Chapter 2. Sorting

Sorting a set with respect to some ordering is a very frequently occuring problem. IBM estimates that about 25% of total computing time are spent on sorting in commercial computing centers. The most important applications are (according to Knuth (73)):

a) Collecting related things: In an airline reservation system one has to manipulate a set of pairs consisting of passenger name and flight number. Suppose that we keep that set in order of passenger name. Sorting according to flight number then gives us a list of passengers for each flight.

b) Finding duplicates: Suppose that we are given a list of 1000 persons and are asked to find out which of them are present in a group of 100 persons. An efficient solution is to create a list of the persons in the group, sort both lists and then compare them in a sequential scan through both lists.

c) Sorting simplifies searching as we will see in Chapter 3.

We next give a formal definition of the sorting problem. Given is a set of n **objects** $R_1, \ldots, R_n$. Each object R_i consists of a **key** (**name**) S_i and some **information** associated with that name. For example, the objects might be entries in a German-English dictionary, the keys are Germans words, and the information part is gender, English translation, ... An important fact is that the keys are drawn from some linearly ordered universe U; we use $\le$ to denote the linear order. In our example the ordering relation is the alphabetic order of words. We want to find a rearrangement $R_{i_1}, R_{i_2}, \ldots, R_{i_n}$ of the objects such that

$$S_{i_1} \le S_{i_2} \le \cdots \le S_{i_n}.$$

Apparently, the information associated with the keys does not play any role in the sorting process. We will therefore never mention it in our algorithms. One further remark should be added. Frequently, the information associated with the keys is much larger than the keys themselves. It is then very costly to move an object. Therefore it is better to replace the objects by records consisting of the key and a pointer to the object and to sort the set of records obtained in this way. Sometimes it will also suffice to only compute the permutation $i_1, i_2, \ldots, i_n$ instead of actually rearranging the set of objects. All algorithms described below are easily adapted to this modified sorting problem.

Let us reconsider example a) above. Suppose that we want for each flight the list of passengers in alphabetic order. Note that we start with a list of pairs sorted by passenger name. It would then be very nice if the sorting algorithm would not distroy that order when it rearranges the pairs according to flight number because then the passenger lists would be in alphabetic order automatically. More formally, we call a sorting algorithm *stable* if it produces a rearrangement $R_{i_1}, \ldots, R_{i_n}$ such that $S_{i_l} = S_{i_{l+1}}$ implies $i_l < i_{l+1}$ for all l. Note that we can always produce a stable rearrangemant by extending the keys to pairs (S_i, i) and by using the lexicographic ordering on these pairs as the linear order.

Sorting algorithms can be divided into two large groups: the algorithms of the first group are **comparison-based**, i.e., they make only use of the fact that the universe is linearly ordered. In these algorithms the *only* operation applied to elements of U is the comparison between elements. We will discuss four algorithms of the first kind: Heapsort which always runs in time $O(n \log n)$, Quicksort which runs in time $O(n \log n)$ on the average and is the algorithm with the smallest average running time, Mergesort which also runs in time $O(n \log n)$ and uses only sequential access and finally A-sort (Section 3.5.3.2) which works particularly well on almost sorted inputs. The algorithms of the first group are also called **general sorting algorithms** because they do not impose any additional restriction on the universe. We will also see that general sorting algorithms require $\Omega(n \log n)$ comparisons in the worst case as well as in the average case. Thus all algorithms mentioned above are optimal up to constant factors.

The algorithms of the second group only work for keys in a restricted domain. We treat sorting reals and sorting words according to lexicographic order.

Remark: For this chapter we extend our machine model by two instructions, which are very helpful for sorting algorithms. One instruction exchanges the contents of the accumulator and some storage cell.

$$\alpha \leftrightarrow op \qquad \text{and} \qquad \alpha \leftrightarrow mop.$$

The second instruction addresses a second general purpose register, which we denote by $\overline{\alpha}$. $\overline{\alpha}$ entspricht etwa dem Multiplikandenregister. We allow loads and stores from $\overline{\alpha}$ and simple arithmetic operations, which leave their result in α, e.g.,

$$\alpha \leftarrow \overline{\alpha} - op \qquad \text{and} \qquad \alpha \leftarrow \overline{\alpha} - mop.$$

Instructions of the second type allow us to make comparisons with $\overline{\alpha}$ without destroying its content. Both type of instructions are available on many machines.

2.1. General Sorting Methods

2.1.1. Sorting by Selection, a First Attempt

We begin with a very simple sorting method. Select the largest element from the input sequence, delete it and add it to the front of the output sequence. The output sequence is empty initially. Repeat this process until the input sequence is exhausted.

We formulate the complete algorithm in our high level programming language first and then translate it into machine code. Suppose that the input sequence is given by array $S[1\,..\,n]$.

```
(1)  $j \leftarrow n$;
(2)  while $j > 1$
     do co $S[j+1\,..\,n]$ contains the output in increasing order,
              $S[1\,..\,j]$ contains the remainder of the input sequence;
(3)       $k \leftarrow j$; max $\leftarrow j$; $S \leftarrow S[j]$;
(4)       while $k > 1$
          do co we search for the maximal element of $S[1\,..\,j]$,
                   $S = S[max]$ is always the largest element of $S[k\,..\,j]$ oc
(5)            $k \leftarrow k - 1$;
(6)            if $S[k] > S$
(7)            then $max \leftarrow k$; $S \leftarrow S[k]$ fi
          od;
(8)       $S[max] \leftarrow S[j]$; $S[j] \leftarrow S$;
(9)       $j \leftarrow j - 1$
     od.
```

Program 8

Next we translate into machine code. We use the following storage assignment. Index register γ_1 holds $j-1$, γ_2 holds $k-1$ and index register γ_3 holds $max - 1$. n is stored in location 0, S is stored in $\overline{\alpha}$ and array $S[1\,..\,n]$ is stored in locations $m+1, \ldots, m+n$. We assume that $n > 1$ and that the keys are integers. In the right column of Program 9 the number of executions of each instruction is given.

Lines 5, 6, 7, 10 are executed for $j = n, \ldots, 2$ and $k = j-1, \ldots, 1$. Hence $A = n(n-1)/2$. Furthermore $B \le A$. Thus the running time of Program 9 $\le 3 \cdot 1 + 10(n-1) + 5A + 3B = 2.5n^2 + 7.5n - 7 + 3B \le 4n^2 + 6n - 7$ time units in the unit cost measure.

Average case running time is slightly better. We compute it on the assumption that the keys are pairwise different and that all $n!$ permutations of keys $S_1, \ldots, S_n$ are equally likely. The first assumption is not crucial but simplifies calculations.

At each execution of the body of the outer loop we interchange S_{max} with S_j. Since there are j different possibilities for max, this step transforms j possible

0:	$\gamma_1 \leftarrow \rho(0)$	$j \leftarrow n$	1
1:	$\gamma_1 \leftarrow \gamma_1 - 1$	"	1
2:	$\gamma_2 \leftarrow \gamma_1$	$k \leftarrow j$	$n-1$
3:	$\gamma_3 \leftarrow \gamma_1$	$max \leftarrow j$	$n-1$
4:	$\overline{\alpha} \leftarrow \rho(m + \gamma_3 + 1)$	$S \leftarrow S[max]$	$n-1$
5:	$\gamma_2 \leftarrow \gamma_2 - 1$	$k \leftarrow k - 1$	A
6:	$\alpha \leftarrow \overline{\alpha} - \rho(m + \gamma_2 + 1)$	line (6)	A
7:	**if** $\alpha \geq 0$ **then goto** 10	"	A
8:	$\gamma_3 \leftarrow \gamma_2$	$max \leftarrow k$	B
9:	$\overline{\alpha} \leftarrow \rho(m + \gamma_3 + 1)$	$S \leftarrow S[max]$	B
10:	**if** $\gamma_2 > 0$ **then goto** 5	line (4)	A
11:	$\overline{\alpha} \leftrightarrow \rho(m + \gamma_1 + 1)$	line (8)	$n-1$
12:	$\rho(m + \gamma_3 + 1) \leftarrow \overline{\alpha}$	"	$n-1$
13:	$\gamma_1 \leftarrow \gamma_1 - 1$	$j \leftarrow j - 1$	$n-1$
14:	**if** $\gamma_1 > 0$ **then goto** 2	line (2)	$n-1$

Program 9

arrangements of keys $S_1, \ldots, S_j$ into *one* arrangement. Hence after the exchange all possible permutations of keys $S_1, \ldots, S_{j-1}$ are equally likely. Therefore we always have a random permutation of keys $S_1, \ldots, S_j$ during the sorting process.

Sequence $S_1, \ldots, S_j$ is scanned from right to left. Instruction 8 of the RAM program is executed whenever a new maximum is found. For example, if 4 5 3 1 2 is sequence $\{S_1, \ldots, S_j\}$ (we assume w.l.o.g. that $\{S_1, \ldots, S_j\} = \{1, \ldots, j\}$; this convention facilitates the discussion), then the maximum changes twice: from 2 to 3 and then to 5. Key S_j is equal to i, $1 \leq i \leq j$, with probability $1/j$. If $S_j = j$ then the maximum is found. If $S_j = i < j$ then we may delete elements $1, \ldots, i-1$ from the sequence. They certainly do not change the maximum. Thus we are left with $j - 1 - (i - 1) = j - i$ keys out of $S_1, \ldots, S_{j-1}$.

The maximum changes once plus the times it will change in the remaining sequence. This is a random permutation of the elements $i + 1, \ldots, j$.

This gives the following recursion for a_j, the expected number of changes of the maximum.

$$a_1 = 0 \qquad \text{and}$$

$$a_j = \frac{1}{j} \sum_{i=1}^{j-1} (1 + a_{j-i}) \quad \text{for } j > 1.$$

Multiplying by j gives

$$j \cdot a_j = (j-1) + \sum_{i=1}^{j-1} a_i$$

and similarly for $j+1$

$$(j+1)a_{j+1} = j + \sum_{i=1}^{j} a_i.$$

Subtraction yields

$$(j+1)a_{j+1} - j \cdot a_j = 1 + a_j.$$

or

$$a_{j+1} = a_j + 1/(j+1).$$

Thus

$$a_j = \sum_{i=2}^{j} 1/i = H_j - 1,$$

where $H_j = \sum_{i=1}^{j} 1/i$ is the j-th harmonic number (cf. Appendix). We have thus shown that instructions 8 and 9 are executed a_{j_0} times on the average when the outer loop is executed with $j = j_0$. Hence

$$\begin{aligned} B = \sum_{j=2}^{n} a_j = \sum_{j=1}^{n} H_j - n &= (n+1)H_n - 2n && \text{(cf. Appendix)} \\ &\le (n+1)\ln n - (n-1). && (H_n \le 1 + \ln n, \text{ cf. Appendix}) \end{aligned}$$

Average case running time of our algorithm is $2.5n^2 + 3(n+1)\ln n + 4.5n - 4 = O(n^2)$ Worst case and average running times are quadratic. It is also easy to see that the algorithm uses exactly $n(n-1)/2$ comparisons between keys.

2.1.2. Sorting by Selection: Heapsort

Is the algorithm described in the preceding section good? Can we improve upon the method for finding the maximum? The answer to the second question is no, in any case if we confine ourselves to comparison-based algorithms. In **comparison-based** algorithms there is no operation other than the comparison of two elements which is applicable to elements of the universe from which the keys are drawn.

Theorem 1. *Any comparison-based algorithm needs at least $n-1$ comparisons to find the maximum of n elements.*

Proof: Interpret a comparison $S_i < S_j$? as a match between S_i and S_j. If $S_i < S_j$ then S_j is the winner and if $S_i > S_j$ then S_i is the winner. If an algorithm uses less than $n-1$ matches (comparisons) then there is at least two players (keys) which are unbeaten at the end of the tournament. Both players could still be best (the maximum), contradiction. ∎

Theorem 1 implies that we have to look for a different sorting method if we want to improve upon the quadratic running time of the naive algorithm. Consider that algorithm on input 4 5 3 1 2. We compare 2 with 1, 2 with 3, 3 with 5 and finally 5 with 4. We can summarize our knowlegde at the end of the first maximum selection in Figure 11.

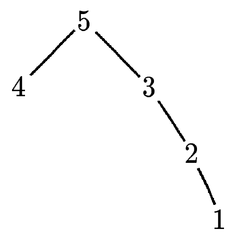


Figure 11. Known size relations

A parent-child relation represents the "greater than" relation on the labels of the nodes. Next we exchange 5 and 2 and obtain 4 2 3 1. At the second iteration we compare 1 with 3, 3 with 2, 3 with 4 and obtain Figure 12.

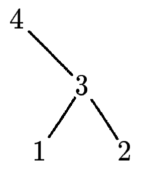


Figure 12. After the second iteration

Two of these comparisons were unnecessary. We knew already from the first iteration that 3 is larger than 1 and 2. Let us take a more global look: At the end of the first iteration we have a tree-like structure with the following property: Whenever one considers a path through that tree then the labels along the path are monotonically decreasing. This is also called the **heap property**.

From Figure 11 after removing the 5 we obtain Figure 13.

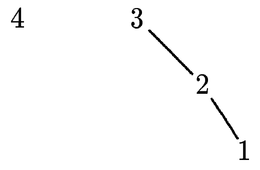


Figure 13. After removal of the 5

It suffices now to compare 4 and 3. This gives that 4 is larger than 3 and the second selection of a maximum is completed.

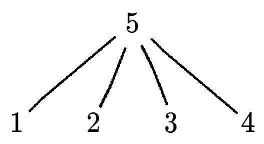


Figure 14. New example after the first iteration

Let us consider another example, the sequence 1 2 3 4 5. We obtain Figure 14 at the end of the first iteration.

After removing 5 we have 4 uncorrelated elements, i.e., we have not saved any information from the first iteration to the second. We can see from these examples that the trees we build should have small fan-out. In this way only little information is lost when a maximum is removed. Let us consider one more example, the sequence 2 10 3 5 1 7 9 6 4 8. The "ideal" tree has the form from Figure 15.

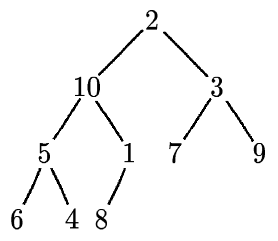


Figure 15. Ideal tree for the third example

Unfortunately, the labelling does not satisfy the heap property yet. We can install the heap property in a bottom-up process. In the tree shown in Figure 16 the three bottom levels of the tree already satisfy the heap property but the root label does not satisfy it yet.

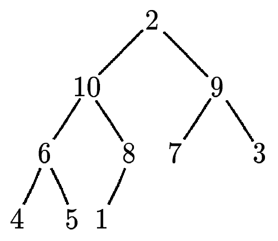


Figure 16. Only the root violates the heap property

We sink the 2 to its proper position by interchanging it with the larger of its children and obtain Figure 17.

In this tree the 2 is still not larger than its two children and so we have to interchange it again with the larger of its children (the 8).

We obtain in Figure 18 a tree having the heap property. So the maximum is the root label. We remove it and obtain Figure 19.

How should we restore the heap property? We might compare the 8 with the 9 and move up the 9. Next we compare the 7 with the 3 and obtain Figure 20.

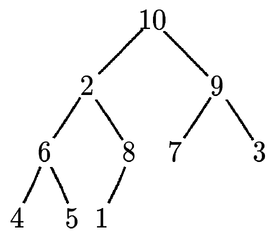


Figure 17. The root is greater than both children

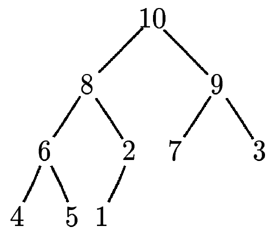


Figure 18. Whole tree satisfies the heap property

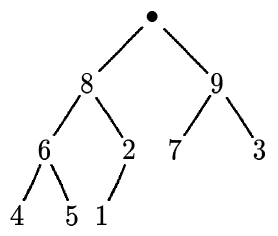


Figure 19. After removal of the root

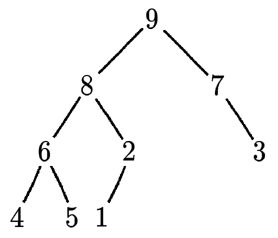


Figure 20. Heap is restored

We can now remove the 9 and repeat the process. This gives us Figure 21.

Although this approach leads to an $O(n \log n)$ sorting method we face one problem. After a few iterations the tree looses its perfect shape. There is another approach which avoids this problem. After removal of the 10 we take the 1, the label of the rightmost leaf on the bottom level, and move it to the root. Next we sink the 1 as described above. This method produces the sequence of trees shown in Figure 22.

In this way we always keep the tree as an almost complete binary tree. All

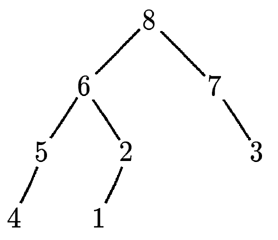


Figure 21. After removal of the 9

leaves are on two adjacent levels, all levels except the bottom level are complete, and the leaves on the bottom level are as far to the left as possible. Trees of this form can be stored in very compact form: Number the nodes starting at the root (with number 1) according to increasing depth and for fixed depth from left to right. Then the children of node k are numbered $2k$ and $2k+1$ and the parent of node k is numbered $\lfloor k/2 \rfloor$ ($\lfloor x \rfloor$ is the largest integer $\leq x$). Thus we can store the tree in a single array and save the space for pointers.

Definition: An array $S[1\,..\,n]$ satisfies the **heap property**, if

$$S\big[\lfloor k/2 \rfloor\big] \geq S[k] \quad \text{for } 2 \leq k \leq n.$$

The array is a **heap starting at l**, $1 \leq l \leq n$, if

$$S\big[\lfloor k/2 \rfloor\big] \geq S[k] \quad \text{for } l \leq \lfloor k/2 \rfloor < k \leq n. \qquad \blacksquare$$

Notice that every array $S[1\,..\,n]$ is a heap starting at $\lfloor n/2 \rfloor + 1$. The tree in Figure 23 corresponds to array 2 10 9 5 8 7 3 6 4 1 and is a heap starting at position 2.

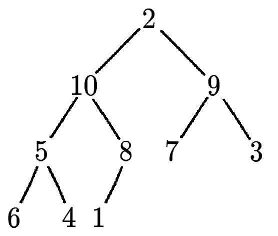


Figure 23. Tree with sequence 2 10 9 5 8 7 3 6 4 1

Next we give the complete implementation for Heapsort; we assume $n \geq 2$ in Program 10;

Line (10) is executed at each iteration of the inner loop and the inner loop is entered at least once at each iteration of the outer loop. Hence total running time is proportional to the number of comparisons in line (10). We count the number of comparisons. For simplicity, let $n = 2^k - 1$ for some $k \geq 1$.

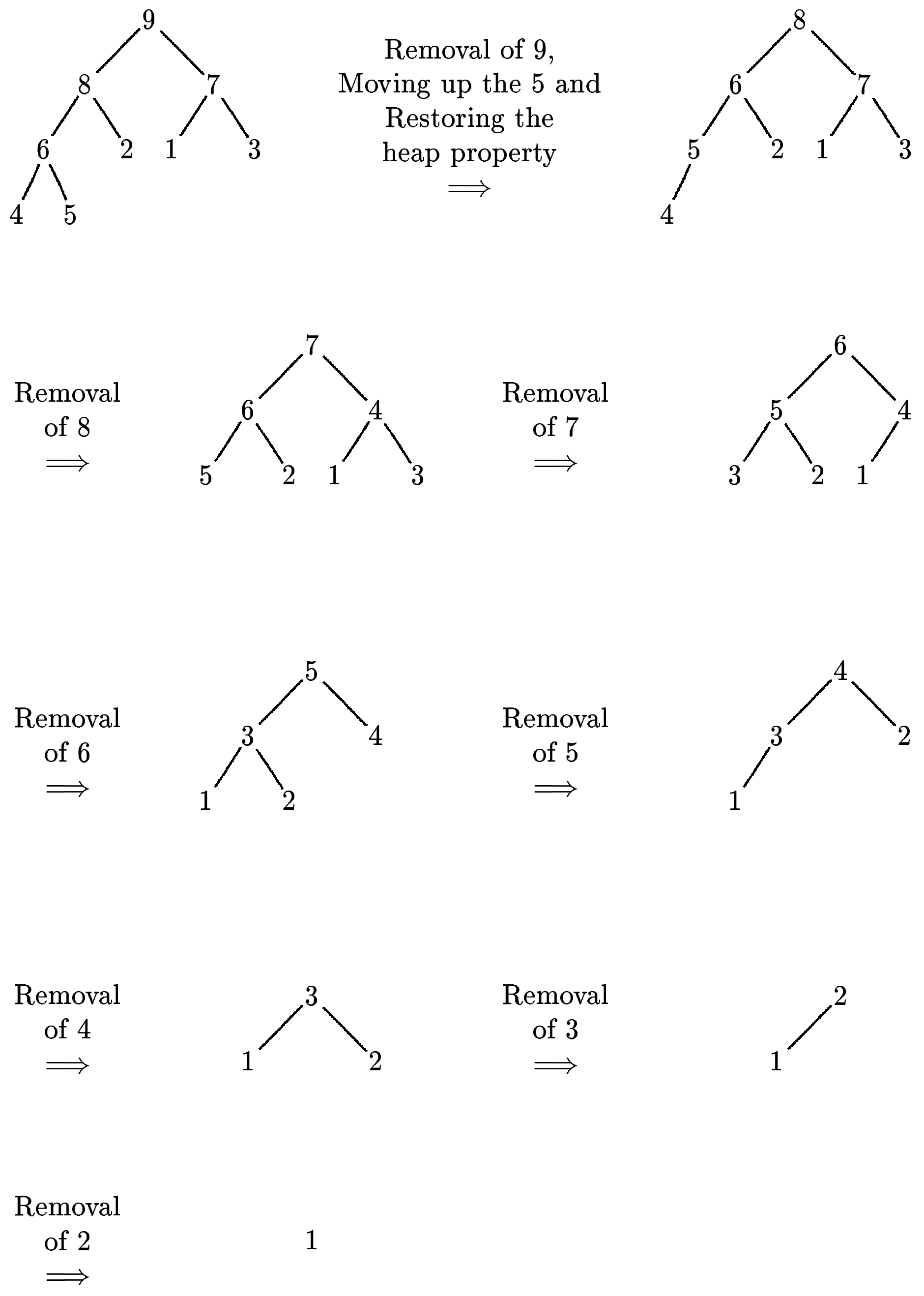


Figure 22. Selection phases

Build-up Phase : The tree has 2^i nodes of depth i, $0 \leq i < k$. In the build-up phase we add the nodes of depth $k-2$ to the heap, then the nodes of depth $k-3, \ldots$. When we add a node at level i then it can sink down to level $k-1$ for a cost of 2

```
(1)  $l \leftarrow \lfloor n/2 \rfloor + 1$; $r \leftarrow n$;
(2)  while $r \geq 2$
     do co either $r = n$ and then $S[1\,.\,.\,n]$ is a heap
             starting at $l$ (build up phase) or
             $l = 1$ and the $n - r$ largest elements are stored in increasing
             order in $S[r+1], \ldots, S[n]$ (selection phase) oc
(3)       if $l > 1$
          then co we are building the heap and
                  add the element $S[l-1]$ oc
(4)            $l \leftarrow l - 1$; $j \leftarrow l$
          else  co we are in the selection phase. $S[1]$ is the maximum of
                  $S[1], \ldots, S[r]$. We exchange $S[1]$ and $S[r]$ and have to
                  restore the heap property on $S[1\,.\,.\,r-1]$ oc
(5)            exchange $S[1]$ and $S[r]$; $r \leftarrow r - 1$; $j \leftarrow 1$
          fi;
(6)       $S \leftarrow S[j]$;
(7)       while $2j \leq r$
          do co we sink $S[j]$ by interchanging it
                 repeatedly with the larger of its children.oc
(8)          $k \leftarrow 2j$;
(9)          if $k < r$ and $S[k] < S[k+1]$ then $k \leftarrow k + 1$ fi;
(10)         if $S < S[k]$
(11)         then $S[j] \leftarrow S[k]$; $j \leftarrow k$
(12)         else  goto E fi
          od;
(13) E: $S[j] \leftarrow S$;
     od.
```

Program 10

comparisons per level. Thus the total cost of the build-up phase is bounded by

$$\sum_{i=0}^{k-2} 2(k-1-i)2^i = 2^{k+1} - 2(k+1). \qquad \text{(cf. Appendix, Formula S1)}$$

Selection phase : In the selection phase we repeatedly remove the root and move up element $S[r]$. This element can sink down all the way to the bottom level of the tree. More precisely, if an element of depth i is moved up then restoring the heap property can cost us up to $2i$ comparisons. Thus the total number of comparisons in the selection phase is at most

$$2\sum_{i=0}^{k-1} i \cdot 2^i = 2(k-2)2^k + 4. \qquad \text{(cf. Appendix, Formula S1)}$$

Altogether, Heapsort uses at most $2k(2^k - 1) - 2(2^k - 1) = 2n\log(n+1) - 2n$ comparisons. For $n \neq 2^k - 1$ counting is slightly more complicated.

Theorem 2. *Heapsort sorts n elements in at most $2n\log(n+1)-2n$ comparisons and time $O(n\log n)$.* ▮

By analogy to sorting by selection from 2.1.1, compiling into RAM code gives us a running time of $\le 20n\log(n+1)-n-7$ time units in the worst case. Average running time of Heapsort is only slightly better because of the following two reasons. We put new elements into the root and sink them to their proper position. Since most nodes of a binary tree are close to the leaves most elements will sink almost down to the leaves. Furthermore, in the selection phase we sink elements which we get from a leaf and which hence tend to be small. A complete average case analysis of Heapsort still needs to be done.

2.1.3. Sorting by Partitioning: Quicksort

Quicksort is an example of a very powerful problem solving method: **divide and conquer**. A problem is split into several smaller parts (divide) which are then solved using the same algorithm recursively (conquer). Finally the answer is put together from the answers to the subproblems.

In the case of **Quicksort** this means: Choose an element from sequence $S_1,\ldots,S_n$, say S_1, and partition the sequence into two parts. The first subsequence consists of all S_i with $S_i < S_1$ and and no S_j with $S_j > S_1$ and the second subsequence consists of all S_j with $S_j > S_1$ and and no S_i mit $S_i < S_1$. The partitioning algorithm stores the first sequence in positions 1 through $k-1$ of an array, S_1 in position k, and the second sequence in positions $k+1,\ldots,n$. Then the same algorithm is applied recursively to the two subsequences. Putting the sorted subsequences and the partitioning element S_1 together to a single sorted sequence is trivial. Nothing has to be done. One could think of splitting the sequence into three parts; the third part consisting of all $S_i = S_1$. In general, this is not worth the additional effort (cf. Exercise 8).

Next we take a closer look at the partitioning phase. A good solution encloses array $S[1\mathinner{..}n]$ with two additional elements $S[0]$ und $S[n+1]$ such that $S[0]\le S[i]\le S[n+1]$ for all i and uses two pointers with starting values 1 and $n+1$. Pointers i and k have the following meaning: the first subproblem consists of $S[2],\ldots,S[i]$, the second subproblem consists of $S[k],\ldots,S[n]$, and elements $S[i+1],\ldots,S[k-1]$ still have to be distributed. We increase i until we find $S[i]\ge S[1]$ and we decrease k until $S[k]\le S[1]$. Note that no test for "index out of bounds" is required because of the addition of elements $S[0]$ and $S[n+1]$. Two cases can arise. If $k>i$, then we exchange $S[k]$ and $S[i]$ and repeat the process. If $k\le i$, we are done. It is easy to see that either $k=i$ (if $S[i]=S[1]$) or $k=i-1$ (if $S[i]\ne S[1]$). Interchanging $S[k]$ and $S[1]$ completes partitioning. Program 11 uses this idea.

A remark on the correctness of the algorithm should be made. In line (3) and (4) we assume the existence of keys $S[l-1]$ and $S[r+1]$ with $S[l-1]\le S[i]\le S[r+1]$ for all i with $l\le i\le r$. Existence of these keys is ensured for call *Quicksort*$(1,n)$ by

```
    procedure Quicksort(l, r);
    co Quicksort(l, r) sorts the subarray $S[l] \dots S[r]$
        into increasing order oc
(1) begin $i \leftarrow l$; $k \leftarrow r + 1$; $S \leftarrow S[l]$;
(2) while $i < k$
(3) do repeat $i \leftarrow i + 1$  until $S[i] \geq S$;
(4)     repeat $k \leftarrow k - 1$ until $S[k] \leq S$;
(5)     if $k > i$ then exchange $S[k]$ and $S[i]$ fi
    od;
(6) exchange $S[l]$ and $S[k]$;
(7) if $l < k - 1$ then Quicksort($l, k - 1$) fi;
(8) if $k + 1 < r$ then Quicksort($k + 1, r$) fi
    end.
```

Program 11

adding $S[0]$ and $S[n+1]$ with the required properties, for recursive calls it is obvious from the description of the algorithm. Since we only have $S[l-1] \leq S[i] \leq S[r+1]$ we have to test for $\geq$ and $\leq$ in lines (3) and (4) in order to save the test for index out of bounds.

Worst case behavior of Quicksort is easily determined. Consider lines (1) through (6) for fixed values of l and r, say $l = 1$ und $r = n$. In line (3) we compare and increase i, in line (4) we compare and decrease k. Before entering the loop in line (2) we have $k - i = r - l + 1 = n$, after termination of the loop we have $k - i = 0$ or $k - i = -1$. Hence k and i are changed for a total of n or $(n+1)$-times and n or $n + 1$ comparisons are made. Actual running time of lines (2) to (5) is $O(\text{number of comparisons}) = O(n)$. The cost of lines (1) and (6)–(8) is $O(1)$ per call of Quicksort. Since there are $\leq n - 1$ calls altogether including the call in the main program (cf. Exercise 6) the total cost of lines (1) and (6)–(8) summed over all calls is $O(n)$.

Let us return to the maximal number $QS(n)$ of key comparisons which are needed on an array of n elements. We have

$$QS(0) = QS(1) = 0 \qquad \text{and}$$
$$QS(n) \leq n + 1 + \max_{1 \leq k \leq n} \{QS(k-1) + QS(n-k)\} \quad \text{for } n \geq 2.$$

It is easy to show by induction that $QS(n) \leq (n+1)(n+2)/2 - 3$ for $n > 0$. This bound is sharp as can be seen from the example $1, 2, \dots, n$. Here the partitioning phase splits off exactly one element and therefore subproblems of size $n, n-1, n-2, \dots, 2$ have to be solved. This requires $(n+1) + n + (n-1) + \cdots + 3$ comparisons which sums exactly to the bound given above. Quicksort's worst case behavior is quadratic.

Average case behavior is much better. We analyze it on the assumption that keys are pairwise distinct and that all permutations of the keys are equally likely.

We may then assume w.l.o.g. that the keys are the integers $1, \ldots, n$. Key S_1 is equal to k with probability $1/n$, $1 \leq k \leq n$. Then subproblems of size $k-1$ and $n-k$ have to be solved recursively and these subproblems are again random sequences, i.e., they satisfy the probability assumption postulated above. This can be seen as follows. If $S_1 = k$ then array S is as follows just prior to execution of line (6):

$$k \; i_1 \; i_2 \; \ldots \; i_{k-1} \; j_{k+1} \; j_{k+2} \; \cdots \; j_n.$$

Here $i_1, \ldots, i_{k-1}$ is a permutation of integers $1, \ldots, k-1$ and $j_{k+1}, \ldots, j_n$ is a permutation of integers $k+1, \ldots, n$.

What did the array look like before the partitioning step? If l interchanges occur in line (5) then there are l positions in the left subproblem, i.e., among array indices $2, \ldots, k$, and l positions in the right subproblem, i.e., among $k+1, \ldots, n$, such that the entries in these positions were interchanged pairwise, namely the leftmost selected entry in the left subproblem with the rightmost selected in the right subproblem, ... Thus there are exactly

$$\sum_{l \geq 0} \binom{k-1}{l} \binom{n-k}{l}$$

arrays before partitioning which produce the array by the partitioning process. The important fact to observe is that this expression only depends on k but not on the particular permutations $i_1, \ldots, i_{k-1}$ and $j_{k+1}, \ldots, j_n$. Thus all permutations are equally likely, and hence the subproblems of size $k-1$ and $n-k$ are again random. Let $QS_{av}(n)$ be the expected number of comparisons on an input of size n. Then

$$QS_{av}(0) = QS_{av}(1) = 0$$

and

$$\begin{aligned} QS_{av}(n) &= \frac{1}{n} \sum_{k=1}^{n} (n + 1 + QS_{av}(k-1) + QS_{av}(n-k)) \\ &= n + 1 + \frac{2}{n} \sum_{k=0}^{n-1} QS_{av}(k) \qquad \text{for } n \geq 2. \end{aligned}$$

We solve this recurrence by the method already used in 2.1.1. Multiplication by n gives us

$$n \cdot QS_{av}(n) = n(n+1) + 2 \sum_{k=0}^{n-1} QS_{av}(k) \qquad \text{for } n \geq 2.$$

Subtracting from the equality for $n+1$ instead of n, yields

$$QS_{av}(n+1) = 2 + \frac{n+2}{n+1} \cdot QS_{av}(n).$$

This recurrence is of the form

$$\begin{aligned} a_1 &= b_1 \\ a_{n+1} &= b_{n+1} + c_{n+1} a_n \end{aligned}$$

with $b_1 = 0,\ b_i = 2$ and $c_i = (i+1)/i$ for $i > 1$ and has solution

$$a_{n+1} = \sum_{i=1}^{n+1} \left(\prod_{j=i+1}^{n+1} c_j \right) b_i,$$

as is easily verified by induction. Thus

$$\begin{aligned} QS_{av}(n) &= \sum_{i=2}^{n} \frac{n+1}{i+1} \cdot 2 \\ &= 2(n+1)(H_{n+1} - 3/2) \\ &\le 2(n+1)\ \ln(n+1), \end{aligned}$$

where H_{n+1} is the $(n+1)$-th harmonic number (cf. Appendix). Let us return to running time. We argued above, that the running time of the partitioning phase is proportional to the number of comparisons and that the total cost of all other operations is $O(n)$. Thus

Theorem 3. *Quicksort sorts n elements with at most $(n^2 + 3n - 4)/2$ key comparisons and running time $O(n^2)$ in the worst case. It uses $\le 2(n+1)\ln(n+1)$ comparisons and time $O(n \log n)$ on the average.* ∎

Translation in RAM code as described in 1.5 produces a program which sorts n numbers in expected time $\le 13(n+1)\ln(n+1) + 29n - 33 \approx 9(n+1)\log(n+1) + 29n - 33$ (cf. Exercise 7).

Quicksort has quadratic worst case behavior; the worst case behavior occurs on the completely sorted sequence. Also almost sorted sequences, which occur frequently in practice, are unsuited for Quicksort. There is an interesting way out of this dilemma: randomized Quicksort. We change the algorithm by replacing line (1) by

```
(1a) begin $i \leftarrow l$; $k \leftarrow r+1$;
(1b)         $j \leftarrow$ random element of $[0, \ldots, r-l]$;
(1c)         interchange $S[l]$ and $S[l+j]$;
(1d)         $S \leftarrow S[l]$;
```

What does that do for us? Let Π be any permutation of numbers $1, \ldots, n$ and let $QS_{ran}(\Pi)$ be the expected number of comparisons in randomized Quicksort applied to sequence $\Pi(1), \Pi(2), \ldots, \Pi(n)$. In lines (1a)–(1d) we choose a random element of Π as the partitioning element, i.e., $S = S[h]$ with probability $1/n$ for $1 \le h \le n$. Then subproblems Π_1 and Π_2 of size $S[h]-1$ and $n - S[h]$ respectively have to be solved recursively. Of course, Π_1 and Π_2 depend on Π and h $(= l + j)$. We write $\Pi_1(\Pi, h)$ and $\Pi_2(\Pi, h)$ in order to make the dependence explicit. Then

$$QS_{ran}(\Pi) = \frac{1}{n}\sum_{h=1}^{n}\Big(n + 1 + QS_{ran}(\Pi_1(\Pi, h)) + QS_{ran}(\Pi_2(\Pi, h))\Big)$$

It is now trivial to prove that $QS_{ran}(\Pi) = QS_{av}(n)$ where n is the number of elements in permutation Π (use induction on n). Hence $QS_{ran}(\Pi) \le 2(n+1)\ln(n+1)$ for any permutation Π of n elements, i.e., randomized Quicksort sorts any *fixed* sequence in expected time $O(n \log n)$.

The reader should for a moment think about the meaning of this sentence. Standard Quicksort has average running time $O(n \log n)$. When deriving this result we postulated a distribution on the set of inputs, and the user of Quicksort has to behave according to that postulate if he wants to observe small running times. If the user mostly deals with nearly sorted sequences he should better keep away from standard Quicksort. Not so for randomized Quicksort. Expected running time is $O(n \log n)$ on every single problem instance (and worst case running time is $O(n^2)$ on every single problem instance). There are no bad inputs for randomized Quicksort, randomized Quicksort behaves the same on all input sequences.

At this point it is worthwhile to consider divide and conquer strategies in more detail. Let us assume, that a problem of size $n \ge b^{-1}(1)$ is split into $a(n)$ problems of size $b(n)$ and that the cost of dividing into subproblems and pasting together the answers is $f(n)$. Probleme der Größe $n < b^{-1}(1)$ lösen wir direkt ohne rekursive
i.D. mehr Aufrufe und bezeichnen mit $f(n)$ die Kosten des direkten Lösens. Then we obtain the following recurrence for $T(n)$, the running time on a problem of size n.

$$T(n) = \begin{cases} f(n) & \text{for } 1 \le n < b^{-1}(1); \\ a(n)\cdot T(b(n)) + f(n) & \text{for } n \ge b^{-1}(1). \end{cases}$$

We solve the recurrence for $T(n)$ in a two-step process. As a first step we solve the "simpler" homogeneous recurrence

$$h(n) = \begin{cases} 1 & \text{for } 1 \le n < b^{-1}(1); \\ a(n)\cdot h(b(n)) & \text{for } n \ge b^{-1}(1). \end{cases}$$

In all our applications the solution of the homogeneous recurrence will be easily obtained explicitly. Setting $R(n) = T(n)/h(n)$ the recurrence for $R(n)$ transforms into

$$R(n) = \begin{cases} f(n) & \text{for } 1 \le n < b^{-1}(1); \\ R(b(n)) + f(n)/h(n) & \text{for } n \ge b^{-1}(1). \end{cases}$$

Thus $R(n)$ can be computed by a simple summation. With $g(n) = f(n)/h(n)$ we have $R(n) = g(n) + g(b(n)) + \cdots$.

Theorem 4. *Let a, b, $f : [1, \infty] \mapsto \mathbb{R}_+$, b strictly increasing and $b(n) < n$ for all n. Then the recurrence*

$$T(n) = \begin{cases} f(n) & \text{for } 1 \leq n < b^{-1}(1); \\ a(n) \cdot T(b(n)) + f(n) & \text{for } n \geq b^{-1}(1) \end{cases}$$

has solution

$$T(n) = h(n) \sum_{i=0}^{rd(n)} g(b^{(i)}(n)),$$

where

$$rd(n) = \min\{d;\ b^{(d)}(n) < b^{-1}(1)\},$$

$$h(n) = \begin{cases} 1 & \text{for } 1 \leq n < b^{-1}(1); \\ a(n) \cdot h(b(n)) & \text{for } n \geq b^{-1}(1), \end{cases}$$

and

$$g(n) = f(n)/h(n).$$

Proof: By the discussion above. ▮

Theorem 4 can be visualized as follows. Define the notion of a recurrence tree by induction on n. If $n < b^{-1}(1)$ then the recurrence tree consists of a single leaf labelled $f(n)$. If $n \geq b^{-1}(1)$ then the tree for n consists of a root labelled $f(n)$ and $a(n)$ subtrees, each being a recurrence tree for $b(n)$. Then $rd(n)$ is the depth of this tree and $h(n)$ is the number of leaves of this trees.

More generally, the number of leaves below a node of depth d is $h(b^{(d)}(n))$. $T(n)$ is the sum of the labels of all nodes and leaves of the tree. We can determine $T(n)$ by summing the labels by depth. A node of depth d has label $f(b^{(d)}(n))$. If we distribute that label over all leaves below the node then each leaf receives a portion $f(b^{(d)}(n))/h(b^{(d)}(n)) = g(b^{(d)}(n))$. Thus

$$T(n) = h(n) \sum_{d=0}^{rd(n)} g(b^{(d)}(n)).$$

We can now classify the solutions of the recurrence according to the growth of function g.

Corollary 1. *On the assumptions of Theorem 4 we have*

a) $T(n) = O(h(n))$, *if* $g(n) = O(q^{-rd(n)})$ *for some* $q > 1$;

b) $T(n) = O(h(n))$, *if* $g(n) = O(rd(n)^p)$ *for some* $p < -1$;

c) $T(n) = O(h(n) \cdot \log rd(n))$, *if* $g(n) = O(1/rd(n))$;

d) $T(n) = O(h(n) \cdot (rd(n))^{p+1})$, *if* $g(n) = O(rd(n)^p)$ *for some* $p > -1$;

e) $T(n) = \Omega(h(n) \cdot q^{rd(n)})$, *if* $g(n) = \Omega(q^{rd(n)})$ *for some* $q > 1$;
f) $T(n) = \Theta(f(n))$, *if* $g(n) = \Theta(q^{rd(n)})$ *for some* $q > 1$.

Proof: Let $n_0 = b^{(rd(n))}(n)$. Then we can rewrite the conclusion of Theorem 4 as

$$\begin{aligned}T(n) &= h(n) \sum_{i=0}^{rd(n)} g(b^{(i)}(n))\\ &= h(n) \sum_{i=0}^{rd(n)} g(b^{(-rd(n)+i)}(n_0))\\ &= h(n) \sum_{i=0}^{rd(n)} g(b^{(-i)}(n_0)).\end{aligned}$$

a) From $g(n) = O(q^{-rd(n)})$ and hence $g(b^{(-i)}(n_0)) = O(q^{-i})$ we conclude

$$T(n) = O\left(h(n) \sum_{i=0}^{rd(n)} q^{-i}\right) = O(h(n)).$$

b-d) From $g(n) = O(rd(n)^p)$ and hence $g(b^{(-i)}(n_0)) = O(i^p)$ we conclude

$$\begin{aligned}T(n) &= O\left(h(n) \sum_{i=0}^{rd(n)} i^p\right)\\ &= \begin{cases} O(h(n)) & \text{if } p < -1;\\ O(h(n) \cdot \log rd(n)) & \text{if } p = -1;\\ O(h(n) \cdot rd(n)^{p+1}) & \text{if } p > -1.\end{cases}\end{aligned}$$

e) From $g(n) = \Omega(q^{rd(n)})$ and hence $g(b^{(-i)}(n_0)) = \Omega(q^i)$ we conclude

$$T(n) = \Omega\big(h(n) \sum_{i=0}^{rd(n)} q^i\big) = \Omega\big(h(n) q^{rd(n)}\big)$$

f) As in part e) we conclude

$$\begin{aligned}T(n) &= \Theta(h(n) \sum_{i=0}^{rd(n)} q^i)\\ &= \Theta\big(h(n) \cdot q^{rd(n)}\big)\\ &= \Theta\big(h(n) \cdot g(n)\big)\\ &= \Theta(f(n))\end{aligned}$$

∎

Again it is very helpful to visualize Corollary 1 in terms of recurrence trees. In cases a) and b) of Corollary 1 the cost of the leaves dominates, in case f) the cost of the root dominates and in case d), $p = 0$ the cost is the same on all levels of the tree.

We end with a brief discussion of a frequently occuring special case: $a(n) = a$ and $b(n) = n/b$ for all n.

Corollary 2. *Let a and $b \in \mathbb{R}_+$, $b > 1$, and let $f : [1, \infty] \mapsto \mathbb{R}_+$. Then recurrence*

$$T(n) = \begin{cases} f(n) & \text{if } 1 \le n < b; \\ a \cdot T(n/b) + f(n) & \text{if } n \ge b \end{cases}$$

has solution

$$T(n) = \sum_{i=0}^{\lfloor \log_b n \rfloor} a^i \cdot f(n/b^i).$$

In particular,

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } f(n) = O(n^p) \text{ with} \\ & \quad p < \log_b a; \\ O(n^{\log_b a} \cdot (\log_b n)^{p+1}) & \text{if } f(n) = O(n^{\log_b a} \cdot (\log_b n)^p) \text{ and} \\ & \quad p > -1; \\ O(n^{\log_b a} \cdot \log \log n) & \text{if } f(n) = O(n^{\log_b a} \cdot (\log_b n)^p) \text{ and} \\ & \quad p = -1; \\ O(n^{\log_b a}) & \text{if } f(n) = O(n^{\log_b a} \cdot (\log_b n)^p) \text{ and} \\ & \quad p < -1; \\ O(n^p) & \text{if } f(n) = \Theta(n^p) \text{ with} \\ & \quad p > \log_b a. \end{cases}$$

Proof: Using the notation of Theorem 4 and Corollary 1 we have $a(n) = a$, $b(n) = n/b$, $rd(n) = \lfloor \log_b n \rfloor$ and $h(n) = a^{\lfloor \log_b n \rfloor} = O(a^{\log_b n}) = O(2^{(\log n)(\log a)/\log b}) = O(n^{\log_b a})$. Thus

$$\begin{aligned} T(n) &= a^{\lfloor \log_b n \rfloor} \sum_{i=0}^{\lfloor \log_b n \rfloor} f(n/b^i)/a^{\lfloor \log_b n - i \rfloor} \\ &= \sum_{i=0}^{\lfloor \log_b n \rfloor} a^i \cdot f(n/b^i). \end{aligned}$$

Next note that $f(n) = O(n^p)$ for some $p < \log_b a$ implies $g(n) = O\big(q^{-rd(n)}\big)$ for some $q > 1$ and hence $T(n) = O(h(n))$. Also, if $f(n) = \Theta(n^{\log_b a} \cdot (\log_b n)^p)$ then $g(n) = O((\log_b n)^p) = O(rd(n)^p)$. Thus $T(n)$ is as given by Corollary 1, cases b)–d). Finally case f) handles $f(n) = \Theta(n^p)$ with $p > \log_b a$. ∎

2.1.4. Sorting by Merging

Quicksort's bad worst case behavior stems from the fact that the size of the subproblems created by partitioning is badly controlled. Corollary 6 tells us that $a = b = 2$ would produce an efficient method. How can we achieve the splitting into two subproblems of size $n/2$ each? There is a simple answer: Take the first half of the input sequence and sort it, take the second half of the input sequence and sort it. Then merge the two sorted subsequences to a single sorted sequence. These considerations lead to Program 12.

```
procedure Mergesort(S);
begin let $n = |S|$;
        split $S$ into two subsequences $S_1$ and $S_2$ of length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$
        respectively;
        Mergesort($S_1$);
        Mergesort($S_2$);
        suppose that the first recursive call produces sequence
        $x_1 \le x_2 \le \cdots \le x_{\lceil n/2 \rceil}$;
        the second recursive call produces $y_1 \le y_2 \le \cdots \le y_{\lfloor n/2 \rfloor}$;
        merge the two sequences into a single sorted sequence
        $z_1 \le z_2 \le \cdots \le z_n$
end.
```

Program 12

Let us take a closer look at the merge algorithm. An obvious approach is to repeatedly compare the front elements of the x-sequence and the y-sequence, to remove the smaller one and add it to the end of the z-sequence. In Program 13 this is implemented in our programming language.

```
$i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$;
while $i \le \lceil n/2 \rceil$ and $j \le \lfloor n/2 \rfloor$
do if $x_i < y_j$
    then $z_k \leftarrow x_i$; $i \leftarrow i + 1$; $k \leftarrow k + 1$
    else  $z_k \leftarrow y_j$; $j \leftarrow j + 1$; $k \leftarrow k + 1$
    fi
od;
if $i \le \lceil n/2 \rceil$
then while $i \le \lceil n/2 \rceil$ do $z_k \leftarrow x_i$; $i \leftarrow i + 1$; $k \leftarrow k + 1$ od;
else  while $j \le \lfloor n/2 \rfloor$ do $z_k \leftarrow y_j$; $j \leftarrow j + 1$; $k \leftarrow k + 1$ od
fi;
```

Program 13

This algorithm merges two sequences with at most $\leq \lceil n/2 \rceil + \lfloor n/2 \rfloor - 1 = n-1$ comparisons and total running time $\Theta(n)$. Note that at least one element is added to the z-sequence without a comparison. This is optimal as we show in

Theorem 5. *Every comparison-based algorithm for merging two sorted sequences $x_1 \leq x_2 \leq \ldots \leq x_{\lceil n/2 \rceil}$ and $y_1 \leq y_2 \leq \ldots \leq y_{\lfloor n/2 \rfloor}$ into a single sorted sequence $z_1 \leq z_2 \leq \ldots \leq z_n$ needs $n-1$ comparisons in the worst case.*

Proof: We choose the x- and y-sequence such that the elements are pairwise different and such that the following relations among the elements hold: $x_1 < y_1 < x_2 < y_2 < \cdots < x_{\lfloor n/2 \rfloor} < y_{\lfloor n/2 \rfloor}$ $(< x_{\lceil n/2 \rceil})$. Element $x_{\lceil n/2 \rceil}$ only exists if n is odd. Let us assume that some algorithm produces the correct z-sequence and does so in less than $n-1$ comparisons. Then it has not compared a pair x_i, y_i or y_i, x_{i+1}.

Let us assume the former case, i.e., it has not compared x_i with y_i for some i, $1 \leq i \leq n/2$. Then let us consider the algorithm on an input where the following relations hold: $x_1 < y_1 < x_2 < \cdots < y_{i-1} < y_i < x_i < x_{i+1} < \cdots$. All comparisons which are made by the algorithm have the same outcome on the original and the modified input and hence the algorithm produces the same z-sequence on both inputs, contradiction. ∎

We will next compute the exact number of comparisons which Mergesort uses on an input of length n in the worst case. We use $M(n)$ to denote that number. Apparently

$$\begin{aligned} &M(1) = 0 \qquad \text{and} \\ &M(n) = n - 1 + M(\lceil n/2 \rceil) + M(\lfloor n/2 \rfloor) \quad \text{if } n > 1. \end{aligned}$$

We use induction on n to show

$$M(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

This is correct for $n = 1$. So let $n > 1$.

Case 1: $n \neq 2^k + 1$.
Then $\lceil \log \lfloor n/2 \rfloor \rceil = \lceil \log \lceil n/2 \rceil \rceil = \lceil \log n \rceil - 1$, and therefore

$$\begin{aligned} M(n) &= n - 1 + \lceil n/2 \rceil \cdot \lceil \log \lceil n/2 \rceil \rceil - 2^{\lceil \log \lceil n/2 \rceil \rceil} + 1 \\ &\qquad + \lfloor n/2 \rfloor \cdot \lceil \log \lfloor n/2 \rfloor \rceil - 2^{\lceil \log \lfloor n/2 \rfloor \rceil} + 1 \\ &= n + n(\lceil \log n \rceil - 1) - 2^{\lceil \log n \rceil} + 1 \\ &= n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1. \end{aligned}$$

Case 2: $n = 2^k + 1$.
Then $\lceil \log \lfloor n/2 \rfloor \rceil = \lceil \log \lceil n/2 \rceil \rceil - 1 = \lceil \log n \rceil - 2$, and therefore

$$\begin{aligned} M(n) &= n - 1 + \lceil n/2 \rceil \cdot (\lceil \log n \rceil - 1) - 2^{\lceil \log n \rceil - 1} + 1 \\ &\qquad + \lfloor n/2 \rfloor \cdot (\lceil \log n \rceil - 2) - 2^{\lceil \log n \rceil - 2} + 1 \\ &= n \lceil \log n \rceil - \lfloor n/2 \rfloor - 2^{\lceil \log n \rceil} + 2^{\lceil \log n \rceil - 2} + 1 \\ &= n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1. \end{aligned}$$

We introduced Mergesort by way of a recursive program. It is easy to replace recursion by iteration in the case of Mergesort. Split the input sequence into n sequences of length 1. A sequence of length 1 is sorted. Then we pair the sequences of length 1 and merge them. This gives us $\lfloor n/2 \rfloor$ sorted sequences of length 2 and maybe one sequence of length 1. Then we merge the sequences of length 2 into sequences of length 4, and so on. It is clear that the running time of this algorithm is proportional to the number of comparisons.

Theorem 6. *Mergesort sorts n elements with at most $n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$ comparisons and running time $\Theta(n \log n)$.* ∎

Translation into RAM code results in a running time of $12n \log n + 40n + 97 \log n + 29$ in the worst case (Exercise 10).

Sorting by merging accesses data in a purely sequential manner. Hence it is very appropriate for sorting with secondary memory, such as disks and tapes. In this context the following problem is of interest. Very often we do not start with sorted sequences of length one but we are given n sorted sequences $S_1, \ldots, S_n$ of length $w_1, \ldots, w_n$, respectively. The problem is to find the optimal order of merging these sequences into a single sequence. Here we assume that it costs $x+y$ time units to merge a sequence of length x with a sequence of length y. Any merging pattern can be represented as a binary tree with n leaves. The n leaves represent the n initial sequences and the $n-1$ internal nodes represent the sequences obtained by merging. Tree from Figure 24 represents the merging pattern:

$$\begin{aligned} &S_6 \leftarrow \text{Merge}(S_1, S_3); \\ &S_7 \leftarrow \text{Merge}(S_6, S_4); \\ &S_8 \leftarrow \text{Merge}(S_2, S_5); \\ &S_9 \leftarrow \text{Merge}(S_7, S_8). \end{aligned}$$

Definition: Let T be a binary tree with n leaves $v_1, \ldots, v_n$, let $CONT$: leaves of $T \mapsto \{w_1, \ldots, w_n\} \subseteq \mathbb{R}$ be a bijection and let d_i be the depth of leaf v_i. Then

$$Cost(T) = \sum_i d_i \cdot CONT(v_i)$$

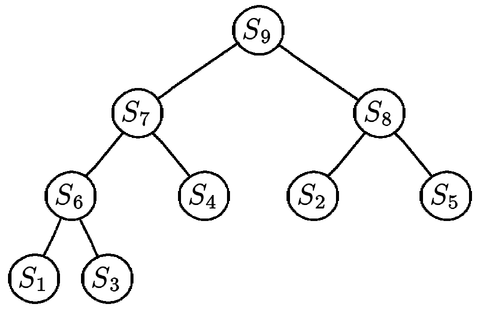


Figure 24. A merging pattern

is called the **cost of tree** T with respect to labelling *CONT*. ▮

This definition requires a little bit of explanation: Tree T is a merging pattern, the leaves of T are labelled by the n initial sequences, their lengths (weights) respectively. What is the cost of merging the n sequences according to pattern T? Note that in our example above sequence S_1 is merged three times into larger sequences, namely first with S_3, then as a part of S_6 with S_4 and then as a part of S_7 with S_8. Also, three is the depth of the leaf labelled S_1. In general, a leaf v of depth d is merged d-times into larger sequences for a total cost of $d \cdot \mathit{CONT}(v)$. Thus the cost of a merging pattern T is as given in the definition above. We want to find the merging pattern of minimal cost.

Definition: Tree T with labelling *CONT* is **optimal** if
$\mathit{Cost}(T) \leq \mathit{Cost}(T')$ for any other tree T' and labelling CONT'. ▮

Theorem 7. *If* $0 \leq w_1 \leq w_2 \leq \cdots \leq w_n$ *then an optimal tree* T *and labelling* *CONT can be found in linear time.*

Proof: We construct tree T in a bottom-up fashion. We start with a set $V = \{v_1, \ldots, v_n\}$ of n leaves and labelling $\mathit{CONT}(v_i) = w_i$ for $1 \leq i \leq n$ and an empty set I of internal nodes and set k to zero; k counts the number of internal nodes so far constructed.

```
while $k < n - 1$
do select $x_1$, $x_2 \in I \cup V$ with the smallest values of $\mathit{CONT}$;
    co ties are broken arbitrarily; oc
    construct a new node $x$ with
        $\mathit{CONT}(x) = \mathit{CONT}(x_1) + \mathit{CONT}(x_2)$
    and add $x$ to $I$;
    $k \leftarrow k + 1$;
    delete $x_1$ and $x_2$ from $I \cup V$
od.
```

Before we proof correctness and analyze running time we illustrate the algorithm on an example. Let $n = 5$ and $\{w_1, \ldots, w_5\} = \{1, 2, 4, 4, 4\}$. We start with 5 leaves of weight $1, 2, 4, 4, 4$. As a first step we combine the leaves of weight 1 and 2 and obtain a node with weight (content) 3, ... (cf. Fig. 25).

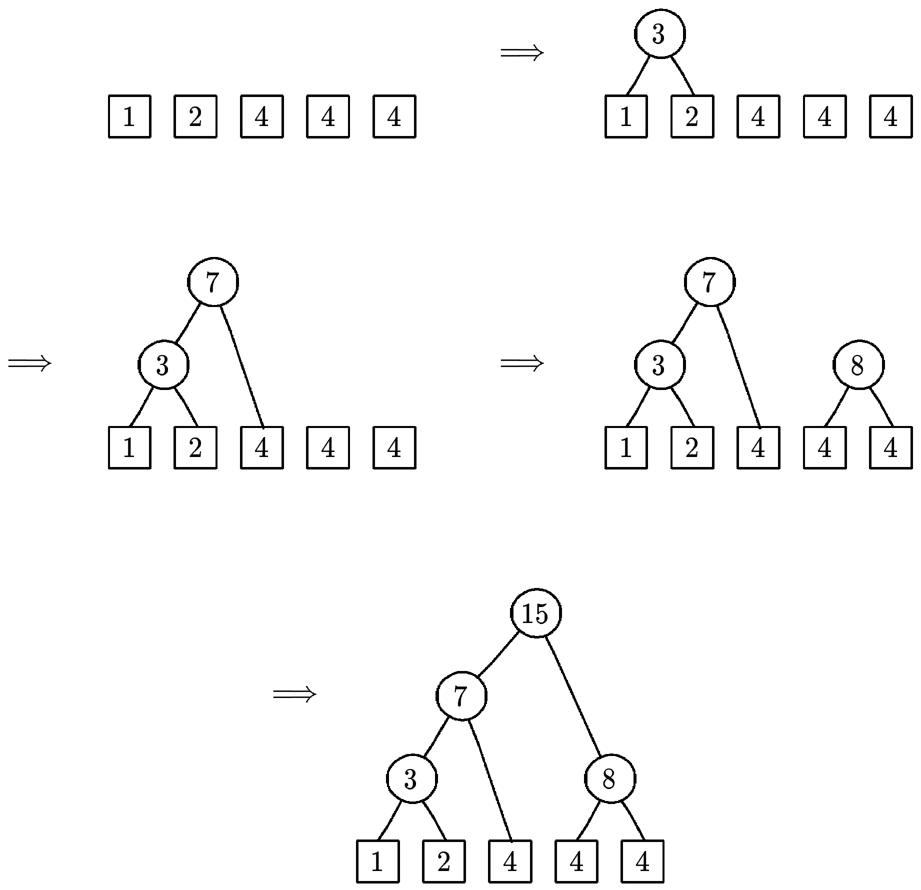


Figure 25. Optimal tree for weights 1 2 4 4 4

The proof of the theorem will be completed by the next 4 lemmata. Let T_{opt} with labelling $CONT_{opt}$ be an optimal tree. Let $\{y_1, \ldots, y_n\}$ be the set of leaves of T_{opt}. Assume w.l.o.g. that $CONT_{opt}(y_i) = w_i$ for $1 \leq i \leq n$. Let d_i^{opt} be the depth of leaf y_i in tree T_{opt}.

Lemma 1. *If $w_i < w_j$ then $d_i^{opt} \geq d_j^{opt}$ for all i, j.*

Proof: (by contradiction). Assume otherwise, say $w_i < w_j$ and $d_i^{opt} < d_j^{opt}$ for some i and j. If we interchange the labels of leaves y_i and y_j then we obtain a tree with cost

$$\begin{aligned} &Cost(T_{opt}) - d_i^{opt} w_i + d_j^{opt} w_i - d_j^{opt} w_j + d_i^{opt} w_j \\ &= Cost(T_{opt}) - (w_j - w_i)(d_j^{opt} - d_i^{opt}) \\ &< Cost(T_{opt}), \end{aligned}$$

a contradiction. ∎

Lemma 2. *There is an optimal tree in which the leaves with content w_1 and w_2 are brothers.*

Proof: Let y be a node of maximal depth in T_{opt} and let y_i and y_j be its children. Then y_i and y_j are leaves. Assume w.l.o.g. that $CONT_{opt}(y_i) \le CONT_{opt}(y_j)$. From Lemma 1 we infer that either $CONT(y_i) = w_1$ or $d_i \le d_1$ and hence $d_i = d_1$ by the choice of y. In either case we may exchange leaves y_1 and y_i without affecting cost. This shows that there is an optimal tree such that y_1 is a child of y. Similarly, we infer from Lemma 1 that either $CONT(y_j) = w_2$ or $d_j \le d_2$ and hence $d_j = d_2$. In either case we may exchange leaves y_2 and y_j without affecting cost. In this way we obtain an optimal tree in which y_1 and y_2 are brothers. ∎

Lemma 3. *The algorithm above constructs an optimal tree.*

Proof: (by induction on n). The claim is obvious for $n \le 2$. So let us assume $n \ge 3$ and let T_{alg} be the tree constructed by our algorithm for weights $w_1 \le w_2 \le \cdots \le w_n$. The algorithm combines weights w_1 and w_2 first and constructs a node of weight (content) $w_1 + w_2$. Let T'_{alg} be the tree constructed by our algorithm for set $\{w_1 + w_2, w_3, w_4, \ldots, w_n\}$ of weights. Then

$$Cost(T_{alg}) = Cost(T'_{alg}) + w_1 + w_2,$$

because T_{alg} can be obtained from T'_{alg} by replacing a leaf of weight $w_1 + w_2$ by a node with two leaf children of weight w_1 and w_2, respectively. Also, T'_{alg} is optimal for the set of $n-1$ weights $w_1 + w_2, w_3, \ldots, w_n$ by induction hypothesis.

Let T_{opt} be an optimal tree satisfying Lemma 2, i.e., the leaves with content w_1 and w_2 are brothers in T_{opt}. Let T' be the tree obtained from T_{opt} by replacing leaves w_1 and w_2 and their parent by a single leaf of weight $w_1 + w_2$. Then

$$\begin{aligned} Cost(T_{opt}) &= Cost(T') + w_1 + w_2 \\ &\ge Cost(T'_{alg}) + w_1 + w_2 \\ &= Cost(T_{alg}), \end{aligned}$$

since $Cost(T') \ge Cost(T'_{alg})$ by induction hypothesis. ∎

It remains to analyze the running time of the algorithm. The crucial observation is:

Lemma 4. *Let $z_1, z_2, \ldots, z_{n-1}$ be the nodes created by the algorithm in order. Then $CONT(z_1) \le CONT(z_2) \le \cdots \le CONT(z_{n-1})$. Furthermore, we have $V = \{v_i, \ldots, v_n\}$, $I = \{z_j, \ldots, z_k\}$ for some $i \le n+1$, $j \le k+1 \le n$ when entering the body of the loop.*

Proof: (by induction on k). The claim is certainly true when $k = 0$. At each iteration of the loop we increase k by 1 and $i + j$ by 2. Also $CONT(z_{k+1}) \ge CONT(z_k)$ is immediately obvious from the construction. ▮

Lemma 4 suggests a linear time implementation. We keep the elements of V and I in two separate sets, both ordered according to $CONT$. Since $w_1 \le \cdots \le w_n$ a queue will do for V and since $CONT(z_1) \le \cdots \le CONT(z_{n-1})$ a queue will do for I. It is then easy to select $x_1, x_2 \in I \cup V$ with the two smallest values of $CONT$ by comparing the front elements of the queues. Also x_1, x_2 can be deleted in time $O(1)$ and the newly created node can be added to the I-queue in constant time. ▮

Theorem 7 can be generalized to non-binary trees.

2.1.5. Comparing Different Algorithms

We considered four sorting methods so far: maximum selection, Heapsort, Quicksort and Mergesort. We see one more in Section 3.5.3.2: A-Sort. Figure 26 summarizes our knowledge. Here c denotes a constant and F denotes the number of inversions in the input sequence; $0 \le F \le n(n-1)/2$ (cf. 3.5.3.2 for details).

Access: Heapsort, Quicksort and A-Sort require random access, Mergesort accesses the keys in a purely sequential manner. Therefore Mergesort is well suited for use in connection with secondary storage which allows only sequential access (e.g., tapes).

Storage Requirement: Heapsort needs space for a few pointer on top of the storage required for the input sequence. Quicksort also needs space for a pushdown store which holds the arguments of pending recursive calls. Maximal stack height is $n/2$, namely if $k = r - 2$ always and therefore the size of the subproblem which has to be solved in line (7) is only by two smaller than the size of the original problem. Maximal stack height can be kept to $\log n$ by a slight modification of the program: always solve the smaller subproblem first, i.e., replace lines (7) and (8) by Program 14.

This modification has the following effect. The size of the subproblem which is solved first has at most $1/2$ the size of the original problem. Therefore there are never more than $\log n$ recursive calls pending.

	Heapsort	Quicksort	Mergesort
# of comparisons			
worst case	$2n \log n$	$n^2/2$	$n \log n$
average case	$\approx 2n \log n$	$1.44n \log n$	$n \log n$
running time			
worst case	$20n \log n$	$\Theta(n^2)$	$12n \log n$
average case	$\leq 20n \log n$	$9n \log n$	$12n \log n$
storage requirement	$n + c$	$n + \log n + c$	$2n + c$
access	random	random	sequential
stable	no	no	yes

	Single maximum selection	A-Sort
# of comparisons		
worst case	$n^2/2$	$O(n \log F/n)$
average case	$n^2/2$	$O(n \log F/n)$
running time		
worst case	$2.5n^2$	$O(n \log F/n)$
average case	$2.5n^2$	$O(n \log F/n)$
storage requirement	$n + c$	$5n$
access	random	random
stable	no	yes

Figure 26. Overview table

```
if $k \leq (l+r)/2$
then if $l < k-1$ then $Quicksort(l, k-1)$ fi;
     if $k+1 < r$ then $Quicksort(k+1, r)$ fi
else if $k+1 < r$ then $Quicksort(k+1, r)$ fi;
     if $l < k-1$ then $Quicksort(l, k-1)$ fi
fi;
```

Program 14

Mergesort requires two arrays of length n and space for some additional pointers. A-sort is based on (a, b)-trees (cf. 3.5.2 and 3.5.3). For $a = 2$ and $b = 3$ a node of an (a, b)-tree requires 5 storage locations and may contain only one key. Hence A-sort requires up to $5n$ storage locations.

Average asymptotic running time: The table contains the dominating terms of the bounds on running time derived above. The relation of the running times of Quicksort : Mergesort : Heapsort is 1 : 1.33 : 2.2, which is typical of many computers, not only of the treated RAM. Note however, that the worst case running time of Mergesort and Heapsort is not much larger than their expected running time. The relative efficiency of Quicksort is based on the following facts: All comparisons in the partitioning phase are made with the same element and therefore this element can be kept in a special register. Also, an exchange of keys is required only after every fourth comparison on the average (Exercise 6), after every second in Heapsort and after every comparison in Mergesort. However, Heapsort compares twice as often as Mergesort.

For small n, the relations somewhat change. Simple maximum selection is better than Quicksort for $n \le 22$. Therefore Quicksort may be improved by terminating recursion for some n_0 and by switching to maximum selection for smaller n (Exercise 11).

A-sort is inferior to all other methods with respect to average and worst case running time. However, A-sort has one major advantage. It runs fast on nearly sorted inputs, i.e., if F is small, say $F = O(n \log n)$, then A-sort runs in time $O(n \log \log n)$. This is in marked contrast to the other methods. Heapsort's and Mergesort's behavior hardly depends on the statistical properties of the input and Quicksort even runs slower on nearly sorted inputs.

2.1.6. Lower Bounds

In this section we prove a lower bound on the number of comparisons required for sorting and related problems. We will first show an $\Omega(n \log n)$ lower bound on the average and worst case complexity of general sorting algorithms. We will then extend this lower bound to randomized general sorting problems and to decision tree algorithms allowing comparisons of rational functions of the inputs.

General sorting algorithms are comparison based, i.e., the only operation applicable to elements of the universe from which the keys are drawn is the comparison between two elements with outcome $\le$ or $>$. In particular, the pair of keys compared at any moment of time depends only on the outcome of previous comparisons and on nothing else. For purposes of illustration let us consider sorting by simple maximum selection on sequence S_1, S_2, S_3. The first comparison is $S_3 : S_2$. If $S_3 > S_2$ then S_3 is compared with S_1 next, if $S_3 \le S_2$ then S_2 is compared with S_1 next, ... We can illustrate the complete sorting algorithm by a tree, a decision tree (cf. Fig. 27).

Node $i : j$ denotes a comparison between S_i and S_j. The edge to the right child corresponds to $S_i > S_j$, the edge to the left corresponds to $S_i \le S_j$.

Definition: A **decision tree** is a binary tree whose nodes have labels of the form $S_i : S_j$. The two outgoing edges are labelled $\le$ and $>$. ▮

Let $S_1, \ldots, S_n$ elements of the universe U. The computation of decision tree T on input $S_1, \ldots, S_n$ is defined in a natural way. We start in the root of the tree. Suppose now that we reached node v which is labelled $S_i : S_j$. We then compare S_i with S_j and proceed to one of the children of v depending on whether $S_i \le S_j$ or $S_i > S_j$. The leaves of a decision tree represent the different outcomes of the algorithm.

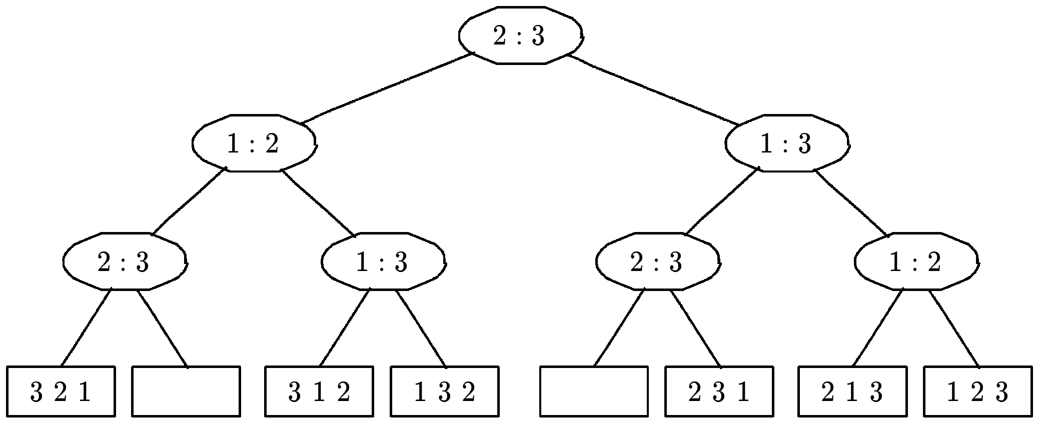


Figure 27. Decision tree for sorting 3 objects

Definition: Let T be a decision tree. T **solves the sorting problem** of size n if there is a labelling of the leaves of T by permutations of $\{1,\ldots,n\}$ such that for every input $S_1,\ldots,S_n \in U$: with $S_i \neq S_j$ for all $i \neq j$: if the leaf reached on $S_1,\ldots,S_n$ is labelled by π then $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$. ▮

We can now define the worst case and average case complexity of the sorting problem. For a decision tree T and permutation π let l_π^T be the depth of the leaf which is reached on input $S_1,\ldots,S_n$ with $S_{\pi(1)} < \cdots < S_{\pi(n)}$. Define

$$S(n) = \min_T \max_\pi \; l_\pi^T \qquad \text{and}$$
$$A(n) = \min_T \frac{1}{n!} \sum_\pi l_\pi^T,$$

where T ranges over all decision trees for sorting n elements and π ranges over all permutations of n elements. l_π^T is the number of comparisons used on input π by decision tree T. Thus $S(n)$ is the minimal worst case complexity of any algorithm and $A(n)$ is the minimal average case complexity of any algorithm. We prove lower bounds on $S(n)$ and $A(n)$.

Suppose $S(n) \le k$. A binary tree of depth $\le k$ has at most 2^k leaves. A decision tree for sorting n elements has at least $n!$ leaves. Thus

$$2^{S(n)} \ge n! \qquad \text{or}$$
$$S(n) \ge \lceil \log n! \rceil,$$

since $S(n)$ is an integer. Stirling's approximation for $n!$ gives us

$$\log n! = \left(n + \frac{1}{2}\right)\log n - \frac{n}{\ln 2} + O(1)$$
$$= n\log n - 1.44n + O(\log n).$$

An upper bound for $S(n)$ is derived from the analysis of sorting algorithms. In particular, we infer from the analysis of Mergesort

$$S(n) \le n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1,$$

and hence

$$\lim_{n\to\infty} \frac{S(n)}{n \log n} = 1.$$

We have thus determined the exact asymptotic worst case complexity of sorting. The result derived above is often called the **information-theoretic bound**. Any sorting algorithm has to distinguish $n!$ possibilities, and thus has to gain $\log n!$ bits of information. One comparison gives at most 1 bit of information. The bound follows.

We turn to $A(n)$. We give a different interpretation of $(1/n!)\sum_\pi l_\pi^T$ first. Let $b_1, b_2, \ldots$ be the leaves of T. Define

$$\alpha_i = \begin{cases} 1/n! & \text{if leaf } b_i \text{ is reached on some input sequence;} \\ 0 & \text{otherwise.} \end{cases}$$

Then $(1/n!)\sum_\pi l_\pi^T = \sum_i \alpha_i \cdot \text{depth}(b_i)$ and hence $(1/n!)\sum_\pi l_\pi^T$ is equal to the weighted path length of tree T with respect to distribution $\alpha_1, \alpha_2, \ldots$ (cf. 3.4 for a detailed discussion of weighted path length). We show in 3.4 in a more general context (Use Theorem 5 from 3.4.1 with $B = 0$ und lassen Sie d gegen ∞ gehen.
i.D.mehr This implies $H \le P$), that

$$\begin{aligned} \sum_\pi l_\pi^T \frac{1}{n!} &\ge H\Big(\frac{1}{n!}, \ldots, \frac{1}{n!}\Big) \\ &= \sum_\pi \frac{1}{n!} \log n! = \log n! \end{aligned}$$

for any tree T. Here H is the entropy of the distribution (cf. 3.4.1). We summarize in

Theorem 8. *Any decision tree algorithm for sorting n elements needs* $\ge \lceil \log n! \rceil$ *comparisons in the worst case and* $\ge \log n!$ *comparisons on the average.* ∎

Theorem 8 can be used to prove lower bounds on the other problems besides sorting. The **element uniqueness problem** is defined as follows. Given n elements $S_1, \ldots, S_n$ in U one has to decide whether $S_i \ne S_j$ for $i \ne j$.

Theorem 9. *Every decision tree algorithm for the element uniqueness problem of size n needs at least* $\log n!$ *comparisons in the worst case.*

Proof: Let T be a decision tree for the element uniqueness problem of size n, i.e., the nodes of T are labelled by comparisons $S_i : S_j$ and the leaves are labelled yes

or no. On input $(S_1, \ldots, S_n) \in U^n$ a leaf labelled yes is reached $\Longleftrightarrow$ $S_i \neq S_j$ for $i \neq j$. We will show that one can use T for sorting.
For permutation π of n elements let $v(\pi)$ the leaf reached on an input $(S_1, \ldots, S_n) \in U^n$ with $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$. Note that this leaf is well-defined because the computation on any input $(S_1, \ldots, S_n) \in U^n$ with $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$ will end in the same leaf.

Claim: *$v(\pi) \neq v(\rho)$ if $\pi \neq \rho$ (π, ρ permutations)*

Proof: (by contradiction). Assume otherwise, i.e., there are permutations π and ρ such that $v(\pi) = v(\rho)$. Note that leaf $v(\pi)$ is labelled yes. We will now construct an input $(S_1, \ldots, S_n)$ such that the computation on input $(S_1, \ldots, S_n)$ ends in leaf $v(\pi)$, yet $S_i = S_j$ for some $i \neq j$. This is a contradiction.

$(S_1, \ldots, S_n)$ is constructed as follows. Let $w_1, \ldots, w_t$ be the nodes on the path to leaf $v(\pi)$. Consider partial order $P(v(\pi))$ on $(S_1, \ldots, S_n)$ which is defined as follows: For k, $1 \leq k \leq t$: if w_k is labelled $S_i : S_j$ and the $\leq$-edge ($>$-edge) is taken from w_k to w_{k+1} then $S_i < S_j$ ($S_i > S_j$) in partial order $P(v(\pi))$. Then $P(v(\pi))$ is the smallest partial order (with respect to set inclusion) satisfying these constraints. The following two observations are important:

1) If $(R_1, \ldots, R_n) \in U^n$ satisfies the partial order $P(v(\pi))$ then the computation on input $(R_1, \ldots, R_n)$ ends in leaf $v(\pi)$.
2) If $(R_1, \ldots, R_n) \in U^n$ is such that $R_{\pi(1)} < \cdots < R_{\pi(n)}$ or $R_{\rho(1)} < \cdots < R_{\rho(n)}$ then $(R_1, \ldots, R_n)$ satisfies partial order $P(v(\pi))$. Since $\pi \neq \rho$, $P(v(\pi))$ is not a linear order.

Since $P(v(\pi))$ is not a linear order there are a and b, $a \neq b$, such that S_a and S_b are not related in partial order $P(v(\pi))$. Let $(S_1, \ldots, S_n) \in U^n$ be such that $(S_1, \ldots, S_n)$ satisfies partial order $P(v(\pi))$ and $S_a = S_b$. Then the computation on input $(S_1, \ldots, S_n)$ ends in leaf $v(\pi)$, and hence the result is "Yes", contradiction. ∎

With the above follows that T has at least $n!$ leaves and thereby depth at least $\log n!$. ∎

Can a randomized sorting algorithm, i.e., an algorithm which uses random choices in order to determine which comparisons to make next, be any faster? After all, we saw that Quicksort's quadratic worst case behavior disappears when we switch to randomized Quicksort. The answer is "No". A randomized comparison-based algorithm can be represented by a random decision tree.

In a **random decision tree** there are two types of nodes. The first type of node is the ordinary comparison node; the second type is a coin tossing node. It has two outgoing edges labelled 0 and 1 which are taken with probability $\frac{1}{2}$ each. For the sake of simplicity we restrict our attention to finite random decision trees, i.e., we assume that the number of coin tosses on inputs of size n is bounded by $k(n)$.

The notion "a random decision tree solves the sorting problem of size n" is defined exactly as above. Note however, that the leaf reached not only depend on the input but also on the sequence $s \in \{0,1\}^{k(n)}$ of outcomes of coin tosses. For any permutation π and sequence $s \in \{0,1\}^{k(n)}$ let $l^T_{\pi,s}$ be the depth of the leaf reached on sequence s of coin tosses and input sequence $S_1, \ldots, S_n$ where $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$. Define

$$S_{ran}(n) = \min_T \max_\pi \frac{1}{2^{k(n)}} \sum_{s \in \{0,1\}^{k(n)}} l^T_{\pi,s}$$

and

$$A_{ran}(n) = \min_T \frac{1}{n!} \sum_\pi \left(\frac{1}{2^{k(n)}} \sum_{s \in \{0,1\}^{k(n)}} l^T_{\pi,s} \right),$$

where T ranges over all random decision trees which solve the sorting problem of size n and π ranges over all permutations of n elements. A_{ran} (S_{ran}) is the minimal average (worst) case complexity of any randomized comparison-based sorting algorithm. We can now use the argumentation outlined at the end of Section 1.2 to show that randomization cannot improve the complexity of sorting below $\log n!$. We have

$$\begin{aligned} S_{ran}(n) &\geq A_{ran}(n) \\ &\geq \min_T \frac{1}{n!} \sum_\pi \frac{1}{2^{k(n)}} \sum_{s \in \{0,1\}^{k(n)}} l^T_{\pi,s} \\ &\geq \min_T \frac{1}{2^{k(n)}} \sum_{s \in \{0,1\}^{k(n)}} \frac{1}{n!} \sum_\pi l^T_{\pi,s} \\ &\geq \min_T \frac{1}{2^{k(n)}} \sum_{s \in \{0,1\}^{k(n)}} A(n) \\ &\geq A(n). \end{aligned}$$

The next to last inequality holds since for every fixed sequence s of coin tosses random decision tree T defines an ordinary decision tree and hence $(1/n!) \sum_\pi l^T_{\pi,s} \geq A(n)$ for every fixed sequence s. Thus randomization cannot improve expected running time below $\log n!$.

We will next strengthen our basic lower bound in a different direction. Suppose that the inputs are not taken from an arbitrary ordered universe but from the set of real numbers. Reals cannot only be compared but can also be added, multiplied, ... This leads to the concept of rational decision trees.

Definition: A **rational decision tree** for inputs $S_1, \ldots, S_n$ is a binary tree whose nodes are labelled by expressions $p(S_1, \ldots, S_n) : q(S_1, \ldots, S_n)$ where p and q are rational functions, whose edges are labelled by $\leq$ and $>$, and whose leaves are labelled by rational functions $r(S_1, \ldots, S_n)$ of $S_1, \ldots, S_n$. ▮

Rational decision trees compute functions $f : \mathbb{R}^n \mapsto \mathbb{R}$ in an obvious way. Let $(S_1, \ldots, S_n) \in \mathbb{R}^n$. Computation starts in the root. Suppose that computation has reached node v which is labelled $p(S_1, \ldots, S_n) : q(S_1, \ldots, S_n)$. We evaluate rational functions p and q and proceed to the appropriate child of v depending on the outcome of the comparison. If the computation reaches a leaf labelled by a rational function $r(S_1, \ldots, S_n)$ then r is evaluated and the value of r is the result of the computation.

Theorem 10. *Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a function and let T be a rational decision tree which computes f. Let $W_1, \ldots, W_q$ be pairwise disjoint subsets of $\mathbb{R}$ of non-zero measure and let $r_1, \ldots, r_q$ be pairwise different rational functions. If $f|_{W_i} = r_i$ for $1 \le i \le q$ then rational decision tree T has depth at least* $\log q$.

Proof: Let T be a rational decision tree which computes f. For v, a leaf of T let

$$D_v = \{(S_1, \ldots, S_n) \in \mathbb{R}^n;\ \text{the computation on input } (S_1, \ldots, S_n) \text{ ends in leaf } v\}.$$

Then $\{D_v;\ v \text{ leaf of } T\}$ is a partition of $\mathbb{R}^n$ and hence

$$W_i = \bigcup_{v \text{ leaf of } T} (D_v \cap W_i) \qquad \text{for all } i,\ 1 \le i \le q.$$

Since W_i has non-zero measure there must be a leaf $v(i)$ such that $D_{v(i)} \cap W_i$ has non-zero measure. Note that T has a finite number of leaves. Let $r(v(i))$ be the rational function which labels leaf $v(i)$. Then functions $r(v(i))$ and r_i agree on set $D_{v(i)} \cap W_i$. Since $D_{v(i)} \cap W_i$ has non-zero measure we must have that functions $r(v(i))$ and r_i are identical. Hence $v(i) \ne v(j)$ for $i \ne j$ and T must have at least q leaves. Thus the depth of T is at least log q. ▮

We give two applications of Theorem 10. The first application is sorting. For $(x_1, \ldots, x_n) \in \mathbb{R}^n$ define

$$f(x_1, \ldots, x_n) = x_1^{r_1} + x_2^{r_2} + \cdots + x_n^{r_n}$$

where $r_i = |\{j;\ x_j < x_i\}|$ is the rank of x_i. We call f the **rank function**. For π, a permutation of n elements let

$$W_\pi = \{(x_1, \ldots, x_n) \in \mathbb{R}^n;\ x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}\}.$$

Then W_π has non-zero measure, $f|_{W_\pi}$ is a polynomial and $f|_{W_\pi} \ne f|_{W_\rho}$ for $\pi \ne \rho$. We conclude

Theorem 11. *Any decision tree for the rank function of n arguments has depth at least* $\log n!$. ▮

The second application is searching. For $(y, x_1, \ldots, x_n) \in \mathbb{R}^{n+1}$ define

$$f(y, x_1, \ldots, x_n) = |\{j;\ x_j < y\}|.$$

We call f the **searching function**. For k, $0 \le k \le n$, let

$$W_k = \{(y, x_1, \ldots, x_n) \in R^{n+1};\ \text{exactly } k\ x_j\text{'s are smaller than } y\}.$$

Then W_k has non-zero measure and $f|_{W_k} = k$. Thus $f|_{W_k}$ is a rational function and $f|_{W_k} \ne f|_{W_l}$ for $k \ne l$. We have

Theorem 12. *Any rational decision tree for the searching function of size n has depth at least* $\log(n+1)$. ∎

We will see in Chapter 3 that $\log(n+1)$ is also an upper bound for the complexity of the searching function. Further applications of Theorem 10 can be found in exercises. Unfortunately, Theorem 10 is not strong enough for the element uniqueness problem. Note that there is a rational decision tree of depth one which decides the element uniqueness problem, namely

$$\prod_{1\le i<j\le n} (x_i - x_j) = 0 \ ? \ .$$

i.D.mehr Man beachte, daß es einen rationalen Entscheidungsbaum der Tiefe zwei gibt, der das Elementeindeutigkeitsproblem entscheidet, nämlich wenn

$$\prod_{1\le i<j\le n} (x_i - x_j) = 0 \ ?$$

der Inhalt der Wurzel ist, und der Pfad zu dem mit "Ja" beschrifteten Blatt zweimal dem $\le$-Zeiger folgt und der dazwischenliegende Knoten den gleichen Test wie die Wurzel macht, wobei aber die 0 links vom Gleichheitszeichen steht. This shows that there are problems which are difficult to handle in the restricted model of decision trees but become simple if tests between rational functions are allowed and if no charge is made for computing those functions. Note however, that the best algorithm known for computing the product $\prod_{1\le i<j\le n}(x_i - x_j)$ requires $\Omega(n \log n)$ multiplications and divisions. Hence, if we charge for test *and* algebraic operations then an $\Omega(n\log n)$ lower bound for the element uniqueness problem is conceivable. We will use the remainder of the section to prove such a lower bound. We start by fixing the model of computation.

Definition: An **algebraic computation tree** for inputs $S_1\ldots,S_n$ is a tree T of degree ≤ 2 with a function that assigns

a) to each leaf an output "Yes" or "No"

b) to each vertex v with exactly one child (= **simple node**) an assignment (statement) of the form

$$Y(v) \leftarrow Y(v_1) \text{ op } Y(v_2),$$
$$Y(v) \leftarrow c \text{ op } Y(v_2) \quad \text{or}$$
$$Y(v) \leftarrow \sqrt{Y(v_1)}$$

where op $\in \{+,-,\times,/\}$, v_i is a simple node and a proper ancestor of v in tree T or $Y(v_i) \in \{S_1,\ldots,S_n\}$ and $c \in \mathbb{R}$

c) to each vertex v with exactly two children (= **branching node**) a comparison of the form

$$Y(v_1) \ge 0,$$

$Y(v_1) > 0$ or
$Y(v_1) = 0$

where v_1 is a simple node and a proper ancestor of v or $Y(v_1) \in \{S_1, \ldots, S_n\}$.∎

Algebraic computation trees compute functions $f : \mathbb{R}^n \mapsto \{\text{"Yes"}, \text{"No"}\}$ in an obvious way. Given an input $S = (S_1, \ldots, S_n) \in \mathbb{R}^n$ the program traverses a path $P(S)$ in the tree. In simple nodes an arithmetical operation is performed and the value of the operation is assigned to the variable associated with the simple node. In branching nodes the value of a variable is tested and the computation proceeds to the appropriate child. We require that no computation leads to a division by zero or to taking the square root of a negative number.

The cost of an algebraic computation tree is its depth, the complexity of (the membership problem of) a set $V \subseteq \mathbb{R}^n$ is the minimum cost of any algebraic computation tree which computes the characteristic function of V. The lower bounds on the costs of algebraic computation trees are based on the following fact of algebraic topology.

Fact 1. *Let $q_1, \ldots, q_M$ be polynomials in N variables of degree at most d. Let $V \subseteq \mathbb{R}^N$ be defined by*

$$q_1(x_1, \ldots, x_N) = 0, \quad \ldots \quad , q_M(x_1, \ldots, x_N) = 0.$$

Then the number of components of V is at most $d \cdot (2d-1)^{N-1}$.

Remark: a proof of this can be found in J. Milnor: "On the betti numbers of real algebraic varieties", *Proc. AMS* **15** (1964), 275–280, and is far beyond the scope of this book. We confine ourselves to a short informal discussion. Two points x and x' of V belong to the same component if there is a running line inside V connecting them, i.e., if there is a continous function $h : [0,1] \mapsto V$ with $h(0) = x$ and $h(1) = x'$. The components of V are the equivalence classes under this relation.

The fact above becomes particularly simple if $N = 1$. In this case it is equivalent to the fact that a polynomial of degree d in one variable has at most d real roots. Consider the case $N = 2$ and $d = 2$ next. Then $q_1(x_1, x_2) = 0$ defines an ellipse (parabola, or hyperbola) and so do $q_2, q_3, \ldots$. The crucial observation is now that the equations $q_1(x_1, x_2) = 0$, $q_2(x_1, x_2) = 0$ define a set of at most 6 *points*, i.e., six zero-dimensional sets. All further equations $q_3(x_1, x_2) = 0, \ldots, q_M(x_1, x_2) = 0$ can only weed out some of these points and cannot increase the number of components.

By analogy the truth of the fact is now conceivable for larger degree and larger number of variables as well. The set defined by $q_1(x_1, \ldots, x_N) = 0$ has at most d components, each of which has dimension $N-1$. Intersecting this set with the set $q_2(x_1, \ldots, x_N) = 0$ yields at most $d \cdot (2d-1)$ components, each of which has dimension $N-2$.

Iterating in this way we arrive at $d \cdot (2d-1)^{N-1}$ components of dimension 0 each defined by polynomials $q_1, \ldots, q_N$. Considering more polynomials weeds out some of these points but does not increase the number of components. We end by giving the warning that the argument above is at most a hint at a proof. ∎

We are now ready for the lower bound. We use $\#V$ to denote the number of components of set V.

Theorem 13. *Let T be an algebraic computation for inputs $S_1, \ldots, S_n$ which decides the membership problem for $V \subseteq \mathbb{R}^n$, and let h be the depth of T. Then*

a) $\max(\#V, \#(\mathbb{R}^n - V)) \le 2^h 3^{h+n}$;

b) $h \ge \dfrac{\log \max(\#V, \#(\mathbb{R}^n - V)) - n \log 3}{\log 6}$.

Proof: Part b) follows from part a) by taking logarithms. We still have to prove part a). Since tree T has depth h it has at most 2^h leaves. If w is a leaf of T let $D(w) \subseteq \mathbb{R}^n$ be the set of inputs S for which the computation ends in leaf w. We claim that $\#D(w) \le 3^{h+n}$. Note first that this claim implies part a) since there are at most 2^h leaves w and each leaf satisfies $\#D(w) \le 3^{h+n}$. Thus $\#V \le 2^h 3^{h+n}$ and similarly $\#(\mathbb{R}^n - V) \le 2^h 3^{h+n}$.

$\#D(w) \le 3^{h+n}$ remains to be proved for every leaf w. In order to do so we characterize $D(w)$ by a set of polynomial equations and inequalities of degree at most 2. Let P be the path from the root to leaf w. We traverse the path P and set up a system Γ of equalities and inequalities as follows. Let v be a node on path P.

If v is a simple node then we add for its operation the given equality to Γ:

operation	equation
$Y(v) \leftarrow Y(v_1) \pm Y(v_2)$	$Y(v) = Y(v_1) \pm Y(v_2)$
$Y(v) \leftarrow Y(v_1) \times Y(v_2)$	$Y(v) = Y(v_1) \times Y(v_2)$
$Y(v) \leftarrow Y(v_1)/Y(v_2)$	$Y(v_1) = Y(v)/Y(v_2)$
$Y(v) \leftarrow \sqrt{Y(v_1)}$	$Y(v_1) = Y(v)^2$

and if v is a branching node with a test

$$Y(v_1) > 0 \qquad Y(v_1) \ge 0 \qquad Y(v_1) = 0,$$

then we add this (in-)equality to Γ if the positive outcome is taken on path P and we add

$$-Y(v_1) > 0 \quad -Y(v_1) \ge 0 \quad Y(v)Y(v_1) - 1 = 0$$

otherwise.

Note that in the last case $Y(v)$ is a *new* variable which is not used in any assignment. Also note that the equation $Y(v)Y(v_1) - 1 = 0$ has a solution $\iff$ $Y(v_1) \ne 0$.

The system Γ involves variables $S_1, \ldots, S_n, Y(v_1), \ldots, Y(v_{r'})$ for some integer r'. Also Γ contains exactly r equalities and some number s of inequalities, where $r \ge r'$. Clearly $r + s \le h$, since each node on P adds one equality or inequality. Let $W \subseteq \mathbb{R}^{n+r}$ be the set of solutions to the system Γ. Then the projection of W

onto the first n coordinates is $D(w)$. Since the projection function is continous we conclude that $\#D(w) \leq \#W$ and it therefore suffices to show $\#W \leq 3^{h+n}$.

What do we have obtained at this point? The set W is defined by a set of r equalities and s inequalities. Each equality or inequality is given by a polynomial degree at most 2 in $n+r$ variables which we call $x_1, \ldots, x_{n+r}$ for simplicity now on, i.e., we have a system

$$\begin{array}{rcl} p_1(x_1,\ldots,x_{n+r}) = 0, & \ldots & , p_r(x_1,\ldots,x_{n+r}) = 0, \\ q_1(x_1,\ldots,x_{n+r}) > 0, & \ldots & , q_m(x_1,\ldots,x_{n+r}) > 0, \\ q_{m+1}(x_1,\ldots,x_{n+r}) \geq 0, & \ldots & , q_s(x_1,\ldots,x_{n+r}) \geq 0, \end{array}$$

where each p_i, q_j has degree at most 2.

In order to apply the fact above we transform this system into a set of equalities in a higher dimensional space. Let $t = \#W$ (Milnor proves that t is finite) and let $P_1, \ldots, P_t$ be points in the different components of W. Let

$$\epsilon = \min\{q_j(P_i);\ 1 \leq j \leq m,\ 1 \leq i \leq t\}.$$

Then is $\epsilon > 0$. Let W_ϵ be defined by the system

$$\begin{array}{rcl} p_1(x_1,\ldots,x_{n+r}) = 0, & \ldots & , p_r(x_1,\ldots,x_{n+r}) = 0, \\ q_1(x_1,\ldots,x_{n+r}) \geq \epsilon, & \ldots & , q_m(x_1,\ldots,x_{n+r}) \geq \epsilon, \\ q_{m+1}(x_1,\ldots,x_{n+r}) \geq 0, & \ldots & , q_s(x_1,\ldots,x_{n+r}) \geq 0. \end{array}$$

Then is $W_\epsilon \subseteq W$ and $P_i \in W_\epsilon$ for all i. Hence $\#W \leq \#W_\epsilon$. It therefore suffices to show $\#W_\epsilon \leq 3^{n+h}$. Let $Z \subseteq \mathbb{R}^{n+r+s}$ be defined by the system

$$\begin{array}{ll} p_1(x_1,\ldots,x_{n+r}) = 0, & \ldots\ ,\ p_r(x_1,\ldots,x_{n+r}) = 0, \\ q_1(x_1,\ldots,x_{n+r}) = x_{n+r+1}^2 + \epsilon, & \ldots\ ,\ q_m(x_1,\ldots,x_{n+r}) = x_{n+r+m}^2 + \epsilon, \\ q_{m+1}(x_1,\ldots,x_{n+r}) = x_{n+r+m+1}^2, & \ldots\ ,\ q_s(x_1,\ldots,x_{n+r}) = x_{n+r+s}^2, \end{array}$$

where $x_{n+r+1}, \ldots, x_{n+r+s}$ are new variables. Then W_ϵ is the projection of Z onto the first $n+r$ coordinates and hence $\#W_\epsilon \leq \#Z$. Furthermore, Z is defined by a system of polynomial equations of degree at most two in $n+r+s \leq n+h$ variables. Hence

$$\#Z \ \leq\ 2 \cdot (4-1)^{n+h-1} \ \leq\ 3^{n+h}$$

by the fact above. ∎

We end this section with three applications of Theorem 13.

Theorem 14. *The complexity of the element uniqueness problem in the algebraic computation is $\Omega(n \log n)$.*

Proof: Let $V \subseteq \mathbb{R}^n$ be defined by $\Pi_{1 \leq i < j \leq n}(x_i - x_j) \neq 0$. Then $\#V \geq n!$ as we will now argue. For $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, $x_i \neq x_j$ for $i \neq j$, let σ be the order type of x, i.e., σ is a permutation with $x_{\sigma(1)} < x_{\sigma(2)} < \cdots < x_{\sigma(n)}$. Then it is easy to see that if $x, x' \in \mathbb{R}^n$ have different order types then any line connecting them must go through a point having two equal coordinates, i.e., a point outside V, and hence x and x' belong to different components of V. This proves $\#V \geq n!$. Since $\log n! = \Omega(n \log n)$ the proof is completed by an application of Theorem 13. ▮

Our second application is the convex hull problem (cf. Chapter 8): Given n points in the plane, does their convex hull have n vertices?

Theorem 15. *The complexity of the convex hull problem in the model of algebraic computation trees is* $\Omega(n \log n)$.

Proof: Let

$$\begin{aligned} V = \{(x_1, y_1, \ldots, x_n, y_n);\ &\text{the convex hull of point set} \\ &\qquad \{(x_i, y_i);\ 1 \leq i \leq n\} \text{ has } n \text{ vertices}\} \\ \subseteq \mathbb{R}^{2n}. \end{aligned}$$

As in the proof of Theorem 14 we assign an order type to every element of V. Let $(x_1, y_1, \ldots, x_n, y_n) \in V$ and let p be any point in the interior of the convex hull of point set $\{(x_i, y_i);\ 1 \leq i \leq n\}$. Then the order type of this tuple is the circular ordering of the points around p.

Note that there are $(n-1)!$ different order types and that any line in $\mathbb{R}^{2n}$ connecting tuples of different order types must go through a point outside V. Hence $\#V \geq (n-1)!$ and the theorem follows from Theorem 13. ▮

Our last application is the computation of the elementary symmetric functions: Given $x_1, \ldots, x_n \in \mathbb{R}$, compute the elementary symmetric functions $p_0(x_1, \ldots, x_n)$, $p_1(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)$. Here

$$p_j(x_1, \ldots, x_n) = (-1)^j \sum_{i_1 < i_2 < \cdots < i_j} x_{i_1} x_{i_2} \cdots x_{i_j}.$$

It is a well-known fact in the algebra that the elementary symmetric functions are the coefficients of the interpolation polynomial through points

$$(x_1, 0), \ldots, (x_n, 0), (0, (-1)^n x_1 \cdots x_n).$$

Theorem 16. *The complexity of computing the elementary symmetric functions is* $\Omega(n \log n)$ *in the model of algebraic computation trees.*

Proof: Let $a_i = p_i(1, 2, \ldots, n)$, $1 \leq i \leq n$. An algorithm that computes $p_1(x_1, \ldots, x_n), \ldots, p_n(x_1, \ldots, x_n)$ can be used to test $a_i = p_i(x_1, \ldots, x_n)$, $1 \leq i \leq n$, in n more steps. Since this is true $\iff$ $\{x_1, \ldots, x_n\} = \{1, \ldots, n\}$ (This is a direct consequence of the uniqueness of the interpolation polynomial) it suffices to show that $\#V = n!$ where $V = \big\{(x_1, \ldots, x_n);\ \{x_1, \ldots, x_n\} = \{1, \ldots, n\}\big\}$. The proof that $\#V = n!$ can be given along the lines of the proof of Theorem 14. ▮

Theorems 14, 15 and 16 show the strength of Theorem 13. It can be used to prove lower bounds for a great variety of algorithmic problems. We should also note that Theorem 13 is not strong enough to imply lower bounds on the complexity of RAM computations because indirect addressing is a "non-algebraic" operation (a lower bound on the complexity of RAM computation will be shown in Section 2.3) and that is not strong enough to imply lower bounds for computations over the integers when integer division belongs to the instruction repertoire.

2.2. Sorting by Distribution

2.2.1. Sorting Words by Distribution

Let Σ be a finite alphabet of size m and let $\le$ be a linear order on Σ. We may assume w.l.o.g. that $\Sigma = \{1, 2, \ldots, m\}$ with the usual ordering relation. The ordering on Σ is extended to an ordering on the words over Σ as usual.

Definition: Let $x = x_1 \ldots x_k$ and $y = y_1 \ldots y_l$ be words over Σ, i.e., $x_i, y_i \in \Sigma$. Then is x **smaller** than y in the algebraic ordering (denoted $x < y$) if there is an i, $0 \le i \le k$, such that $x_j = y_j$ for $1 \le j \le i$ and either $i = k < l$ or $i < k$, $i < l$ and $x_{i+1} < y_{i+1}$. ▮

The definition of alphabetic ordering conforms to everyday usage, e.g., we have $x = ABC < AD = y$ because $x_1 = y_1$ and $x_2 < y_2$.

We treat the following problem in this section: Given n words $x^1, x^2, \ldots, x^n$ over alphabet $\Sigma = \{1, \ldots, m\}$ sort them according to alphabetic order.

There are many different ways of storing words. One popular method stores all words in a common storage area called string space. The characters of any word are stored in consecutive storage locations. Each string variable then consists of a pointer into the string space and a location containing the current length of the word. Figure 28 shows an example for $x^1 = ABD$, $x^2 = DBA$ und $x^3 = Q$.

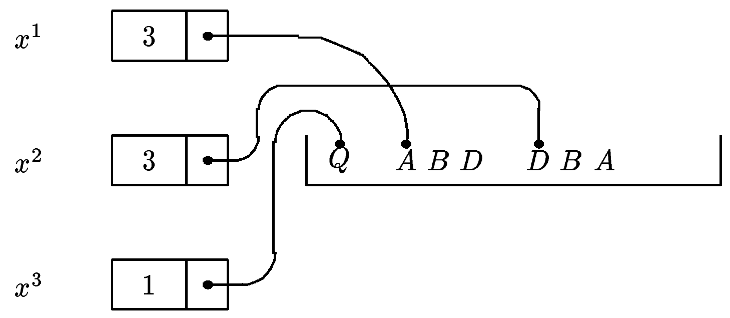


Figure 28. Storing 3 words

The basic idea of **bucketsort** is most easily explained when we consider a special case first: all words $x^1, \dots, x^n$ have length 1, i.e., the input consists of n numbers between 1 and m. Bucketsort starts with m empty buckets. Then we process number by number, throwing x^i into the x^i-th bucket. Finally, we step through the buckets in order and collect the elements in sorted order.

This description of Bucketsort is fairly poetic in style; let us fill in some implementation details next. Buckets are linear lists, the heads of the lists are stored in array $K[1\,..\,m]$. Initially we have to create m empty lists. This is easily done in time $O(m)$ by initializing array K. In order to process x^i we have to access $K[x^i]$ and to insert x^i into the list with head $K[x^i]$. This can be done in time $O(1)$ if we insert x^i at the front end of the list or at the back end of the list. In the latter case $K[x^i]$ must also contain a pointer to the end of the list. Thus we can distribute n words in time $O(n)$. Finally we have to collect the buckets. We step through array $K[1\,..\,m]$ and concatenate the front of the $(j+1)$-st list with the end of the j-th list. This takes time $O(m)$, if array K also contains pointers to the back ends of the lists, and time $O(n+m)$ otherwise. Note that the total length of all m lists is n. In either case total running time is $O(n+m)$. We have to discuss one more detail before we proceed. Should we add x^i to the front or to the rear end of the x^i-th list? If we always add to the rear, then the order of elements x^i, x^j with $x^i = x^j$ is unchanged, i.e., bucketsort is stable. This will be important for what follows. i.D.anders

We proceed to a slightly more difficult case next. The x^i, $1 \le i \le n$, are proper words and all of them have equal length $|x^i| = k$ for $1 \le i \le n$. Then $x^i = x^i_1 x^i_2 \dots x^i_k$ with $x^i_j \in \Sigma$. There are two ways of extending our basic algorithm to the more general case.

In the first approach we sort according to the first letter first. This divides our set of words into m groups, some of which may be empty. Then we sort each group separately according to the second character, ... until each group consists of a single word only. This approach has a serious disadvantage. Groups become smaller and smaller, but the complexity of bucketsort is at least the size of the alphabet, i.e., the overhead may be large for small group size. It is shown in Exercise 18 that total running time of this approach may be as large as $\Omega(n \cdot k \cdot\ m)$.

In the second approach one sorts according to the last (k-th) letter first. After having done so we sort the entire list of n words, which is sorted according to the last letter, according to the next to last letter. The crucial observation is: The words are sorted to the last two letters now, because bucketsort is stable. Next we sort according to the $(k-2)$-th letter, and so on. The second approach requires k passes over the set of n words, each pass having a cost of $O(n+m)$ time units. Thus total running time is $O(k \cdot (n+m))$.

Can we improve this? Let us consider an example with $m = 4$, $n = 5$ and $k = 3$. The 5 words are 123, 124, 223, 324, 321. The first pass yields:

bucket one :	321
bucket two :	
bucket three:	123, 223
bucket four :	124, 324

And hence the input sequence for the second pass is 321, 123, 223, 124, 324. The second pass yields:

bucket one :	
bucket two :	321, 123, 223, 124, 324
bucket three:	
bucket four :	

And hence the input sequence for the third is 321, 123, 223, 124, 324. The third pass yields:

bucket one :	123, 124
bucket two :	223
bucket three:	321, 324
bucket four :	

And hence the final result sequence is 123, 124, 223, 321, 324.

Note that we collected a total of $3 \cdot 4 = 12$ buckets, but only 7 buckets were non-empty altogether. It would improve running time if we knew ahead of time which buckets to collect in each pass. Let us assume that s_j buckets are non-empty in the j-th pass, $1 \le j \le k$. If we could avoid considering empty buckets then the cost of the j-th pass would be $O(n + s_j)$, since $s_j \le n$, the cost of a pass would be only $O(n)$ instead of $O(n + m)$.

There is a very simple method for determining which buckets are non-empty in i.D.mehr the j-th pass, i.e., which letters occur in the j-th position. Create set $\{(j, x_j^k);\ 1 \le j \le k,\ 1 \le i \le n\}$ of size $n \cdot k$ and sort it by bucketsort according to the second component and then according to the first. Then the j-th bucket contains all characters which appear in the j-th position in sorted order. The cost of the first pass is $O(n \cdot k + m)$, the cost of the second is $O(n \cdot k + k)$. Total cost is thus $O(n \cdot k + m)$.

Before we give a complete algorithm for bucketsort we discuss the extension to words of arbitrary length. Let $x^i = x_1^i \dots x_{l_i}^i$, $1 \le i \le n$; l_i is the length of x_i. We basically proceed as above, however we have to make sure that x_i has to be considered for the first time when we sort according to the l_i-th letter. So we sort the words according to their length first. This leads to the following algorithm. Let $L = \sum l_i \ge n$.

1) Determine the length of x^i, $1 \le i \le n$, and create pairs (l_i, pointer to x^i). This takes time $O(L)$.

2) Sort the pairs (l_i, pointer to x^i) by bucketsort according to the first component. Then the k-th bucket contains all x^i with $l_i = k$, i.e., all these strings are contained in linear list. Call this list *length*$[k]$. The cost of step 2) is $O(n + L)$ because L buckets certainly suffice.

3) Create L pairs (j, x_j^i), $1 \le i \le n$, $1 \le j \le l_i$, and sort them according to the second and then according to the first component. Let *Nonempty*$[j]$, $1 \le j \le l_{max} = \max(l_1, \dots, l_n)$ be the j-th bucket after the second pass. *Nonempty*$[j]$

contains all letters in $\{1, \ldots, m\}$ which appear in the j-th position in sorted order. Delete all duplicates from *Nonempty*$[j]$. The cost of step 3) is $O(L + m) + O(L + l_{max}) = O(L + m)$.

4) We finally sort words x^i by distribution. All lists in Program 15 are used as queues; the head of each list contains a pointer to the last element. Also, x is a string variable and x_j is the j-th letter of string x.

```
(1)  $W \leftarrow$ empty queue;
(2)  **for** $k$ **from** 1 **to** $m$ **do** $S[k] \leftarrow$ empty queue **od**;
(3)  **for** $j$ **from** $l_{max}$ **to** 1
(4)  **do** add *length*$[j]$ to the front of $W$ and call the new queue $W$;
(5)      **while** $W \neq \emptyset$
(6)      **do** let $x$ be the first element of $W$, delete $x$ from $W$;
(7)          add $x$ to the end of $S[x_j]$
         **od**;
(8)      **while** *Nonempty*$[j] \neq \emptyset$
(9)      **do** let $k$ be the first element of *Nonempty*$[j]$;
(10)         delete $k$ from *Nonempty*$[j]$;
(11)         add $S[k]$ to the end of $W$;
(12)         set $S[k]$ to the empty queue
(13)     **od**
(14) **od**.
```

Program 15

Correctness of this algorithm is obvious from the preceding discussion. Line (2) costs $O(m)$ time units, single execution of any other line has a cost of $O(1)$. Note that only a pointer to string x is moved in line (7). Lines (3) and (4) are executed l_{max} times. In lines (5)–(7) we operate exactly l_i times on string x^i. Hence the total cost of these lines is $O(L)$. We associate the cost of a single execution of lines (9)–(12) with the first element of $S[k]$; k as in line (9). In this way we associate at most $O(l_i)$ time units with x^i, $1 \leq i \leq n$, and hence the total cost of lines (8)–(12) is $O(L)$. Altogether we have shown that the total cost of step (4)–(12) is $O(m + L)$.

Theorem 1. *Bucketsort sorts n words of total length L over an alphabet of size m in time $O(m + L)$.* ∎

2.2.2. Sorting Reals by Distribution

We briefly describe distribution sort applied to real numbers. For simplicity, we assume that we are given a sequence x_i, $1 \leq i \leq n$, of reals from the interval $(0, 1]$.

We use the following very simple algorithm, called **Hybridsort**. α is some fixed real and k is equal to $\alpha \cdot n$.

1) Create k empty buckets. Put x_i into bucket $\lceil k \cdot x_i \rceil$ for $1 \le i \le n$.
2) Apply Heapsort to every bucket and concatenate the buckets.

The correctness of this algorithm is obvious.

Theorem 2. a) *Worst case running time of Hybridsort is $O(n \log n)$.*

b) *If the x_i's are drawn independently from a uniform distribution over the interval $(0, 1]$, then Hybridsort has expected running time $O(n)$.*

Proof: a) Running time of the first phase is clearly $O(n)$. Let us assume that t_i elements end up in the i-th bucket, $1 \le i \le k$. Then the cost of the second phase is $O(\sum_i t_i \log t_i)$, where $0 \log 0 = 0$ and $\sum_i t_i = n$. But $\sum_i t_i \log t_i \le \sum_i t_i \log n = n \log n$.

b) Let B_i be the random variable representing the number of elements in the i-th bucket after pass 1. Then the probability that $B_i = h$ is defined by

$$prob(B_i = h) = \binom{n}{h} \cdot \left(\frac{1}{k}\right)^h \cdot \left(1 - \frac{1}{k}\right)^{n-h}$$

since any single x_j is in the i-th bucket with probability $1/k$. Expected running time of phase 2 is

$$\begin{aligned}
&E\left(\sum_{i=1}^{k} B_i \log B_i\right) \\
&= k \sum_{h=2}^{n} (h \log h) \cdot \binom{n}{h} \cdot \left(\frac{1}{k}\right)^h \cdot \left(1 - \frac{1}{k}\right)^{n-h} \\
&\le k \sum_{h=2}^{n} h^2 \cdot \binom{n}{h} \cdot \left(\frac{1}{k}\right)^h \cdot \left(1 - \frac{1}{k}\right)^{n-h} \\
&= k \cdot \left(\frac{n(n-1)}{k^2} + \frac{n}{k}\right) \\
&= O(n).
\end{aligned}$$

The next to last equation holds since $h^2\binom{n}{h} = (h(h-1) + h)\binom{n}{h} = n(n-1)\binom{n-2}{h-2} + n\binom{n-1}{h-1}$ and the last one since $k = \alpha \cdot n$. ∎

2.3. Lower Bounds on Sorting, Revisited

In Section 2.1.6 we proved an $\Omega(n \log n)$ lower bound on the average and worst case complexity of sorting. However, in 2.2 we showed an $O(n)$ upper bound on the average case complexity of sorting reals. What went wrong? An upper bound which is smaller than the corresponding lower bound? The answer is that we have to keep the models of computation in mind for which we proved the bounds.

The lower bound of 2.1.6 was shown for rational decision trees. In rational decision trees we can compare rational functions of the inputs at every step. Hybridsort uses a larger set of primitives; in particular it uses division, rounding and indirect addressing. Note that indices into arrays are computed as functions of the inputs. Also, the upper bound on running time is expected case and not worst case. It is still an open problem whether any variant of Hybridsort can sort real numbers in linear worst time.

However, it can be shown that any RAM (in the sense of 1.1) which operates on natural numbers and has basic operations $+, \ominus$ (subtraction in $\mathbb{N}_0$), $\times$ in addition to the comparison operator $\leq$ requires time $\Omega(n \log n)$ in the unit cost measure for sorting in the worst case. This is even true for coin tossing RAMs (in the sense of 1.2). It is still an open problem whether the result holds true if division is also a basic operation.

Before we state the precise result we briefly recall the RRAM model. It is convenient to use indirect addressing instead of index registers as a means for address calculations. We saw in 1.1 that this does not affect the running time by more than a constant; however it makes the proof more legible. The basic operations are $(i, k, k_1, k_2 \in \mathbb{N}_0)$:

$$\alpha \leftarrow i, \qquad \alpha \leftarrow \rho(i), \qquad \alpha \leftarrow \rho(\rho(i)),$$

$$\rho(i) \leftarrow \alpha, \qquad \rho(\rho(i)) \leftarrow \alpha,$$

$\alpha \leftarrow \alpha$ op $\rho(i)$, where op $\in \{+, \times, \ominus\}$,

goto k,

if $\rho(i) > \alpha$ **then goto** k_1 **else goto** k_2 **fi**,

$\alpha \leftarrow$ random

Here

$$x \ominus y = \begin{cases} x - y & \text{if } x \geq y; \\ 0 & \text{if } x < y, \end{cases}$$

and $\alpha \leftarrow$ random assigns 0 or 1 to α with probability 1/2, each. A RRAM program is a sequence of numbered instructions starting at 0. The labels in the **goto** instructions refer to this numbering.

Definition: Let A be a RRAM. A **solves the sorting problem** of size n if for all $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ with $x_i \neq x_j$, for $i \neq j$: if A is started with x_i in location i, $1 \leq i \leq n$, and 0 in the accumulator and all locations j, $j > n$, and if π is a permutation such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ then A stops and $x_{\pi(i)}$ is in location i, $1 \leq i \leq n$, when A stops. ▮

Theorem 1. *Let A be any RRAM which solves the sorting problem of size n. Then there is an $x \in \mathbb{R}^n$ such that the expected running time of A on x is at least* $\log n!$, *i.e., the worst case running time of A is at least* $\log n!$.

Proof: Let A be a RRAM which solves the sorting problem of size n and stops on all inputs $x \in \mathbb{R}^n$ and all sequences of coin tosses in at most t steps. Here t is some integer. The proof is by reduction to the decision tree model studied in Section 2.1.6. The first step is to replace all instructions of the form $\alpha \leftarrow \alpha \ominus \rho(i)$ by the equivalent sequence

$$\textbf{if } \rho(i) \geq \alpha \textbf{ then } \alpha \leftarrow 0 \textbf{ else } \alpha \leftarrow \alpha - \rho(i) \textbf{ fi}.$$

After the modification A uses only true subtractions.

As a second step we associate with program A a (maybe infinite) decision tree $T(A)$ by unfolding. $T(A)$ is a tree of degree ≤ 2 whose nodes are labelled by instructions and tests and some of whose edges are labelled by 0 or 1. More precisely, let A consists of m instructions. Then $T_j(A)$, $1 \leq j \leq m+1$ is defined by

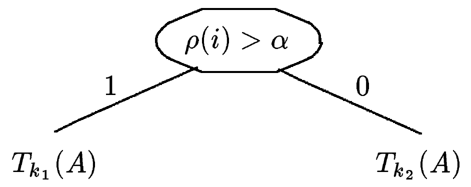
(1) $T_{m+1}(A)$ is a single leaf

(2) if line j of A is instruction $\alpha \leftarrow i$, $\alpha \leftarrow \rho(i)$, $\alpha \leftarrow \rho(\rho(i))$, $\rho(i) \leftarrow \alpha$, $\rho(\rho(i)) \leftarrow \alpha$ or $\alpha \leftarrow \alpha$ op $\rho(i)$ then $T_j(A)$ consists of a root labelled by that instruction with a single child. The single subtree is $T_{j+1}(A)$.

(3) if line j is **goto** k then $T_j(A)$ is equal $T_k(A)$.

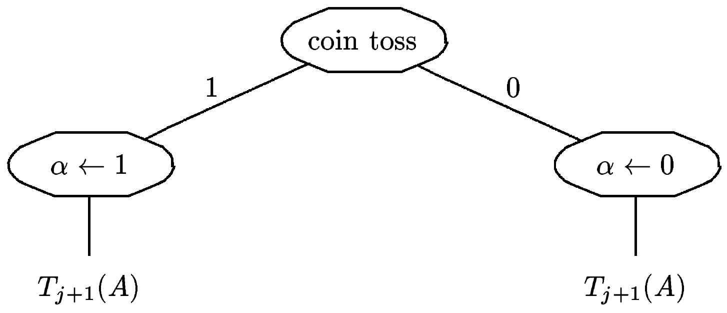
(4) if line j is **if** $\rho(i) > \alpha$ **then goto** k_1 **else goto** k_2 **fi**, then

$T_j(A) =$



(5) if line j is $\alpha \leftarrow$ random then

$T_j(A) =$



Finally $T(A) = T_1(A)$.

Tree $T(A)$ simulates RRAM A in an obvious way; in coin tossing nodes the two outgoing edges are taken with equal probability. Tree $T(A)$ might be infinite but because A stops on every input $x \in \mathbb{R}^n$ within t steps no node of depth $> 2t$ is ever reached in any computation. Therefore we may prune $T(A)$ at depth $2t$.

As a third step we associate with every edge e of the decision tree a set $\{p_{e,i}\,;\ i \in \mathbb{N}_0\}$ of polynomials in indeterminates $X_1, \ldots, X_n$. The intended meaning of $p_{e,i}$ is the content of location i (the accumulator for $i = 0$) if the input is $x \in \mathbb{R}^n$ and control has reached edge e. Polynomials $p_{e,i}$ will not have the intended meaning for all inputs x but only for a sufficiently large set of nice inputs as made precise below. If p and q are polynomials we write $p \equiv q$ to denote the fact that p and q define the same rational function. The inductive definition is started by:

If e is the (conceptual) edge entering the root then

$$p_{e,i} = \begin{cases} X_i & \text{if } 1 \le i \le n; \\ 0 & \text{otherwise.} \end{cases}$$

For the induction step let e be an edge leaving node v, let d be the edge into v and let $ins(v)$ be the label of node v.

(1) if $ins(v)$ is $\rho(i) > \alpha$ or coin toss then $p_{e,i} = p_{d,i}$ for all $i \ge 0$.

(2) if $ins(v)$ is $\alpha \leftarrow \alpha$ op $\rho(i)$, op $\in \{+, -, \times\}$ then

$$p_{e,j} = \begin{cases} p_{d,0} \text{ op } p_{d,i} & \text{if } j = 0; \\ p_{d,j} & \text{otherwise.} \end{cases}$$

(3) if $ins(v)$ is $\alpha \leftarrow i$ then

$$p_{e,j} = \begin{cases} i & \text{if } j = 0; \\ p_{d,j} & \text{otherwise.} \end{cases}$$

(4) if $ins(v)$ is $\alpha \leftarrow \rho(i)$ or $\alpha \leftarrow \rho(\rho(k))$ and $p_{d,k} \equiv i$ for some $i \in \mathbb{N}_0$ then

$$p_{e,j} = \begin{cases} p_{d,i} & \text{if } j = 0; \\ p_{d,j} & \text{otherwise.} \end{cases}$$

(5) if $ins(v)$ is $\alpha \leftarrow \rho(\rho(k))$ and $p_{d,k} \not\equiv i$ for all $i \in \mathbb{N}_0$ then let w be the closest ancestor (i.e., an ancestor of maximal depth) of v such that the label of w is $\rho(\rho(l)) \leftarrow \alpha$ for some $l \in \mathbb{N}$ and $p_{f,l} \equiv p_{d,k}$ where f is the edge entering w. If w does exist then

$$p_{e,j} = \begin{cases} p_{f,0} & \text{if } j = 0; \\ p_{d,j} & \text{otherwise,} \end{cases}$$

and if w does not exist then

$$p_{e,j} = \begin{cases} 0 & \text{if } j = 0; \\ p_{d,j} & \text{otherwise.} \end{cases}$$

(6) if $ins(v)$ is $\rho(i) \leftarrow \alpha$ or $\rho(\rho(k)) \leftarrow \alpha$ and $p_{d,k} \equiv i$ for some $i \in \mathbb{N}_0$ then

$$p_{e,j} = \begin{cases} p_{d,0} & \text{if } j = i; \\ p_{d,j} & \text{otherwise.} \end{cases}$$

(7) if $ins(v)$ is $\rho(\rho(k)) \leftarrow \alpha$ and $p_{d,k} \not\equiv i$ for all $i \in \mathbb{N}_0$, then $p_{e,j} = p_{d,j}$ for all j.

We will next derive a sufficient condition for the "correctness" of the intended meaning of polynomials $p_{e,j}$. Let

$$\begin{aligned} I = \ & \{1, \ldots, n\} \\ & \cup \{j \in \mathbb{N};\ \text{there is some instruction with address } j \text{ in } T(A) \\ & \qquad \text{or } p_{e,i} \equiv j \text{ for some edge } e \text{ and } i \in \mathbb{N}_0\}. \end{aligned}$$

Note that I is finite since for every edge e there are only finitely many (at most $n +$ depth of e) i's with $p_{e,i} \not\equiv 0$. This is easily seen by induction on the depth of e. Let Q_A, a set of polynomials, be defined by

$$Q_A = I \cup \{p_{e,i};\ e \text{ is edge of } T(A) \text{ and } i \geq 0\}.$$

Q_A is a finite set of polynomials by the remark above. A point $x \in \mathbb{R}^n$ is **admissible** if $p(x) \neq q(x)$ for any $p, q \in Q_A$ with $p \not\equiv q$.

Lemma 1. *If $x \in \mathbb{R}^n$ is admissible for Q_A then all $p_{e,j}$'s have their intended meaning for all edges e and all $j \in I$, i.e., $p_{e,j}(x)$ is the content of location j if the input is x and control reaches e for all $j \in I$.*

Proof: The proof is by induction on the depth of e. The claim is clearly true for the edge entering the root. So let us consider the case that edge e emanates from node v.

Let d be the edge entering v. The induction step is by case analysis paralleling the definition of $p_{e,j}$. The induction step is easy in cases (1), (2), (3), (4) and (6) and therefore left to the reader. The two remaining cases are cases (5) and (7).

(5): Let w, f and l be defined as above. Let us consider the case that w exists first. Let $h = p_{f,l}(x) = p_{d,k}(x)$. Then $p_{f,0}(x)$ was stored in location h by instruction $ins(w)$. It remains to be shown that location h was not written into on the path from w to v. Let z be any node on that path excluding v and w. If $ins(z)$ is not a store instruction then there is nothing to show.

So let us assume that $ins(z)$ is either $\rho(i) \leftarrow \alpha$ or $\rho(\rho(i)) \leftarrow \alpha$. In the former case we have $i \neq h = p_{d,k}(x)$ since $i \in I \subseteq Q_A$, x is admissible for Q_A, and $p_{d,k}$ is not constant, in the latter case we have $p_{g,i}(x) \neq h = p_{f,l}(x)$, where g is the edge entering z, since $p_{g,i} \in Q_A$ and x is admissible for Q_A. In either case $ins(z)$ does not write into location z.

If w does not exist then one proves in the same way that nothing was ever written into location $h = p_{d,k}(x)$ before control reaches edge e.

(7): Since $ins(v)$ is $\rho(\rho(k)) \leftarrow \alpha$ and $p_{d,k} \not\equiv i$ for all $i \in N_0$. Hence $p_{d,k}(x) \notin I$ since x is admissible for Q_A and $I \subseteq Q_A$. Thus the content of locations with addresses in I is not changed by $ins(v)$. ∎

Let π be a permutation of $\{1, 2, \ldots, n\}$. $x \in \mathbb{R}^n$ is of order type π if $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$. Let

$$P = |\{\pi;\ \text{there is a } x \in \mathbb{R}^n \text{ of order type } \pi \text{ which is admissible for } Q_A\}|.$$

We show below that $P = n!$. Let $x(\pi)$ be admissible for Q_A of order type π. Execution of $T(A)$ on input $x(\pi)$ ends in some leaf with ingoing edge e; of course, the leaf depends on sequences $s \in \{0,1\}^{2t}$ of coin tosses used in the computation. We have $p_{e,i}(x(\pi)) = x_{\pi(i)}(\pi)$ for all $i \in \{1, \ldots, n\}$. Since $X_1, \ldots, X_n \in Q_A$ and $x(\pi)$ is admissible for Q_A this is only possible if $p_{e,i} \equiv X_{\pi(i)}$ for all $i \in \{1, \ldots, n\}$. Thus for every sequence $s \in \{0,1\}^{2t}$ of coin tosses at least $n!$ different leaves are reachable, one for each order type. Let $d(\pi, s)$ be the number of nodes of outdegree 2 on the path from the root to the leaf which is reached on input $x(\pi)$ when sequence s of coin tosses is used. Then

$$\frac{1}{n!} \sum_{\pi} d(\pi, s) \geq \log n!$$

(compare the proof of 2.1.6, Theorem 8) for every $s \in \{0,1\}^{2t}$ and hence

$$\frac{1}{2^{2t}} \sum_{s \in \{0,1\}^{2t}} \frac{1}{n!} \sum_{\pi} d(\pi, s) \ \geq\ \log n!$$

or

$$\frac{1}{n!} \sum_{\pi} \frac{1}{2^{2t}} \sum_{s \in \{0,1\}^{2t}} d(\pi, s) \ \geq\ \log n!\,.$$

Thus there must be some π such that

$$\frac{1}{2^{2t}} \sum_{s \in \{0,1\}^{2t}} d(\pi, s) \;\geq\; \log n!\,,$$

i.e., the expected running time of A on input $x(\pi)$ is at least $\log n! \geq n \log n - 2n$. It remains to prove that $P = n!$. Q_A is a finite set of polynomials. x is admissible for Q_A if $p(x) \neq q(x)$ for any pair $p, q \in Q_A$ with $p \not\equiv q$, i.e., if $R_A(x) \neq 0$ where

$$R_A = \prod \{p - q;\ p, q \in Q_A,\ p \not\equiv q\}.$$

R_A is a single polynomial. It therefore suffices to prove Lemma 2.

Lemma 2. *Let $R \not\equiv 0$ be a polynomial of degree m in variables $\{X_1, \ldots, X_n\}$ and let π be any permutation of $\{1, \ldots, n\}$. Then there is an $x \in \{1, \ldots, n+m\}^n$ of order type π with $R(x) \neq 0$.*

Proof: Since $R(X_{\pi(1)}, \ldots, X_{\pi(n)})$ is again a polynomial of degree m it suffices to prove the claim for the case that π is the identity permutation. This is done by induction on n. For $n = 1$ the claim follows from the fact that a univariate polynomial of degree m has at most m zeroes. For the induction step let

$$R(X) = \sum_{i=0}^{s} R_i(X_1, \ldots, X_{n-1}) X_n^i$$

where $R_s \not\equiv 0$. Let r be the degree of R_s. Then $r + s \leq m$. By induction hypothesis there is some $x \in \{1, \ldots, r+n-1\}^{n-1}$, $x_1 < x_2 < \cdots < x_{n-1}$ with $R_s(x) \neq 0$. Choose $x_n \in \{r+n, \ldots, n+r+s\}$ such that the s-th degree polynomial

$$\sum_{i=0}^{s} R_i(x_1, \ldots, x_{n-1}) X_n^i$$

in X_n is non-zero. ▮▮

Similar lower bounds can be shown for related problems, e.g., the nearest neighbour problem (Exercise 20) and the problem of searching an ordered table (Exercise 21). It is open whether Theorem 1 still holds true if integer division is added as an additional primitive operation. It should be noted however that one can sort in linear time (in the unit cost model) if bitwise boolean operations (componentwise negation and componentwise AND of register contents, which are imagined to be binary representations of numbers) and integer division are additional primitives.

Theorem 2. *RAMs with integer division and bitwise boolean operations can sort n numbers in time $O(n)$ in the unit cost model.*

Proof: The idea of the proof is to exploit the parallelism provided by bitwise boolean operations and to use integer division for cutting large numbers into pieces. The details are as follows.

Let $x_1, \ldots, x_n$ be integers. We first find the maximum, say x_m, compute $x_m \vee \overline{x_m}$ (bitwise) and add 1. This produces 10^k where k is the length of the binary representation of x_m. Note that 10^k is the binary representation of 2^{k+1}. We shall compute the rank of every x_i, i.e., compare each x_i with all x_j and count how many x_j are smaller than x_i. Note that if we have

$$\begin{array}{ll} A & \ldots 1\ x_i\ 1 \ldots \\ B & \ldots \underset{\substack{\uparrow \\ \text{indicator}}}{0}\ x_j\ 0 \ldots \end{array}$$

in registers A and B, with x_i and x_j in matching positions, then regardless of the rest of registers A and B, $C = A - B$ will contain a 1 at the indicated position (indicator bit) $\iff$ $x_i \geq x_j$. So, for sorting, we obtain in a single register n copies of $1x_n11x_{n-1}11x_{n-2}\ldots 1x_11$ concatenate to each other, and $(0x_n0)^n(0x_{n-1}0)^n \ldots (0x_10)^n$ in another with the length of all x_i's padded out to k with leading zeroes. This can be done in time $O(n)$ as follows. We show the construction for $(0x_n0)^n(0x_{n-1}0)^n \ldots (0x_10)^n$. Note first that we have 2^{k+1} available and therefore can compute $a \leftarrow 2^{(k+2)n}$ in time $O(n)$. Next note that $b = \sum_{i=1}^{n} x_i 2^{(k+2)n(i-1)+1}$ has binary representation

$$\underbrace{0\ldots 0x_n0}_{(k+2)n \text{ bits}} \qquad \underbrace{0\ldots 0x_{n-1}0}_{(k+2)n \text{ bits}} \qquad \ldots \qquad \underbrace{0\ldots 0x_10}_{(k+2)n \text{ bits}}$$

and can be computed in time $O(n)$ given a, $x_1, \ldots, x_n$. Finally observe, that $\sum_{i=0}^{n-1} b \cdot 2^{(k+2)i}$ is the desired result.

At this point, a single subtraction yields all comparisons. An AND with bit string $(10^{k+1})^{n^2}$, which can be computed in time $O(n)$, retrieves all the indicator bits. More precisely, we have a bitstring of length $(k+2)n^2$ such that bit $(k+2)n(j-1) + (k+2)i$ is one $\iff$ $x_i \geq x_j$ and all other bits are zero. We cut this bit string into pieces of length $(k+2)n$ by repeatedly dividing it by $2^{(k+2)n}$ and sum all the pieces. This yields

$$\text{rank}(x_n)\ldots \qquad \underbrace{0\ldots 0\text{rank}(x_2)}_{(k+2) \text{ bits}} \qquad \underbrace{0\ldots 0\text{rank}(x_1)}_{(k+2) \text{ bits}} \qquad \underbrace{0\ldots 0}_{(k+1) \text{ bits}}$$

The ranks are now easily retrieved and the sort is completed in time $O(n)$. ∎

2.4. The Linear Median Algorithm

Selection is a problem which is related to sorting but which is simpler. We are given a sequence $S_1, \dots, S_n$ of pairwise distinct elements and an integer i, $1 \le i \le n$, and want to find the i-th smallest element of the sequence, i.e., an S_j such that there are $i-1$ keys S_l with $S_l < S_j$ and $n-i$ keys S_l with $S_l > S_j$. For $i = \lfloor n/2 \rfloor$ such a key is called **median**. Of course, selection can be reduced to sorting. We might first sort sequence $S_1, \dots, S_n$ and then find the i-th smallest element by a single scan of the sorted sequence. This results in an $O(n \log n)$ algorithm. However, there is a linear time solution. We describe a simple, linear expected time solution (procedure *Find*) first and then extend it to a linear worst case time solution (procedure *Select*).

Procedure *Find* in Program 16 is based on the partitioning algorithm used in Quicksort. We choose some element of the sequence, say S_1, as partitioning element and then divide the sequence into the elements smaller than S_1 and the elements larger than S_1. We then call *Find* recursively on the appropriate subsequence.

```
     procedure Find($M, i$);
     co finds the $i$-th smallest element of set $M$ oc
(1)  begin
(2)  $S \leftarrow$ some element of $M$;
(3)  $M_1 \leftarrow \{m \in M;\ m < S\}$;
(4)  $M_2 \leftarrow \{m \in M;\ m > S\}$;
(5)  case $|M_1|$ of
(6)     $< i-1$ :  Find($M_2, i - |M_1| - 1$);
(7)     $= i-1$ :  return $S$;
(8)     $> i-1$ :  Find($M_1, i$)
(9)  esac
(10) end.
```

Program 16

When set M is stored in an array then lines (2)–(4) of *Find* are best implemented by lines (2)–(6) of Quicksort. Then a call of *Find* has cost $O(|M| +$ the time spent in the recursive call). The worst case running time of *Find* is clearly $O(n^2)$. Consider the case $i = 1$ and $|M_1| = |M| - 1$ always.

Expected running time of algorithm *Find* is linear as we show next. We use the same randomness assumption as for the analysis of Quicksort, i.e., the elements of M are pairwise distinct and each permutation of the elements is equally likely. In particular, this implies that the element S chosen in line (2) is the k-th largest element of M with probability $1/|M|$. It also implies that both subproblems M_1 and M_2 again satisfy the randomness assumption (cf. the analysis of Quicksort).

Let $T(n, i)$ be the expected running time of *Find*(M, i), where $|M| = n$ and let

$T(n) = \max_i T(n,i)$. We have $T(1) = c$ for some constant c and

$$T(n,i) \le c \cdot n + \frac{1}{n}\left[\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k-1, i)\right],$$

since the partitioning process takes time $c \cdot n$ and the recursive call takes expected time $T(n-k, i-k)$ if $k = |M_1| + 1 < i$ and time $T(k-1, i)$ if $k = |M_1| + 1 > i$. Thus

$$T(n) \le c \cdot n + \frac{1}{n} \max_i \left[\sum_{k=1}^{i-1} T(n-k) + \sum_{k=i}^{n-1} T(k)\right].$$

We show $T(n) \le 4\,c \cdot n$ by induction on n. This is clearly true for $n = 1$. For $n > 1$ we have

$$\begin{aligned} T(n) &\le c \cdot n + \frac{1}{n} \max_i \left[\sum_{k=n-i+1}^{n-1} 4\,c \cdot k + \sum_{k=i}^{n-1} 4\,c \cdot k\right] \\ &\le c \cdot n + \frac{4\,c}{n} \max_i \left[n(n-1) - (n-i)(n-i+1)/2 - i(i-1)/2\right] \\ &\le 4\,c \cdot n, \end{aligned}$$

since the expression in square brackets is maximal for $i = \lceil (n+1)/2 \rceil$ (note that the expression is symmetric in i and $n - i + 1$) and then has value $n(n-1) - (n+1)(n-1)/4 \le 3n^2/4$. We have thus shown:

Theorem 1. *Algorithm Find has linear expected running time.* ▮

The expected linearity of *Find* stems from the fact that the expected size of the subproblem to be solved is only a fraction of the size of the original problem. However, the worst case running time of *Find* is quadratic because the size of the subproblem might only be smaller by one than the size of the original problem. If one wants a linear worst case algorithm one has to choose the partitioning element more carefully.

A first approach is to take a reasonable size sample of M, say of size $|M|/5$, and to take the median of the sample as partitioning element. However, this idea is not good enough yet because the sample might consist of small elements only. A better way of choosing the sample is to divide M into small groups of say 5 elements each and to make the sample the set of medians of the groups. Then it is guaranteed that a fair fraction of elements is smaller (larger) than the partitioning element. This leads to Program 17.

It is very helpful to illustrate algorithm *Select* pictorially. For simplicity we assume that 10 divides n. In line (4) M is divided into groups of 5 elements each and the median of each group is determined in lines (5) and (6). At this point we have $\lceil n/5 \rceil$ linear orders of 5 elements each. Next we find $\overline{m}$, the median of the

```
     procedure $Select(M, i)$;
     co finds the $i$-th smallest element of set $M$ oc
(1)  begin
(2)  $n \leftarrow |M|$;
(3)  if $n \leq 100$
     then sorts $M$ and find the $i$-th smallest element directly
(4)  else divide $M$ in $\lceil n/5 \rceil$ subsets $M_1, \ldots, M_{\lceil n/5 \rceil}$ of 5 elements each;
          co the last subset may contain less than 5 elements oc
(5)       sort $M_j$; $1 \leq j \leq \lceil n/5 \rceil$;
(6)       let $m_j$ be the median of $M_j$;
(7)       call $Select(\{m_1, \ldots, m_{\lceil n/5 \rceil}\}, \lceil \lceil n/5 \rceil /2 \rceil)$ and
          determine $\overline{m}$, the median of the medians;
(8)       let $M_1 = \{m \in M;\ m \leq \overline{m}\}$ and $M_2 = \{m \in M;\ \overline{m} < m\}$;
(9)       if $i \leq |M_1|$
(10)        then $Select(M_1, i)$
(11)        else $Select(M_2, i - |M_1|)$
(12)      fi
(13) fi
(14) end.
```

Program 17

medians. Assume w.l.o.g. that $m_1, \ldots, m_{n/10} \leq \overline{m}$ and $\overline{m} < m_{n/10+1}, \ldots, m_{n/5}$. This may be represented by Figure 29, where each of the groups is represented by a vertical line of 5 elements, the largest element at the top. Note that all elements in the solid rectangle are smaller than $\overline{m}$ and hence belong to M_1 and that all points in the dashed rectangle are at least as large as $\overline{m}$ and hence belong to M_2. Each rectangle contains $3n/10$ points.

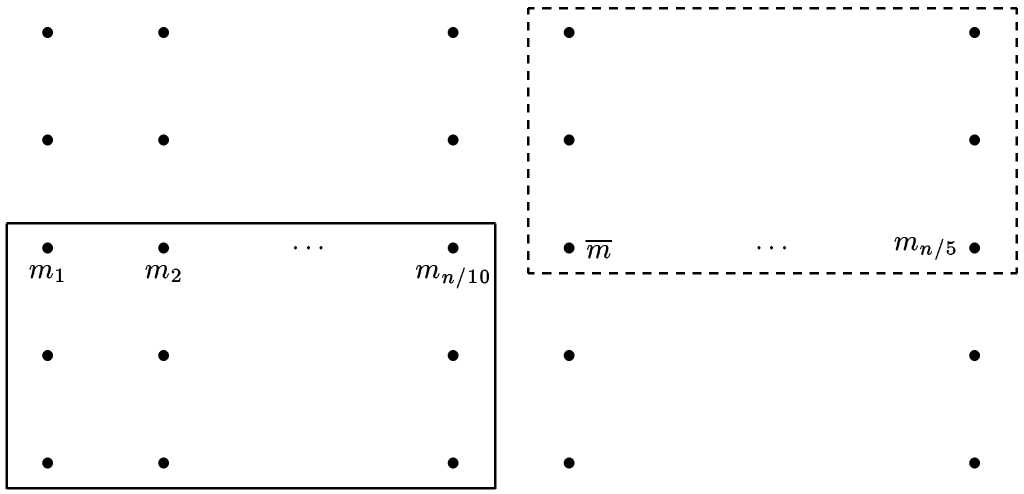


Figure 29. Partitioning of a set by *Select*

Lemma 1. $|M_1|, |M_2| \leq 8n/11$.

Proof: Almost obvious from the discussion above. Note that $|M_1| + |M_2| = n$ and that $|M_1|, |M_2| \geq 3n/10$ if 10 divides n. If 10 does not divide n then $|M_1|, |M_2| \geq 3n/11$ for $n \geq 100$. Details are left to the reader. ▮

Let $T(n)$ be the maximal running time of algorithm *Select* on any set M of n elements and any i.

Lemma 2. *There are constants a, b such that*

$$T(n) \leq \begin{cases} a \cdot n & \text{for } n \leq 100; \\ T(21n/100) + T(8n/11) + b \cdot n & \text{for } n > 100. \end{cases}$$

Proof: The claim is obvious for $n \leq 100$. So let us assume $n > 100$. *Select* is called twice within the body of *Select*, once for a set of $\lceil n/5 \rceil \leq 21n/100$ elements and once for a set of size at most $8n/11$. Furthermore, the total cost of *Select* outside recursive calls is clearly $O(n)$. ▮

Theorem 2. *Algorithm Select works in linear time.*

Proof: We show $T(n) \leq c \cdot n$ where $c = \max(a, 1100\,b/69)$ by induction on n. For $n \leq 100$ there is nothing to show. For $n > 100$ we have

$$\begin{aligned} T(n) &\leq T(21n/100) + T(8n/11) + b \cdot n \\ &\leq c \cdot 21n/100 + c \cdot 8n/11 + b \cdot n \\ &\leq c \cdot n \end{aligned}$$

by definition of c. ▮

2.5. Exercises

1) Let $S[1..n]$ and $P[1..n]$ be two arrays. Let $P[1],\dots,P[n]$ be a permutation of the integers $1,\dots,n$. Describe an algorithm which rearranges S as given by P, i.e., $S_{after}[i] = S_{before}[P[i]]$.

2) Write a RAM program for Heapsort and analyze it.

3) Is there a sequence of n numbers which forces Heapsort to use $2n\cdot(\log(n+1)-1)$ comparisons?

4) When discussing Heapsort we first derived a method which does not keep the heap balanced. Analyze this variant of Heapsort. In particular, treat the following questions. Is it possible to store the tree without explicit parent-child pointers? Storage requirement of the method? Number of comparisons? Running time on a RAM?

5) Discuss Heapsort based on ternary trees.

6) Prove that Procedure Quicksort is called at most $n-1$ times on a problem of size n (use induction). Conclude, that the expected number of calls in lines (7) and (8) is $\frac{n}{2}-1$ each. Also show that line (5) is executed at most $\frac{n}{4}$ times on the average.

7) Translate Quicksort into RAM code and analyze it (use Exercise 6).

8) Do you recommend to modify Quicksort such that a separate list is built for the elements S_i with $S_i = S_1$? Assume that every element S_i occurs exactly k times in the initial sequence. At what value of k does it pay off to partition into three sequences?

9) Solve the following recurrences:

a) $$T(n) = \begin{cases} n^{3/2}\cdot\log n & \text{for } 1 \le n < 10;\\ \sqrt{n}\cdot T(\sqrt{n}) + n^{3/2}\log n & \text{for } n \ge 10.\end{cases}$$

b) $$T(n) = \begin{cases} f(n) & \text{for } 1 \le n < 100;\\ 2\cdot T(n/2) + f(n) & \text{for } n \ge 100.\end{cases}$$

Use $f(n) = n$, $f(n) = n\log n$, $f(n) = n^{3/2}$.

c) $$T(n) = \begin{cases} \sqrt{n} & \text{for } 1 \le n < 10;\\ \log n\cdot T(\log\log n) + \sqrt{n} & \text{for } n \ge 10.\end{cases}$$

10) Write a RASP program for Mergesort and analyze it.

11) Modify Quicksort so that sequences of length at most M are sorted by repeatedly searching for the maximum. What is the best value for M? (Compare the proof of Theorem 2 of Section 5.4). Running time?

12) Would you recommend to use the linear median algorithm of Section 2.4 in the partitioning phase of Quicksort?

13) Prove Theorem 7 of Section 2.1.4 for ternary trees.

14) Let T be any binary tree with n leaves. Interpret the tree as a merging pattern and assume that all initial sequences have length 1. In Section 3.5.3.3, Theorem 15, we give an algorithm which merges two sequences of length x and y in time $O\big(\log\binom{x+y}{y}\big)$. Show that the total cost of merge pattern T is $O(n \log n)$.

15) Does Theorem 10 of Section 2.1.6 hold true if we allow exponentiation as an additional operation?

16) For $(x_1, \ldots, x_n) \in \mathbb{R}^n$ let $f(x_1, \ldots, x_n) = \min\{i;\ x_i \geq x_j \text{ for all } j\}$ be the maximum function. Use Theorem 10 of Section 2.1.6 to prove a $\log n$ lower bound on the depth of rational decision trees for the maximum function. Can you achieve depth $O(\log n)$?

17) For $n \in \mathbb{N}$, $x \in \mathbb{R}$ let

$$f(x) = \begin{cases} 0 & \text{if } x < 0; \\ \lfloor x \rfloor & \text{if } 0 \leq x \leq n; \\ n & \text{if } x > n \end{cases}$$

be the floor-function restricted to interval $[0\,..\,n]$. Use Theorem 10 of Section 2.1.6 to prove a lower bound on the complexity of the floor-function. Does Theorem 10 hold true if the floor-function is added as an additional operation?

18) Let x^i, $1 \leq i \leq n$, be words of length k over alphabet $\Sigma = \{1, \ldots, m\}$. Discuss the following variant of bucketsort. In Phase 1) the input set is divided into m groups according to the first letter. Then the algorithm is applied recursively to each group. Show that the worst case running time of this method is $\Omega(n \cdot m \cdot k)$. Why does the German postal service use this method to distribute the letters?

19) Use Bucketsort to sort n integers in the range $[0\,..\,n^k]$, k fixed, in the worst
i.D.mehr case time $O(n)$. Stellen Sie dazu die Zahlen zur Basis n dar.

20) A program solves the nearest neighbour problem of size n if for all inputs $(x_1, \ldots, x_n, y_1, \ldots, y_n)$ the first n outputs produced by the program are $(x_{i_1}, \ldots, x_{i_n})$ where for all l: $i_l \in \{i;\ |x_i - y_l| \le |x_j - y_l| \text{ for all } j\}$. Show an $n \log n$ lower bound for the nearest neighbour problem in the RRAM model of Section 2.1.6.
i.D. 2.3 [Hint: for π a permutation of $\{1, \ldots, n\}$ say that $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$ is of order type π if for all i, j: $|x_{\pi(i)} - y_i| < |x_j - y_i|$. Modify Lemma 1 in the proof of Theorem 1.]

21) A program solves the searching problem of size n if for all inputs $\{x_1, \ldots, x_n, y\}$ with $x_1 < x_2 < \cdots < x_n$ it computes the rank of y, i.e., it outputs $|\{i;\ x_i < y\}|$.
i.D. 2.3 Prove an $\Omega(\log n)$ lower bound for the RRAM model of Section 2.1.6.

2.6. Bibliographic Notes

Heapsort is by Williams (64) and Floyd (64). Hoare (62) developed Quicksort and procedure *Find*. Sorting by merging and distribution was already known to von Neumann and dates back to methods used on mechanical sorters. Bucketsort as described here is due to Aho/Hopcroft/Ullman (74). Hybridsort is by Meijer/Akl (80). The treatment of recurrences is based on Bentley/Haken/Saxe (80). Theorem 7 (of Section 2.1) is by Huffmann (52), the linear time implementation was described by van Leeuwen (76). The sections on lower bounds are based on A. Schmidt (82) (Theorems 10–12 of 2.1.6) and Hong (79) (Theorem 10–12 of 2.1.6 and Theorem 1 of 2.3), Paul/Simon (80) (Theorems 1 and 2 of 2.3), and Ben-Or (83) (Theorems 13–16 of 2.1.6). Weaker versions of Theorems 15–16 were obtained previously by Yao (81) and Strassen (73) respectively. Finally, the linear time median algorithm is taken over from Blum/Floyd/Pratt/Rivest/Tarjan (72).