $\int_1^n \frac{dt}{t}$ equals $\ln n$. Hence:

$$H_n - 1 \leq \ln n \leq H_n, \text{ for all } n \geq 1.$$

We say that H_n is a "discrete version" of the natural logarithm, which is really a very fundamental function.

The following is *not* part of the mandatory material. Read it if you are interested.

The factorial function We defined $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. This is a very fast growing function. Sometimes it is advantageous to have good estimates for $n!$, as in the proof of the lower bound $n \log_2 n - n \log_2 e$ for sorting.

One can easily obtain quite sharp bounds for $n!$, correct up to a small constant factor. The trick is to compare the integral $\int_1^n \ln t \, dt$ with $\log(n!) = \sum_{1 \leq i \leq n} \ln i$.

Upper bound: $t \mapsto \ln t$ is a concave function, so if we take the linear interpolation between $(i-1, \ln(i-1))$ and $(i, \ln i)$, we are below $\ln t$ always. Integrating and summing we get:

$$\sum_{2 \leq i \leq n} \frac{\ln(i-1) + \ln i}{2} \leq \sum_{2 \leq i \leq n} \int_{i-1}^{i} \ln t \, dt = \int_1^n \ln t \, dt,$$

which (using $\ln 1 = 0$) means

$$\ln(n!) = \sum_{1 \leq i \leq n} \ln i \leq \int_1^n \ln t \, dt + \frac{\ln n}{2}.$$

The antiderivative $\int \ln t \, dt$ is $t(\ln t - 1) + C$, hence $\int_1^n \ln t \, dt = n(\ln n - 1) + 1$. By exponentiating we obtain

$$\boxed{n! \leq e n^{1/2} \left(\frac{n}{e}\right)^n = e\sqrt{n} \left(\frac{n}{e}\right)^n < 2.7183\sqrt{n} \left(\frac{n}{e}\right)^n.}$$

Lower bound: By concavity of $t \mapsto \ln t$ we know it runs below its tangent in every point. So

$$\int_{i-\frac{1}{2}}^{i+\frac{1}{2}} \ln t \, dt \leq \ln i, \text{ for } 1 < i < n,$$

and $\int_1^{\frac{3}{2}} \ln t \, dt = \frac{3}{2}(\ln(\frac{3}{2}) - 1) + 1$ and $\int_{n-\frac{1}{2}}^{n} \ln t \, dt \leq \frac{1}{2} \ln n$. Putting this together we get

$$n(\ln n - 1) + 1 = \int_1^n \ln t \, dt \leq \tfrac{3}{2}(\ln(\tfrac{3}{2}) - 1) + 1 + \ln(n!) - \frac{1}{2} \ln n,$$

hence

$$n(\ln n - 1) - \tfrac{3}{2}(\ln(\tfrac{3}{2}) - 1) + \frac{1}{2} \ln n \leq \ln(n!).$$

Exponentiating we get

$$\boxed{2.4395\sqrt{n}\left(\frac{n}{e}\right)^n \leq n!.}$$

These upper and lower bounds for the factorial function are already quite satisfying, since they agree up to a constant factor.

"Stirling's formula" determines the constant exactly and gives much more precise bounds:

$$\boxed{e^{\frac{1}{12n+1}} \leq \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} \leq e^{\frac{1}{12n}}.}$$

Note that $e^{\frac{1}{12n}} \approx 1 + \frac{1}{12n} \to 1$ for $n \to \infty$.