Contents

Chapter 1

Sorting

This chapter is in bad shape. I have taken material from the LEDAbook and from my old books from 84. Some of the old material does not run through LATEX yet and hence does not appear in this output.

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 **??**.

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 $\leq$ 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

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

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**??**) 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.

sorting is ubiquitous; describe some uses of sort;

For the sorting algorithms in Sections [**?**] to [**?**] we make the following assumption.

- The elements to be sorted are given in an array A of size n.
- The elements are from a linearly ordered type T and the comparison operators are available on T.
- Output is again in A and A is sorted in increasing order.

arrays have a member function $A.swap(i, j)$ that interchanges the contents of $A[i]$ and $A[j]$.

```
template <class T>
void swap (int i, int j)   // two indices into an array
{ T x  = A[i];
  A[i] = A[j];
  A[j] = x;
}
```

1.1 Basic Sorting Algorithms

do straight insertion sort, do selection sort and bubble sort in the exercises

1.1.1 Selection Sort

Selection Sort follows a simple principle. Determine the smallest element in A and interchange it with $A[0]$. Then determine the smallest element among $A[1]$ to $A[n-1]$ and swap it with $A[1]$. Then determine the smallest element among $A[2]$ to $A[n-1]$ and interchange it with $A[2]$, ...

How do we determine the smallest element among $A[i]$ to $A[n-1]$. We simply run over the elements $A[i]$ to $A[n-1]$ and remember the smallest element encountered (more precisely, we remember its index). The following lines of code realize selection

```
int min = i;
for (int j = i + 1; j < n; j++) if ( A[j] < A[min] ) min = j;
```

and putting one more loop around gives us selection sort.

$\langle$*selection sort*$\rangle \equiv$

```
template<class E>
void selection_sort(array<E>& A)
{
  int n = A.size();
  for (int i = 0; i < n; i++)
  { // we have : A[0] <= A[1] <= ... <= A[i-1] and
    //           A[i-1] <= A[j] for j >= i
    int min = i;
    for (int j = i + 1; j < n; j++) if ( A[j] < A[min] ) min = j;
    A.swap(i,min);
  }
}
```

We have added loop invariants .

Let us next analyse the running time of selection sort. We count the number of comparisons between elements and the number of exchanges. The running time of selection sort (and of any sorting algorithm) to come is O(number of comparisons).

Lemma 1 *Selection sort makes $n(n-1)/2$ comparisons and n exchanges.*

Proof: In the iteration with $i = 0$, we determine the minimum of n elements. This requires $n - 1 = n - 1 - i$ comparisons. In a general iteration, we determine the minimum of $n - i$ elements. This requires $n - i - 1$ comparisons. The total number of comparisons is therefore equal to

$$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 .$$

In every iteration we make one exchange (this may exchange $A[i]$ with itself) and hence the total number of exchanges is equal to n. ∎

The running time of selection sort depends only a little on the actual input sequence. The number of comparisons and exchanges is independent of the input, the number of increments of i and j is independent. It is only the number of assignments to *min* which depends on the actual input. This number is n in the best case, n plus the number of comparisons in the worst case, and is $O(n \log n)$ on average. For an analysis of the latter quantity, we refer the reader to Section **??**.

1.1.2 Checking

Selection sort is so simple that it hardly requires a correctness proof[1] Now, we will explore another venue. Is there a simple way of checking whether selection sort worked correctly. We certainly need

$$A[0] \leq A[1] \leq \ldots \leq A[n-1],$$

and this post-condition is easy to check.

⟨*sorting check*⟩≡

```
template <class E>
bool is_non_decreasing(array<E> A)
{ int n = A.size();
  for (int i = 1; i < n; i++)
    if ( A[i - 1] > A[i] ) return false;
  return true;
}
```

We also need that the array stores the same values as at the beginning. We know of no simple way of checking this condition. In fact, it can be shown that checking whether two arrays contains the same set of values is as difficult as sorting the arrays (see exercise **??**). Fortunately, there is no need to check the condition, because of the following simple observations: (1) the only operation in selection sort which writes into the array to be sorted is the operation *swap*, and (2) swap leaves the set of values in the array unchanged. We conclude that any sequence of swaps does not change the set of values in our array. The array may not be sorted at end of selection sort, but it certainly contains the same set of values. Thus we only need to check whether the array is sorted in non-decreasing order.

1.1.3 Straight Insertion Sort

Straight insertion sort is almost as simple as selection sort. It also works incrementally. In selection sort we repeatedly selected the minimum and, in this way, constructed the subsequence of the smallest i elements for increasing values of i. In straight insertion sort the idea is to . sort the subarray consisting of the first i elements of A for increasing values of i.

The initialization is trivial. The array consisting only of the first element $A[0]$ is sorted. Assume now that we have already sorted an initial segment $A[0], \ldots, A[i-1]$ of A for some $i \geq 1$. Initially, $i = 1$. In the incremental step we add $A[i]$ to the sorted initial segment by inserting it at the proper position. We determine j with $A[j] \leq A[i] < A[j+1]$, move $A[j+1], \ldots, A[i-1]$ one position to the right, and put $A[i]$ into $A[j+1]$, see Figure 1.1. The following lines determine j.

```
int j = i - 1;
while ( A[i] < A[j] ) j--;
```

[1] Many readers will probably think that the correctness of our correctness proof is harder to see than the correctness of the algorithm.

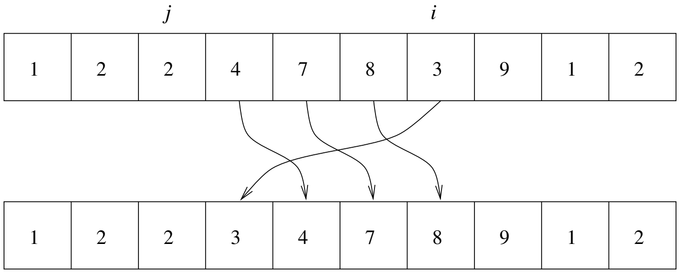

Figure 1.1: We insert $A[i]$ into the already sorted initial segment by inserting it into the proper position, say position $j+1$, and moving elements $A[j+1], \ldots, A[i-1]$ one element to the right.

We simply start with $j = i - 1$ and decrement j as long as $A[i]$ is smaller than $A[j]$. The program terminates with the desired value of j.

Not quite. When $A[i]$ is smaller than all preceding elements, the loop will fall off the left end of the array. We should therefore add the guard `j >= 0` to the loop condition.

```
int j = i - 1;
while ( j >= 0 && A[i] < A[j] ) j--;
```

Assume, we have determined j. We now have to move the elements $A[j+1], \ldots, A[i-1]$ one position to the right. The easiest way to do this, is to incorporate the date movement into the search loop. Of course, we do not want to loose $A[i]$ and therefore store it in a temporary. We obtain

```
E x = A[i];
int j = i - 1;
while ( j >= 0 && x < A[j] )
{ A[j+1] = A[j];
  j--;
}
A[j+1] = x;
```

The last statement puts (what was originally in $A[i]$ into $A[j-1]$. Putting one more loop around gives us straight insertion sort.

$\langle$*straight insertion sort*$\rangle\equiv$

```
template<class E>
void straight_insertion_sort(array<E>& A)
{ int n = A.size();
  for (int i = 1; i < n; i++)
  { E x = A[i];
    int j = i - 1;
    while ( j >= 0 && x < A[j] )
    { A[j+1] = A[j];
      j--;
    }
    A[j+1] = x;
  }
}
```

The running time of straight insertion sort is quadratic. The following lemma gives more detailed information.

Lemma 2 *The number of comparisons of straight insertion sort is $n-1$ in the best case, is $n(n-1)/2$ in the worst case, and is $n^2/4+\Theta(n)$ in the average case.*

Proof: We execute the outer loop for i from 1 to $n-1$.

In each iteration we make at least one comparison, namely the one between $A[i-1]$ and $A[i]$. If the array is already sorted in increasing order, we will make only one comparison. Thus we have $n-1$ comparisons in the best case.

In the worst case, we will compare $A[i]$ will all preceding elements and hence have $i-1$ comparisons. The total number of comparisons is therefore equal to $1+2+\ldots+n-1=n(n-1)/2$ in the worst case.

We come to the average case. The number of comparisons is 1 plus the number of elements smaller than $A[i]$ (the $+1$ is not needed when all elements are preceding elements are smaller than $A[i]$). The expected number of elements smaller than $A[i]$ is simply half the elements preceding it. We conclude that the expected number of comparisons is bounded by $1+i/2$. Summing this bound over all i, we obtain

$$\sum_{1\le i\le n-1} 1+i/2 = n-1+n(n-1)/4\ .$$

This is $n^2/4+\Theta(n)$. ∎

The worst case behavior of selection sort and straight insertion sort is the same: about $n^2/2$ comparisons. A worst case example for both methods is the sequence $\langle n-1, n-2, \ldots, 0\rangle$. On average, straight insertion sort needs only half as many comparisons as selection sort. This is also clearly visible in measured running time, see Figure 1.1.

On an input that is already sorted, straight insertion sort runs in linear time. On an input that is almost sorted, it runs in near linear time. One way to make the notion of near-sortedness precise is to count the number of *inversions*. A inversion is a pair of array elements that are not in their correct order, i.e., a pair of indices i and j such that

$$i<j \quad \text{and} \quad A[i]>A[j]\ .$$

An array that is already sorted in increasing order has no inversions, an array that is sorted in decreasing order has $n(n-1)/2$ inversions, and the number of inversions is always between these bounds.

Lemma 3 *On an array with F inversions the number of comparsions required by stright insertion sort is bounded by $n+F$.*

Proof: This follows from the observation that each comparison (except the first) in the inner loop corresponds to an inversion. ∎

We mention that there are sorting methods with run in $O(n+n\log\max(1,F))$ comparisons, see [**?**].

Straight insertion sort has another nice property. It is *stable*. A sorting method is called stable, if equal elements do not change their relative order.

n	2	4	8	16	32	64
insertion sort	0.04	0.17	0.69	3.01	12.8	51.7
selection sort	0.05	0.19	0.9	3.99	16.7	66.8
insertion sort	0.08	0.32	1.41	5.98	24.6	99.6
selection sort	0.05	0.19	0.88	4	16.6	66.8
insertion sort	0	0	0	0.01	0	0.01
selection sort	0.05	0.21	0.97	4.02	16.6	66.8

Table 1.1: Running times for straight insertion sort and selection sort. All running times are in seconds and for an array of $1000n$ integers. The top part shows the running times for random inputs, the middle part shows the running time on the worst case input $\langle n-1, n-2, \ldots, 0\rangle$, and the bottom part shows the running time on the best case input $\langle 0, 1, \ldots, n-1\rangle$. You may produce you own table by calling insertion_versus_selection_sort_times. The quadratic running times are clearly visible: Doubling n essentially quadruples running time. We can also see that the running time of selection sort basically only depends on the size of the input, that the worst case of insertion sort is about twice its average case, and that the best case of insertion sort is really good.

Summary: Among our simple sorting routines only straight insertion sort is practically useful. It is useful for inputs which are already nearly sorted.

1.2 Merge Sort

We turn to merge sort. It is much more efficient than straight insertion sort and runs in time $O(n \log n)$ on an array of size n. The underlying strategy is also simple. Merge sort operates in phases. At the beginning of the k-th phase, $k \geq 0$, the array is partitioned into sorted blocks of size 2^k. These blocks are paired and any pair is merged into a single sorted block. In the program below we use K to denote 2^k and we use an auxiliary array B with the same index set as A. In even phases the merge step reads from A and writes into B, and in odd phases the roles of A and B are interchanged. In this way the data moves back and forth between A and B. If it ends up in B at the end of *merge_sort*, we need to copy it back to A. We use a boolean variable *even_phase* that is true iff the next phase is even. The actual merging is done by the function *merge*. A call *merge*(X, Y, i, K, h) takes the blocks of X starting at positions i and $i+K$, respectively, and merges them into the block of Y starting at position i and having the combined size of the two blocks to be merged. The last element of the two blocks to be merged is to be found at position h; this information is important if the size of A is not a power of two.

⟨*merge sort*⟩≡

```
  ⟨merge routine⟩
  template<class E>
  void merge_sort(array<E>& A)
  { int n = A.size();
    array<E> B(n);
```

```
    bool even_phase = true;
    for (int K = 1; K < n; K = 2*K)
    { for (int i = 0; i < n; i = i + 2*K)
      { if ( even_phase ) merge(A,B,i,K,n);
        else merge(B,A,i,K,n);
      }
      even_phase = !even_phase;
    }
    if ( !even_phase )
      { for (int i = 0; i < n; i++) A[i] = B[i]; }
  }
```

It remains to define *merge*(X, Y, i, K, n). Our goal is to fill the block of Y starting at position i and extending to position m where $m = \min(i + 2K - 1, h)$ from the two blocks of X starting at positions i and $i + K$, respectively. The two blocks in X extend to $ml = \min(i + K - 1, h)$ and m, respectively. We maintain one index in each of the three blocks to control the merging process: The index j indicates the position in Y that is to be filled next and the indices il and ih point to the smallest remaining elements in the two blocks of X. We always move the smaller of $X[il]$ and $X[ih]$ to $Y[j]$. We break ties in favor of $X[il]$. This makes merge sort a stable sorting method.

⟨*merge routine*⟩≡

```
  #include <LEDA/misc.h>  // to include Min
  template<class E>
  void merge(array<E>& X, array<E>& Y, int i, int K, int n)
  { int il = i; int ih = i + K;
    int ml = leda_min(i + K, n); int m = leda_min(i + 2*K, n);
    for (int j = i; j < m ; j++)
    { if ( ih < m && ( il >= ml || X[ih] < X[il] ) )
           { Y[j] = X[ih]; ih++; }
        else
           { Y[j] = X[il]; il++; }
    }
  }
```

Table 1.2 shows the running times of our two sorting procedures in comparison to the member function *sort* for the task of sorting an array of n *ints*. Observe how the running time of insertion sort explodes. Since its running time grows proportional to n^2, it quadruples whenever n is doubled. In contrast, the running time of the two other methods is $O(n \log n)$ and hence basically doubles whenever n is doubled. The member function *sort* beats our implementation of merge sort because it exploits the fact that the objects to be sorted are *ints*, see Section **??**.

1.3 Old Book

1.3.1 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

n	2	4	8	16	32	64	128	256
insertion sort	0.83	3.27	13.5					
merge sort	0.03	0.06	0.14	0.29	0.66	1.28	2.85	5.99
insertion sort	0.47	1.74	7.07					
merge sort	0.02	0.03	0.09	0.17	0.38	0.78	1.66	3.49
member function	0.01	0.01	0.03	0.07	0.15	0.3	0.6	1.33

Table 1.2: Running times of our sorting routines and the member function *sort*. All running times are in seconds and for an array of $1000n$ integers. Insertion sort and merge sort have been compiled without and with the flag `-DLEDA_CHECKING_OFF`. You may produce you own table by calling array_sort_times.

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 $S[1..n]$.

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 **??** 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 $B \leq A$. Thus the running time of Program **??** $\leq 3 \cdot 1 + 10(n-1) + 5A + 3B = 2.5n^2 + 7.5n - 7 + 3B \leq 4n^2 + 6n - 7$ 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 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

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

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.

1.3.2 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.*

TODO

Figure 1.2: Known size relations

TODO

Figure 1.3: After the second iteration

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 **??**. 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 **??**. 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 **??** after removing the 5 we obtain Figure **??**. 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.

Let us consider another example, the sequence 1 2 3 4 5. We obtain Figure **??** 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 **??**. 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 **??** the three bottom levels of the tree already satisfy the heap property but the root label does not satisfy it yet. We sink the 2 to its proper position by interchanging it with the larger of its children and obtain Figure **??**. 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 **??**

TODO

Figure 1.4: After removal of the 5

TODO

Figure 1.5: New example after the first iteration

TODO

Figure 1.6: Ideal tree for the third example

a tree having the heap property. So the maximum is the root label. We remove it and obtain Figure **??**. 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 **??**. We can now remove the 9 and repeat the process. This gives us Figure **??**. 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 **??**. In this way we always keep the tree as an almost complete binary tree. All 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 1 *An array $S[1 \mathinner{..} n]$ satisfies the* heap property*, if*

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

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

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

Notice that every array $S[1 \mathinner{..} n]$ is a heap starting at $\lfloor n/2 \rfloor + 1$. The tree in Figure **??** corresponds to array 2 10 9 5 8 7 3 6 4 1 and is a heap starting at position 2. Next we give the complete implementation for Heapsort; we assume $n \geq 2$ in Program **??**; 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 $k \geq 1$.

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 comparisons per

TODO

Figure 1.7: Only the root violates the heap property

TODO

Figure 1.8: The root is greater than both children

TODO

Figure 1.9: Whole tree satisfies the heap property

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 $\leq 20n\log(n+1) - n - 7$ 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.

1.3.3 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

TODO

Figure 1.10: After removal of the root

TODO

Figure 1.11: Heap is restored

TODO

Figure 1.12: After removal of the 9

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 $S[0] \leq S[i] \leq S[n+1]$ 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 $S[i] \geq S[1]$ and we decrease k until $S[k] \leq S[1]$. 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 $k \leq i$, we are done. It is easy to see that either $k = i$ (if $S[i] = S[1]$) or $k = i - 1$ (if $S[i] \neq S[1]$). Interchanging $S[k]$ and $S[1]$ completes partitioning. Program **??** 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 $S[l-1] \leq S[i] \leq S[r+1]$ for all i with $l \leq i \leq r$. Existence of these keys is ensured for call *Quicksort*$(1, n)$ by 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 $O(n)$.

TODO

Figure 1.13: Selection phases

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

$$
\begin{aligned}
&QS(0) = QS(1) = 0 \qquad \text{and}\\
&QS(n) \le n + 1 + \max_{1\le k\le n} \{QS(k-1) + QS(n-k)\} \quad \text{for } n \ge 2.
\end{aligned}
$$

It is easy to show by induction that $QS(n) \le (n+1)(n+2)/2-3$ for $n > 0$. This bound is sharp as can be seen from the example $1, 2, \ldots, n$. Here the partitioning phase splits off exactly one element and therefore subproblems of size $n, n-1, n-2, \ldots, 2$ 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 $1/n$, $1 \le k \le n$. 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):

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

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, $\ldots$. Thus there are exactly

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

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

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

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

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

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 $\leq 2(n+1)\ln(n+1)$ 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 $\leq 13(n+1)\ln(n+1) + 29n - 33 \approx 9(n+1)\log(n+1) + 29n - 33$ (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

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 $1 \leq h \leq n$. 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 $QS_{ran}(\Pi) \leq 2(n + 1)\ln(n + 1)$ 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 $n \geq b^{-1}(1)$ 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 Aufrufe und bezeichnen mit $f(n)$ die Kosten des direkten Lösens. $\otimes$Then we obtain the following recurrence for $T(n)$, the running time on a problem of size n. i.D. mehr

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

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

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

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

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

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, $f : [1, \infty] \mapsto I\!R_+$, b strictly increasing and $b(n) < n$ for all n. Then the recurrence*

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

has solution

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

where

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

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

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 5 *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 **??** in terms of recurrence trees. In cases a) and b) of Corollary **??** 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.

1.3.4 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 **??**. 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 **??** this is implemented in our programming language. 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 6 *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 $n > 1$. 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

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

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

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

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 7 *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 $S_1, \dots, S_n$ of length $w_1, \dots, w_n$, 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 **??** represents the merging pattern:

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

TODO

Figure 1.14: A merging pattern

TODO

Figure 1.15: Optimal tree for weights 1 2 4 4 4

Definition 2 *Let T be a binary tree with n leaves $v_1, \dots, v_n$, let $CONT$: leaves of $T \mapsto \{w_1, \dots, w_n\} \subseteq I\!R$ be a bijection and let d_i be the depth of leaf v_i. Then*

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

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 $d \cdot CONT(v)$. 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 3 *Tree T with labelling $CONT$ is* optimal *if*
$Cost(T) \leq Cost(T')$ for any other tree T' and labelling $CONT'$.

Theorem 8 *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 $V = \{v_1, \dots, v_n\}$ of n leaves and labelling $CONT(v_i) = w_i$ 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.

Before we proof correctness and analyze running time we illustrate the algorithm on an example. Let $n = 5$ and $\{w_1, \dots, w_5\} = \{1, 2, 4, 4, 4\}$. 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. **??**.

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 $\{y_1, \dots, y_n\}$ 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 4 *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 5 *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 6 *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 7 *Let $z_1, z_2, \ldots, z_{n-1}$ be the nodes created by the algorithm in order. Then $CONT(z_1) \leq CONT(z_2) \leq \cdots \leq CONT(z_{n-1})$. Furthermore, we have $V = \{v_i, \ldots, v_n\}$, $I = \{z_j, \ldots, z_k\}$ for some $i \leq n+1$, $j \leq k+1 \leq n$ 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 $CONT(z_{k+1}) \geq CONT(z_k)$ 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 $w_1 \leq \cdots \leq w_n$ a queue will do for V and since $CONT(z_1) \leq \cdots \leq CONT(z_{n-1})$ 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.

1.3.5 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 **??** summarizes our knowledge. Here c denotes a constant and F denotes the number of inversions in the input sequence; $0 \leq F \leq n(n-1)/2$ (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 **??**. 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.

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

	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			
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

Table 1.3: Overview table

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 $n \leq 22$. 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.

1.3.6 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

TODO

Figure 1.16: Decision tree for sorting 3 objects

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 $\leq$ 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 $S_3 \leq S_2$ then S_2 is compared with S_1 next, ... We can illustrate the complete sorting algorithm by a tree, a decision tree cf. **??**. 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 $S_i \leq S_j$.

Definition 4 *A* decision tree *is a binary tree whose nodes have labels of the form $S_i : S_j$. The two outgoing edges are labelled $\leq$ 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 $S_i \leq S_j$ or $S_i > S_j$. The leaves of a decision tree represent the different outcomes of the algorithm.

Definition 5 *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 $S(n) \leq k$. A binary tree of depth $\leq k$ has at most 2^k leaves. A decision tree for sorting n elements has at least $n!$ leaves. Thus

$$2^{S(n)} \geq n! \qquad \text{or}$$

$$S(n) \geq \lceil \log n! \rceil,$$

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

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

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 $(1/n!)\sum_\pi l_\pi^T = \sum_i \alpha_i \cdot \mathrm{depth}(b_i)$ 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⊗This implies $H \le P$), that i.D.mehr

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

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

Theorem 9 *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 10 *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 $(S_1, \dots, S_n) \in U^n$ a leaf labelled yes is reached $\iff$ $S_i \neq S_j$ 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 $(S_1, \dots, S_n) \in U^n$ with $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$. Note that this leaf is well-defined because the computation on any input $(S_1, \dots, S_n) \in U^n$ with $S_{\pi(1)} < S_{\pi(2)} < \cdots < S_{\pi(n)}$ will end in the same leaf.
Claim 1 *$v(\pi) \neq v(\rho)$ 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 $(S_1, \dots, S_n)$ such that the computation on input $(S_1, \dots, S_n)$ ends in leaf $v(\pi)$, yet $S_i = S_j$ for some $i \neq j$. This is a contradiction.*

$(S_1, \dots, S_n)$ is constructed as follows. Let $w_1, \dots, w_t$ be the nodes on the path to leaf $v(\pi)$. Consider partial order $P(v(\pi))$ on $(S_1, \dots, S_n)$ 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 $(R_1, \dots, R_n) \in U^n$ satisfies the partial order $P(v(\pi))$ then the computation on input $(R_1, \dots, R_n)$ ends in leaf $v(\pi)$.

2) If $(R_1, \dots, R_n) \in U^n$ is such that $R_{\pi(1)} < \cdots < R_{\pi(n)}$ or $R_{\rho(1)} < \cdots < R_{\rho(n)}$ then $(R_1, \dots, R_n)$ 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 $(S_1, \dots, S_n) \in U^n$ be such that $(S_1, \dots, S_n)$ satisfies partial order $P(v(\pi))$ and $S_a = S_b$. Then the computation on input $(S_1, \dots, S_n)$ 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 $S_1, \dots, S_n$ where $S_{\pi(1)} < S_{\pi(2)} < \dots < S_{\pi(n)}$. 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

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

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 $(1/n!) \sum_\pi l^T_{\pi,s} \ge A(n)$ 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 6 *A* rational decision tree *for inputs $S_1, \dots, S_n$ is a binary tree whose nodes are labelled by expressions $p(S_1, \dots, S_n) : q(S_1, \dots, S_n)$ where p and q are rational functions, whose edges are labelled by $\le$ and $>$, and whose leaves are labelled by rational functions $r(S_1, \dots, S_n)$ of $S_1, \dots, S_n$.*

Rational decision trees compute functions $f : \mathbb{R}^n \mapsto \mathbb{R}$ in an obvious way. Let $(S_1, \dots, S_n) \in \mathbb{R}^n$. Computation starts in the root. Suppose that computation has reached node v which is labelled $p(S_1, \dots, S_n) : q(S_1, \dots, S_n)$. 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 $r(S_1, \dots, S_n)$ then r is evaluated and the value of r is the result of the computation.

Theorem 11 *Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a function and let T be a rational decision tree which computes f. Let $W_1, \dots, W_q$ be pairwise disjoint subsets of $\mathbb{R}$ of non-zero measure* *and let $r_1, \dots, r_q$ 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

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

Then $\{D_v;\ v \text{ leaf of } T\}$ is a partition of $I\!R^n$ 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 $(x_1, \dots, x_n) \in I\!R^n$ define

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

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

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

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

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

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

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

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

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

We will see in Chapter **??** 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 7 *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*

- $Y(v) \leftarrow Y(v_1)\ op\ Y(v_2)$,
- $Y(v) \leftarrow c\ op\ Y(v_2)$ *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 14 *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) $\max(\#V, \#(\mathbb{R}^n - V)) \leq 2^h 3^{h+n}$;

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

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

$\#D(w) \leq 3^{h+n}$. Note first that this claim implies part a) since there are at most 2^h leaves w and each leaf satisfies $\#D(w) \leq 3^{h+n}$. Thus $\#V \leq 2^h 3^{h+n}$ and similarly $\#(\mathbb{R}^n - V) \leq 2^h 3^{h+n}$.

$\#D(w) \leq 3^{h+n}$ 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

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

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

$$-Y(v_1) > 0 \qquad -Y(v_1) \geq 0 \qquad Y(v)Y(v_1) - 1 = 0,$$

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 $\iff$ $Y(v_1) \neq 0$.

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 $r \geq r'$. Clearly $r + s \leq h$, 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

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

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

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

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 $Z \subseteq I\!R^{n+r+s}$ be defined by the system

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

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

Proof: Let $V \subseteq I\!R^n$ 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 $x = (x_1,\ldots,x_n) \in I\!R^n$, $x_i \neq x_j$ 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 $x, x' \in I\!R^n$ 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 **??**): Given n points in the plane, does their convex hull have n vertices?

Theorem 16 *The complexity of the convex hull problem in the model of algebraic computation trees is* $\Omega(n\log n)$.

Proof: Let $V = \{(x_1, y_1,\ldots,x_n,y_n)$; the convex hull of point set $\{(x_i,y_i);\; 1\leq i\leq n\}$ has n vertices$\}$ $\subseteq I\!R^{2n}$.

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 17 *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.

1.4 Heap Sort

1.5 Quicksort

1.6 A Lower Bound on the Number of Comparisons

1.7 Distribution Sort

1.8 Counting Sort

1.9 Sorting Words