17. Sorting and its complexity

Learning objectives:

- What is sorting?
- basic ideas and intrinsic complexity
- insertion sort
- selection sort
- merge sort
- distribution sort
- a lower bound $\Omega(n \cdot \log n)$
- Quicksort
- Sorting in linear time?
- sorting networks

What is sorting? How difficult is it?

The problem

Assume that S is a set of n elements $x_1, x_2, \ldots, x_n$ drawn from a domain X, on which a total order ≤ is defined (i.e. a relation that satisfies the following axioms):

```
≤ is reflexive (i.e ∀∀x ∈ X:  x ≤ x)
≤ is antisymmetric (i.e ∀∀x, y ∈ X:  x ≤ y  ∧  y ≤ x ⇒ x = y)
≤ is transitive (i.e ∀∀x, y, z ∈ X:  x ≤ y  ∧  y ≤ z  ⇒  x ≤ z)
≤ is total (i.e. ∀∀x, y ∈ X  ⇒ x ≤ y  ∨  y ≤ x)
```

Sorting is the process of generating a sequence

$$x_{i_1}, x_{i_2}, \ldots, x_{i_n}$$

such that $(i_1, i_2, \ldots, i_n)$ is a permutation of the integers from 1 to n and

$$\forall k, 1 \le k \le n-1: \quad x_{i_k} \le x_{i_{k+1}}$$

holds. Phrased abstractly, sorting is the problem of finding a specific permutation (or one among a few permutations, when distinct elements may have equal values) out of n! possible permutations of the n given elements. Usually, the set S of elements to be sorted will be given in a data structure; in this case, the elements of S are ordered implicitly by this data structure, but not necessarily according to the desired order ≤. Typical sorting problems assume that S is given in an array or in a sequential file (magnetic tape), and the result is to be generated in the same structure. We characterize elements by their position in the structure (e.g. A[i] in the array A or by the

value of a pointer in a sequential file). The access operations provided by the underlying data structure determine what sorting algorithms are possible.

Algorithms

Most sorting algorithms are refinements of the following idea:

```
while ∃(i, j):  i < j  ∧  A[i] > A[j]  do  A[i] :=: A[j];
```

where :=: denotes the exchange operator. Even sorting algorithms that do not explicitly exchange pairs of elements, or do not use an array as the underlying data structure, can usually be thought of as conforming to the schema above. An insertion sort, for example, takes one element at a time and inserts it in its proper place among those already sorted. To find the correct place of insertion, we can think of a ripple effect whereby the new element successively displaces (exchanges position with) all those larger than itself.

As the schema above shows, two types of operations are needed in order to sort:

- collecting information about the order of the given elements
- ordering the elements (e.g. by exchanging a pair)

When designing an efficient algorithm we seek to economize the number of operations of both types: We try to avoid collecting redundant information, and we hope to move an element as few times as possible. The nondeterministic algorithm given above lets us perform any one of a number of exchanges at a given time, regardless of their usefulness. For example, in sorting the sequence

$x_1 = 5, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 1$

the nondeterministic algorithm permits any of seven exchanges

$x_1 :=: x_i$ for $2 \le i \le 5$ and $x_j :=: x_5$ for $2 \le j \le 4$.

We might have reached the state shown above by following an exotic sorting technique that sorts "from the middle toward both ends", and we might know at this time that the single exchange $x_1 :=: x_5$ will complete the sort. The nondeterministic algorithm gives us no handle to express and use this knowledge.

The attempt to economize work forces us to depart from nondeterminacy and to impose a control structure that carefully sequences the operations to be performed so as to make maximal use of the information gained so far. The resulting algorithms will be more complex and difficult to understand. It is useful to remember, though, that sorting is basically a simple problem with a simple solution and that all the acrobatics in this chapter are due to our quest for efficiency.

Intrinsic complexity

There are obvious limits to how much we can economize. In the absence of any previously acquired information, it is clear that each element must be inspected and, in general, moved at least once. Thus we cannot hope to get away with fewer than $\Omega(n)$ primitive operations. There are less obvious limits, we mention two of them here.

1. If information is collected by asking binary questions only (any question that may receive one of two answers (e.g. a yes/no question, or a comparison of two elements that yields either $\le$ or $>$), then at least $n \cdot \log_2 n$ questions are necessary in general, as will be proved in the section "A lower bound $\Omega n \cdot \log_n$". Thus in this model of computation, sorting requires time $\Theta(n \cdot \log n)$.
2. In addition to collecting information, one must rearrange the elements. In the section "Permutation" in chapter 16, we have shown that in a permutation the average distance of an element from its correct

position is approximately n/3. Therefore elements have to move an average distance of approximately n/3 elements to end up at their destination. Depending on the access operations of the underlying storage structure, an element can be moved to its correct position in a single step of average length n/3, or in n/3 steps of average length 1. If elements are rearranged by exchanging adjacent elements only, then on average $\Theta(n^2)$ moving operations are required. Therefore, short steps are insufficient to obtain an efficient $\Theta(n \cdot \log n)$ sorting algorithm.

Practical aspects of sorting

Records instead of elements. We discuss sorting assuming only that the elements to be sorted are drawn from a totally ordered domain. In practice these elements are just the keys of records that contain additional data associated with the key: for example,

```
type recordtype =  record
     key: keytype;  { totally ordered by ≤ }
     data: anytype
   end;
```

We use the relational operators =, <, ≤ to compare keys, but in a given programming language, say Pascal, these may be undefined on values of type keytype. In general, they must be replaced by procedures: for example, when comparing strings with respect to the lexicographic order.

If the key field is only a small part of a large record, the exchange operation :=:, interpreted literally, becomes an unnecessarily costly copy operation. This can be avoided by leaving the record (or just its data field) in place, and only moving a small surrogate record consisting of a key and a pointer to its associated record.

Sort generators. On many systems, particularly in the world of commercial data processing, you may never need to write a sorting program, even though sorting is a frequently executed operation. Sorting is taken care of by a sort generator, a program akin to a compiler; it selects a suitable sorting algorithm from its repertoire and tailors it to the problem at hand, depending on parameters such as the number of elements to be sorted, the resources available, the key type, or the length of the records.

Partially sorted sequences. The algorithms we discuss ignore any order that may exist in the sequence to be sorted. Many applications call for sorting files that are *almost sorted*, for example, the case where a sorted *master file* is updated with an unsorted *transaction file*. Some algorithms take advantage of any order present in the input data; their time complexity varies from O(n) for almost sorted files to $O(n \cdot \log n)$ for randomly ordered files.

Types of sorting algorithms

Two important classes of incremental sorting algorithms create order by processing each element in turn and placing it in its correct position. These classes, *insertion sorts* and *selection sorts*, are best understood as maintaining two disjoint, mutually exhaustive structures called 'sorted' and 'unsorted'.

```
Initialize: 'sorted' := ∅;  'unsorted' := {x1, x2, … , xn};
Loop:  for i := 1 to n do
     move an element from 'unsorted' to its correct place in
'sorted';
```

The following illustrations show 'sorted' and 'unsorted' sharing an array[1 .. n]. In this case the boundary between 'sorted' and 'unsorted' is represented by an index i that increases as more elements become ordered. The important distinction between the two types of sorting algorithms emerges from the question: In which of the two

structures is most of the work done? Insertion sorts remove the first or most easily accessible element from 'unsorted' and search through 'sorted' to find its proper place. Selection sorts search through 'unsorted' to find the next element to be appended to 'sorted'.

Insertion sort

The i-th step inserts the i-th element into the sorted sequence of the first (i – 1) elements Exhibit 17.1).

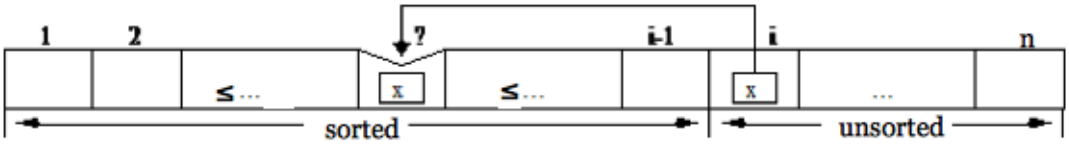

Exhibit 17.1: Insertion sorts move an easily accessed element to its correct place.

Selection sort

The i-th step selects the smallest among the n – i + 1 elements not yet sorted, and moves it to the i-th position (Exhibit 17.2).

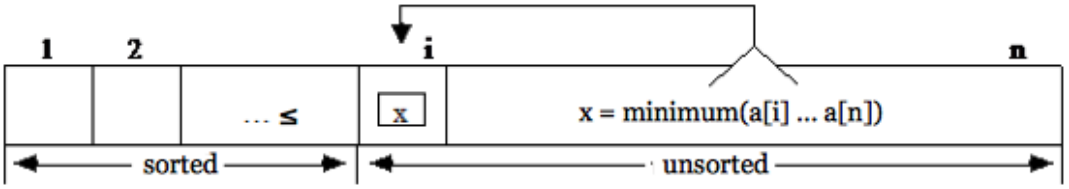

Exhibit 17.2: Selection sorts search for the correct element to move to an easily accessed place.

Insertion and selection sorts repeatedly search through a large part of the entire data to find the proper place of insertion or the proper element to be moved. Efficient search requires random access, hence these sorting techniques are used primarily for *internal sorting* in central memory.

Merge sort

Merge sorts process (sub)sequences of elements in unidirectional order and thus are well suited for *external* sorting on secondary storage media that provide sequential access only, such as magnetic tapes; or random access to large blocks of data, such as disks. Merge sorts are also efficient for internal sorting. The basic idea is to merge two sorted sequences of elements, called *runs*, into one longer sorted sequence. We read each of the input runs, and write the output run, starting with small elements and ending with the large ones. We keep comparing the smallest of the remaining elements on each input run, and append the smaller of the two to the output run, until both input runs are exhausted (Exhibit 17.3).

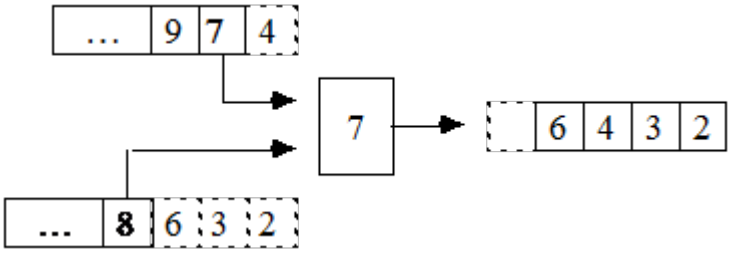

Exhibit 17.3: Merge sorts exploit order already present.

The processor shown at left in Exhibit 17.4 reads two tapes, A and B. Tape A contains runs 1 and 2; tape B contains runs 3 and 4. The processor merges runs 1 and 3 into the single run 1 & 3 on tape C, and runs 2 and 4 into the single run 2 & 4 on tape D. In a second merge step, the processor shown at the right reads tapes C and D and merges the two runs 1 & 3 and 2 & 4 into one run, 1 & 3 & 2 & 4.

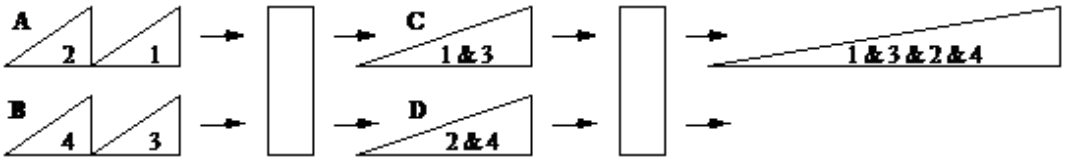

Exhibit 17.4: Two merge steps in sequence.

Distribution sort

Distribution sorts process the *representation* of an element as a value in a radix number system and use primitive arithmetic operations such as "extract the k-th digit". These sorts do not compare elements directly. They introduce a different model of computation than the sorts based on comparisons, exchanges, insertions, and deletions that we have considered thus far. As an example, consider numbers with at most three digits in radix 4 representation. In a first step these numbers are distributed among four queues according to their least significant digit, and the queues are concatenated in increasing order. The process is repeated for the middle digit, and finally for the leftmost, most significant digit, as shown in Exhibit 17.5

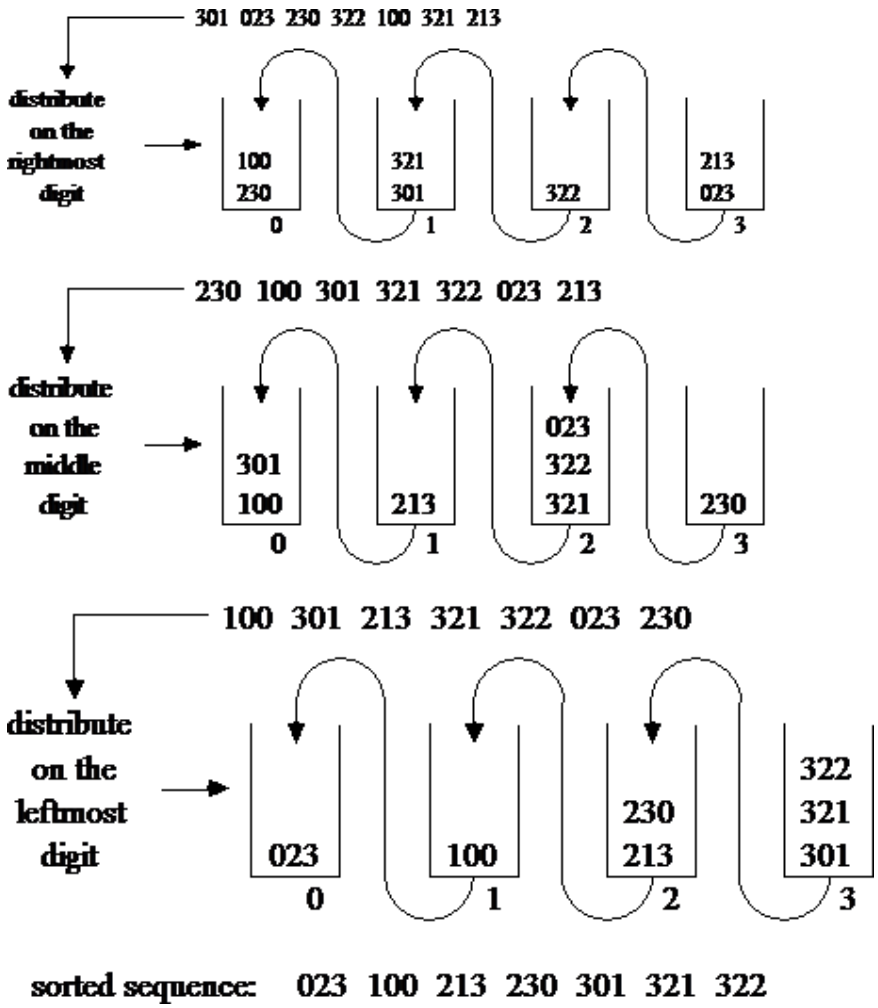

Exhibit 17.5 Distribution sorts use the radix representation of keys to organize elements in buckets

We have now seen the basic ideas on which all sorting algorithms are built. It is more important to understand these ideas than to know dozens of algorithms based on them. To appreciate the intricacy of sorting, you must understand some algorithms in detail: we begin with simple ones that turn out to be inefficient.

Simple sorting algorithms that work in time $\Theta(n^2)$

If you invent your own sorting technique without prior study of the literature, you will probably "discover" a well-known inefficient algorithm that works in time $O(n^2)$, requires time $\Theta(n^2)$ in the worst case, and thus is of time complexity $\Omega(n^2)$. Your algorithm might be similar to one described below.

Consider *in-place algorithms* that work on an array declared as

```
var  A: array[1 .. n] of elt;
```

and place the elements in ascending order. Assume that the comparison operators are defined on values of type elt. Let c_{best}, $c_{average}$, and c_{worst} denote the number of comparisons, and e_{best}, $e_{average}$, and e_{worst} the number of exchange operations performed in the best, average, and worst case, respectively. Let $inv_{average}$ denote the average number of inversions in a permutation.

Insertion sort (Exhibit 17.6)

Let $-\infty$ denote a constant $\leq$ any key value. The smallest value in the domain often serves as a sentinel $-\infty$.

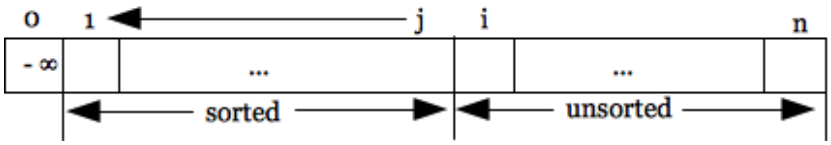

Exhibit 17.6: Straight insertion propagates a ripple-effect across the sorted part of the array.

```
   A[0] := -∞;
   for i := 2 to n do  begin
      j := i;
      while  A[j] < A[j - 1]  do  { A[j] :=: A[j - 1]; { exchange }
j := j - 1 }
   end;
```

$$c_{best} = n - 1$$

$$c_{average} = inv_{average} + (n - 1) = \frac{n^2 + 3 \cdot n - 4}{4}$$

$$c_{worst} = \sum_{i=2}^{n} i = \frac{n^2 + n - 2}{2}$$

$$e_{best} = 0$$

$$e_{average} = inv_{average} = \frac{n^2 - n}{4}$$

$$e_{worst} = \sum_{i=2}^{n} (i - 1) = \frac{n^2 - n}{2}$$

This straight insertion sort is an Θ(n) algorithm in the best case and an Θ(n²) algorithm in the average and worst cases. In the program above, the point of insertion is found by a linear search interleaved with exchanges. A binary search is possible but does not improve the time complexity in the average and worst cases, since the actual insertion still requires a linear-time ripple of exchanges.

Selection sort (Exhibit 17.7)

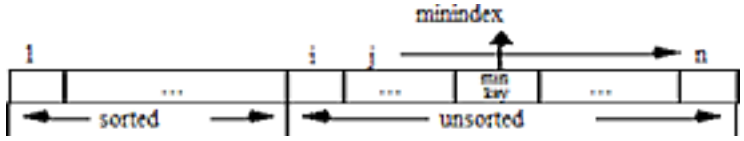

Exhibit 17.7: Straight selection scans the unsorted part of the array.

```
for i := 1 to n - 1 do  begin
   minindex := i;  minkey := A[i];
   for j := i + 1 to n do
      if  A[j] < minkey  then  { minkey := A[j];  minindex := j }
   A[i] :=: A[minindex]  { exchange }
end;
```

$$c_{best} = c_{average} = c_{worst} = \sum_{i=1}^{n-1} (n-i) = \frac{n^2 - n}{2}$$

$$e_{best} = e_{average} = e_{worst} = n - 1$$

The sum in the formula for the number of comparisons reflects the structure of the two nested for loops. The body of the inner loop is executed the same number of times for each of the three cases. Thus this straight selection sort is of time complexity $\Theta(n^2)$.

A lower bound $\Omega(n \cdot \log n)$

A straightforward counting argument yields a lower bound on the time complexity of any sorting algorithm that collects information about the ordering of the elements by asking only binary questions. A binary question has a two-valued answer: yes or no, true or false. A comparison of two elements, $x \le y$, is the most obvious example, but the following theorem holds for binary questions in general.

Theorem: Any sorting algorithm that collects information by asking binary questions only executes at least

$$n \cdot \log_2 (n+1) - \frac{n}{\ln 2}$$

binary questions both in the worst case, and averaged over all n! permutations. Thus the average and worst-case time complexity of such an algorithm is $\Omega(n \cdot \log n)$.

Proof: A sorting algorithm of the type considered here can be represented by a *binary decision tree*. Each internal node in such a tree represents a binary question, and each leaf corresponds to a result of the decision process. The decision tree must distinguish each of the n! possible permutations of the input data from all the others; and thus must have at least n! leaves, one for each permutation.

Example: The decision tree shown in Exhibit 17.8 collects the information necessary to sort three elements, x, y and z, by comparisons between two elements.

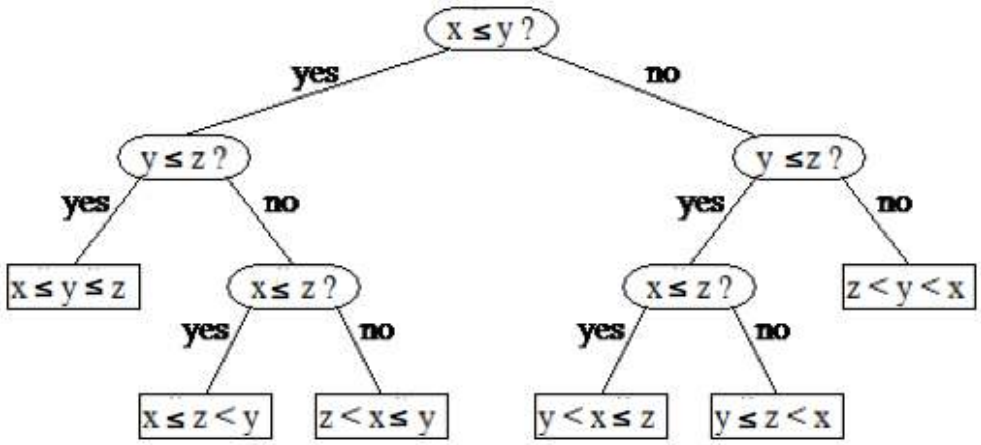

Exhibit 17.8 The decision tree shows the possible n! Outcomes when sorting n elements.

The average number of binary questions needed by a sorting algorithm is equal to the average depth of the leaves of this decision tree. The lemma following this theorem will show that in a binary tree with k leaves the average depth of the leaves is at least $\log_2 k$. Therefore, the average depth of the leaves corresponding to the n! permutations is at least $\log_2 n!$. Since

$$\log_2 n! = \log_2\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log_2 i$$

$$\geq (n+1)\cdot\log_2(n+1) - \frac{n}{\ln 2} - \log_2(n+1) = n\cdot\log_2(n+1) - \frac{n}{\ln 2}$$

it follows that on average at least

$$n*\log_2(n+1) - \frac{n}{\ln_2}$$

binary questions are needed, that is, the time complexity of each such sorting algorithm is $\Omega(n \cdot \log n)$ in the average, and therefore also in the worst case.

Lemma: In a binary tree with k leaves the average depth of the leaves is at least $\log_2 k$.

Proof: Suppose that the lemma is not true, and let T be the counterexample with the smallest number of nodes. T cannot consist of a single node because the lemma is true for such a tree. If the root r of T has only one child, the subtree T' rooted at this child would contain the k leaves of T that have an even smaller average depth in T' than in T. Since T was the counterexample with the smallest number of nodes, such a T' cannot exist. Therefore, the root r of T must have two children, and there must be $k_L > 0$ leaves in the left subtree and $k_R > 0$ leaves in the right subtree of r ($k_L + k_R = k$). Since T was chosen minimal, the k_L leaves in the left subtree must have an average depth of at least $\log_2 k_L$, and the k_R leaves in the right subtree must have an average depth of at least $\log_2 k_R$. Therefore, the average depth of all k leaves in T must be at least

$$\frac{k_L}{k_L + k_R}\cdot\log_2 k_L + \frac{k_R}{k_L + k_R}\cdot\log_2 k_R + 1. \qquad (*)$$

It is easy to see that (*) assumes its minimum value if $k_L = k_R$. Since (*) has the value $\log_2 k$ if $k_L = k_R = k/2$ we have found a contradiction to our assumption that the lemma is false.

Quicksort

Quicksort (C. A. R. Hoare, 1962) [Hoa 62] combines the powerful algorithmic principle of divide-and-conquer with an efficient way of moving elements using few exchanges. The *divide phase* partitions the array into two disjoint parts: the "small" elements on the left and the "large" elements on the right. The *conquer phase* sorts each part separately. Thanks to the work of the divide phase, the *merge phase* requires no work at all to combine two partial solutions. Quicksort's efficiency depends crucially on the expectation that the divide phase cuts two sizable subarrays rather than merely slicing off an element at either end of the array (Exhibit 17.9).

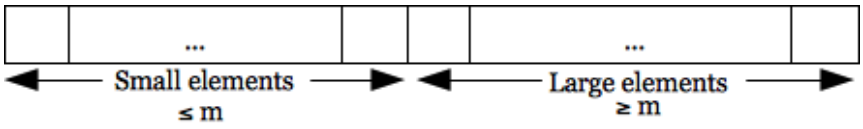

Exhibit 17.9: Quicksort partitions the array into the "small" elements on the left and the "large" elements on the right.

We chose an arbitrary *threshold* value m to define "small" as ≤ m, and "large" as ≥ m, thus ensuring that any "small element" ≤ any "large element". We partition an arbitrary subarray A[L .. R] to be sorted by executing a left-to-right scan (incrementing an index i) "concurrently" with a right-to-left scan (decrementing j) (Exhibit 17.10). The left-to-right scan pauses at the first element A[i] ≥ m, and the right-to-left scan pauses at the first element A[j] ≤ m. When both scans have paused, we exchange A[i] and A[j] and resume the scans. The partition is complete when the two scans have crossed over with j < i. Thereafter, quicksort is called recursively for A[L .. j] and A[i .. R], unless one or both of these subarrays consists of a single element and thus is trivially sorted. *Example* of partitioning (m = 16):

```
25  23   3  16   4    7   29   6
i                             j
6   23   3  16   4    7   29   25
    i                 j
6   7    3  16   4    23  29   25
           i    j
6   7    3  4    16   23  29   25
           j    i
```

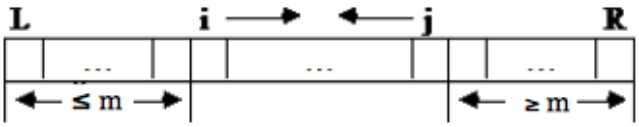

Exhibit 17.10: Scanning the array concurrently from left to right and from right to left.

Although the threshold value m appeared arbitrary in the description above, it must meet criteria of correctness and efficiency. *Correctness:* if either the set of elements ≤ m or the set of elements ≥ m is empty, quicksort fails to terminate. Thus we require that $\min(x_i) \le m \le \max(x_i)$. *Efficiency* requires that m be close to the median.

How do we find the median of n elements? The obvious answer is to sort the elements and pick the middle one, but this leads to a chicken-and-egg problem when trying to sort in the first place. There exist sophisticated algorithms that determine the exact median of n elements in time O(n) in the worst case [BFPRT 72]. The multiplicative constant might be large, but from a theoretical point of view this does not matter. The elements are partitioned into two equal-sized halves, and quicksort runs in time $O(n \cdot \log n)$ even in the worst case. From a practical point of view, however, it is not worthwhile to spend much effort in finding the exact median when there are much cheaper ways of finding an acceptable approximation. The following techniques have all been used to pick a threshold m as a "guess at the median":

- An array element in a fixed position such as A[(L + R) div 2]. *Warning:* stay away from either end, A[L] or A[R], as these thresholds lead to poor performance if the elements are partially sorted.
- An array element in a random position: a simple technique that yields good results.
- The median of three or five array elements in fixed or random positions.

- The average between the smallest and largest element. This requires a separate scan of the entire array in the beginning; thereafter, the average for each subarray can be calculated during the previous partitioning process.

The recursive procedure 'rqs' is a possible implementation of quicksort. The function 'guessmedian' must yield a threshold that lies on or between the smallest and largest of the elements to be sorted. If an array element is used as the threshold, the procedure 'rqs' should be changed in such a way that after finishing the partitioning process this element is in its final position between the left and right parts of the array.

```
procedure rqs (L, R: 1 .. n);  { sorts A[L], … , A[R] }
var  i, j: 0 .. n + 1;

  procedure partition;
  var  m: elt;
  begin  { partition }
    m := guessmedian (L, R);
    { min(A[L], … , A[R]) ≤ m ≤ max(A[L], … , A[R]) }
    i := L;  j := R;
    repeat
      { A[L], … , A[i - 1] ≤ m ≤ A[j + 1], … , A[R] }
      while  A[i] < m  do  i := i + 1;
      { A[L], … , A[i - 1] ≤ m ≤ A[i] }
      while  m < A[j]  do  j := j - 1;
      { A[j] ≤ m ≤ A[j + 1], … , A[R] }
      if  i ≤ j  then  begin
        A[i] :=: A[j];  { exchange }
        { i ≤ j ⇒ A[i] ≤ m ≤ A[j] }
        i := i + 1;  j := j - 1
        { A[L], … , A[i - 1] ≤ m ≤ A[j + 1], … , A[R] }
      end
      else
        { i > j ⇒ i = j + 1 ⇒ exit }
      end
    until  i > j
  end;  { partition }

begin  { rqs }
  partition;
  if L < j then  rqs(L, j);
  if i < R then  rqs(i, R)
end;  { rqs }
```

An initial call 'rqs(1, n)' with n > 1 guarantees that L < R holds for each recursive call.

An iterative implementation of quicksort is given by the following procedure, 'iqs', which sorts the whole array A[1 .. n]. The boundaries of the subarrays to be sorted are maintained on a stack.

```
procedure iqs;
const  stacklength = … ;
type  stackelement = record  L, R: 1 .. n  end;
var  i, j, L, R, s: 0 .. n;
  stack: array[1 .. stacklength] of stackelement;

  procedure partition;  { same as in rqs }
  end;  { partition }

begin  { iqs }
  s := 1;  stack[1].L := 1;  stack[1].R := n;
  repeat
    L := stack[s].L;  R := stack[s].R;  s := s - 1;
```

```
        repeat
          partition;
          if  j - L < R - i  then  begin
            if  i <  R then  { s := s + 1;  stack[s].L := i;
stack[s].R := R };
            R := j
          end
          else  begin
            if  L < j  then  { s := s + 1;  stack[s].L := L;
stack[s].R := j };
            L := i
          end
        until  L ≥ R
      until  s = 0
    end;  { iqs }
```

After partitioning, 'iqs' pushes the bounds of the larger part onto the stack, thus making sure that part will be sorted later, and sorts the smaller part first. Thus the length of the stack is bounded by $\log_2 n$.

For very small arrays, the overhead of managing a stack makes quicksort less efficient than simpler $O(n^2)$ algorithms, such as an insertion sort. A practically efficient implementation of quicksort might switch to another sorting technique for subarrays of size up to 10 or 20. [Sed 78] is a comprehensive discussion of how to optimize quicksort.

Analysis for three cases: best, "typical", and worst

Consider a quicksort algorithm that chooses a guessed median that differs from any of the elements to be sorted and thus partitions the array into two parts, one with k elements, the other with n – k elements. The work q(n) required to sort n elements satisfies the recurrence relation

$$q(n) = q(k) + q(n-k) + a \cdot n + b. \qquad (*)$$

The constant b measures the cost of calling quicksort for the array to be sorted. The term a · n covers the cost of partitioning, and the terms q(k) and q(n – k) correspond to the work involved in quicksorting the two subarrays. Most quicksort algorithms partition the array into three parts: the "small" left part, the single array element used to guess the median, and the "large" right part. Their work is expressed by the equation

$$q(n) = q(k) + q(n-k-1) + a \cdot n + b.$$

We analyze equation (*); it is close enough to the second equation to have the same asymptotic solution. Quicksort's behavior in the best and worst cases are easy to analyze, but the average over all permutations is not. Therefore, we analyze another average which we call the *typical case*.

Quicksort's *best-case behavior* is obtained if we guess the correct median that partitions the array into two equal-sized subarrays. For simplicity's sake the following calculation assumes that n is a power of 2, but this assumption does not affect the solution. Then (*) can be rewritten as

$$q(n) = 2 \cdot q\left(\frac{n}{2}\right) + a \cdot n + b.$$

We use this recurrence equation to calculate

$$q(\frac{n}{2}) = 2 \cdot q(\frac{n}{4}) + a \cdot \frac{n}{2} + b$$

and substitute on the right-hand side to obtain

$$q(n) = 2 \cdot \left(2 \cdot q(\frac{n}{4}) + a \cdot \frac{n}{2} + b\right) + a \cdot n + b = 4 \cdot q(\frac{n}{4}) + 2 \cdot a \cdot n + 3 \cdot b.$$

Repeated substitution yields

$$\begin{aligned} q(n) &= n \cdot q(1) + a \cdot n \cdot \log_2 n + b \cdot (n-1) \\ &= a \cdot n \cdot \log_2 n + g(n) \quad \text{with} \quad g(n) \in O(n). \end{aligned}$$

The constant q(1), which measures quicksort's work on a trivially sorted array of length 1, and b, the cost of a single procedure call, do not affect the dominant term $n \cdot \log_2 n$. The constant factor a in the dominant term can be estimated by analyzing the code of the procedure 'partition'. When these details do not matter, we summarize: Quicksort's time complexity in the best case is $\Theta(n \cdot \log n)$.

Quicksort's *worst-case* behavior occurs when one of the two subarrays consists of a single element after each partitioning. In this case equation ( ) becomes

$$q(n) = q(n-1) + q(1) + a \cdot n + b.$$

We use this recurrence equation to calculate

$$q(n-1) = q(n-2) + q(1) + a \cdot (n-1) + b$$

and substitute on the right-hand side to obtain

$$q(n) = q(n-2) + 2 \cdot q(1) + a \cdot n + a \cdot (n-1) + 2 \cdot b.$$

Repeated substitution yields

$$q(n) = n \cdot q(1) + a \cdot \frac{n^2 + n - 2}{2} + b \cdot (n-1).$$

Therefore the time complexity of quicksort in the worst case is $\Theta(n^2)$.

For the analysis of quicksort's *typical behavior* we make the plausible assumption that the array is equally likely to get partitioned between any two of its elements: For all k, $1 \le k < n$, the probability that the array A is partitioned into the subarrays A[1 .. k] and A[k + 1 .. n] is $1 / (n - 1)$. Then the average work to be performed by quicksort is expressed by the recurrence relation

$$q(n) = \frac{1}{n-1}\sum_{k=1}^{n-1}(q(k)+q(n-k)) + a\cdot n + b$$
$$= \frac{2}{n-1}\sum_{k=1}^{n-1} q(k) + a\cdot n + b.$$

This recurrence relation approximates the recurrence relation discussed in chapter 16 well enough to have the same solution

$$q(n) = (\ln 4)\cdot a\cdot n\cdot \log_2 n + g(n) \quad \text{with} \quad g(n)\in O(n).$$

Since ln 4 ≈ 1.386, quicksort's asymptotic behavior in the typical case is only about 40% worse than in the best case, and remains in Θ(n · log n). [Sed 77] is a thorough analysis of quicksort.

Merging and merge sorts

The *internal* sorting algorithms presented so far require *direct access* to each element. This is reflected in our analyses by treating an array access A[i], or each exchange A[i] :=: A[j], as a primitive operation whose cost is constant (independent of n). This assumption is not valid for elements stored on secondary storage devices such as magnetic tapes or disks. A better assumption that mirrors the realities of *external* sorting is that the elements to be sorted are stored as a *sequential file* f. The file is accessed through a file pointer which, at any given time, provides direct access to a single element. Accessing other elements requires repositioning of the file pointer. Sequential files may permit the pointer to advance in one direction only, as in the case of Pascal files, or to move backward and forward. In either case, our theoretical model assumes that the time required for repositioning the pointer is proportional to the distance traveled. This assumption obviously favors algorithms that process (compare, exchange) pairs of adjacent elements, and penalizes algorithms such as quicksort that access elements in random positions.

The following external sorting algorithm is based on the merge sort principle. To make optimal use of the available main memory, the algorithm first creates initial runs; a *run* is a sorted subsequence of elements $f_i, f_{i+1}, \ldots, f_j$ stored consecutively in file f, $f_k \le f_{k+1}$ for all k with $i \le k \le j - 1$. Assume that a buffer of capacity m elements is available in main memory to create *initial runs* of length m (perhaps less for the last run). In processing the r-th run, r = 0, 1, … , we read the m elements $f_{r\cdot m+1}, f_{r\cdot m+2}, \ldots, f_{r\cdot m+m}$ into memory, sort them internally, and write the sorted sequence to a modified file f, which may or may not reside in the same physical storage area as the original file f. This new file f is partially sorted into runs: $f_k \le f_{k+1}$ for all k with $r \cdot m + 1 \le k < r \cdot m + m$.

At this point we need two files, g and h, in addition to the file f, which contains the initial runs. In a *copy phase* we distribute the initial runs by copying half of them to g, the other half to h. In the subsequent *merge phase* each run of g is merged with exactly one run of h, and the resulting new run of double length is written onto f (Exhibit 17.11). After the first cycle, consisting of a copy phase followed by a merge phase, f contains half as many runs as it did before. After $\lceil \log_2(n/m) \rceil$ cycles f contains one single run, which is the sorted sequence of all elements.

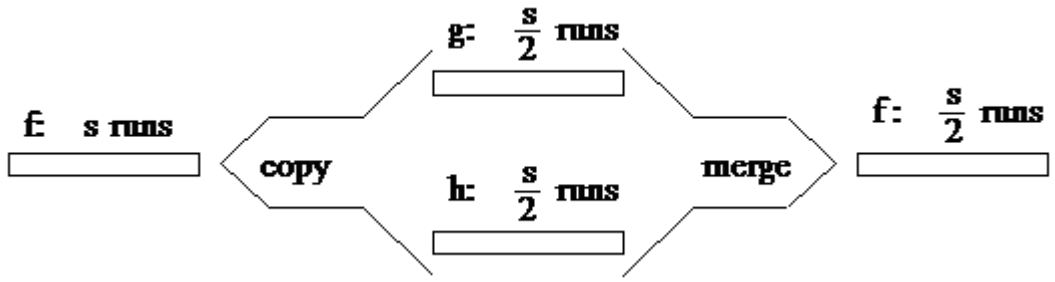

Exhibit 17.11: Each copy-merge cycle halves the number of runs and doubles their lengths.

Exercise: a merge sort in main memory

Consider the following procedure that sorts the array A:

```
const  n = … ;
var  A: array[1 .. n] of integer;
…

procedure sort (L, R: 1 .. n);
  var  m: 1 .. n;

  procedure combine;
  var  B: array [1 .. n] of integer;
         i, j, k: 1 .. n;
  begin  { combine }
    i := L;  j := m + 1;
    for k := L to R do
      if  (i > m) cor ((j ≤ R) cand (A[j] < A[i]))  then
        { B[k] := A[j];  j := j + 1 }
      else
        { B[k] := A[i];  i := i + 1 } ;
    for k := L to R do  A[k] := B[k]
  end;  { combine }

begin  { sort}
  if  L < R  then
    { m := (L + R) div 2;  sort(L, m);  sort(m + 1, R);  combine }
end;  { sort }
```

The relational operators 'cand' and 'cor' are conditional! The procedure is initially called by

```
sort(1,n);
```

(a) Draw a picture to show how 'sort' works on an array of eight elements.

(b) Write down a recurrence relation to describe the work done in sorting n elements.

(c) Determine the asymptotic time complexity by solving this recurrence relation.

(d) Assume that 'sort' is called for m subarrays of equal size, not just for two. How does the asymptotic time complexity change?

Solution

(a) 'sort' depends on the algorithmic principle of divide and conquer. After dividing an array into a left and a right subarray whose numbers of elements differ by at most one, 'sort' calls itself recursively on these two subarrays. After these two calls are finished, the procedure 'combine' merges the two sorted subarrays A[L .. m] and A[m + 1 .. R] together in B. Finally, B is copied to A. An example is shown in Exhibit 17.12.

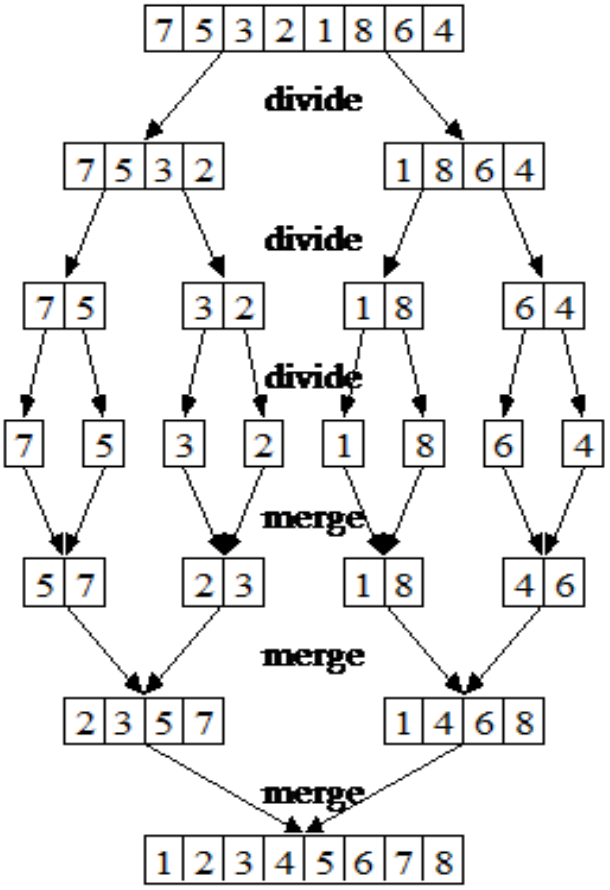

Exhibit 17.12: Sorting an array by using a divide-and-conquer scheme.

(b) The work w(n) performed while sorting n elements satisfies

$$w(n) = 2 \cdot w\left(\frac{n}{2}\right) + a \cdot n + b. \qquad (*)$$

The first term describes the cost of the two recursive calls of 'sort', the term $a \cdot n$ is the cost of merging the two sorted subarrays, and the constant b is the cost of calling 'sort' for the array.

(c) If

$$w\left(\frac{n}{2}\right) = 2 \cdot w\left(\frac{n}{4}\right) + a \cdot \frac{n}{2} + b$$

is substituted in (*), we obtain

$$w(n) = 4 \cdot w\left(\frac{n}{4}\right) + 2 \cdot a \cdot n + 3 \cdot b.$$

Continuing this substitution process results in

$$\begin{aligned} w(n) &= n \cdot w(1) + a \cdot n \cdot \log_2 n + b \cdot (n-1) \\ &= a \cdot n \cdot \log_2 n + g(n) \quad \text{with} \quad g(n) \in O(n). \end{aligned}$$

since w(1) is constant the time complexity of 'sort' is Θ(n · log n).

(d) If 'sort' is called recursively for m subarrays of equal size, the cost w'(n) is

$$w'(n) = m \cdot w'\left(\frac{n}{m}\right) + a \cdot m \cdot n + b.$$

solving this recursive equation shows that the time complexity does not change [i.e. it is Θ(n · log n)].

Is it possible to sort in linear time?

The lower bound Ω(n · log n) has been derived for sorting algorithms that gather information about the ordering of the elements by binary questions and nothing else. This lower bound need not apply in other situations.

Example 1: sorting a permutation of the integers from 1 to n

If we know that the elements to be sorted are a permutation of the integers 1 .. n, it is possible to sort in time Θ(n) by storing element i in the array element with index i.

Example 2: sorting elements from a finite domain

Assume that the elements to be sorted are samples from a finite domain W = 1 .. w. Then it is possible to sort in time Θ(n) if gaps between the elements are allowed (Exhibit 17.13). The gaps can be closed in time Θ(w).

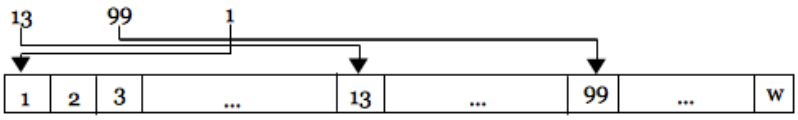

Exhibit 17.13: Sorting elements from a finite domain in linear time.

Do these examples contradict the lower bound Ω(n · log n)? No, because in these examples the information about the ordering of elements is obtained by asking questions more powerful than binary questions: namely, n-valued questions in Example 1 and w-valued questions in Example 2.

A k-valued question is equivalent to $\log_2 k$ binary questions. When this "exchange rate" is taken into consideration, the theoretical time complexities of the two sorting techniques above are Θ(n · log n) and Θ(n · log w), respectively, thus conforming to the lower bound in the section "A lower bound Ω(n · log n)".

Sorting algorithms that sort in linear time (expected linear time, but not in the worst case) are described in the literature under the terms *bucket sort*, distribution sort, and *radix sort*.

Sorting networks

The sorting algorithms above are designed to run on a sequential machine in which all operations, such as comparisons and exchanges, are performed one at a time with a single processor. If algorithms are to be efficient, they need to be rethought when the ground rules for their execution change: when the theoretician uses another model of computation, or when they are executed on a computer with a different architecture. This is particularly true of the many different types of multiprocessor architectures that have been built or conceived. When many processors are available to share the workload, questions of how to distribute the work among them, how to synchronize their operation, and how to transport data, prevail. It is not our intention to discuss sorting on general-purpose parallel machines. We wish to illustrate the point that algorithms must be redesigned when the model of

computation changes. For this purpose a discussion of special-purpose sorting networks suffices. The "processors" in a sorting network are merely *comparator*s: Their only function is to compare the values on two input wires and switch them onto two output wires such that the smaller is on top, the larger at the bottom (Exhibit 17.14).

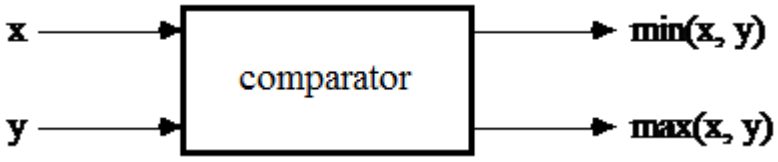

Exhibit 17.14: Building block of sorting networks.

Comparators are arranged into a network in which n wires enter at the left and n wires exit at the right, as Exhibit 17.15 shows, where each vertical connection joining a pair of wires represents a comparator. The illustration also shows what happens to four input elements, chosen to be 4, 1, 3, 2 in this example, as they travel from left to right through the network.

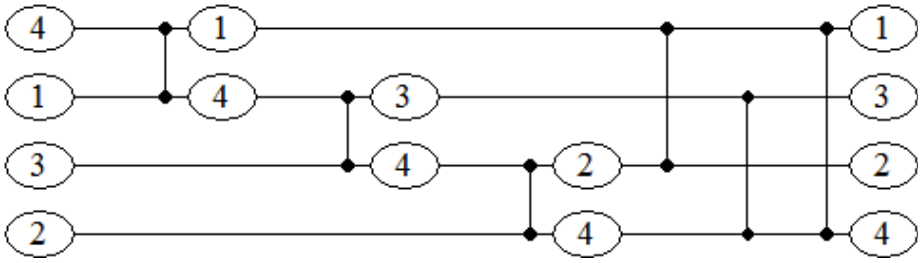

Exhibit 17.15: A comparator network that fails to sort. The output of each comparator performing an exchange is shown in the ovals.

A network of comparators is a *sorting network* if it sorts every input configuration. We consider an input configuration to consist of distinct elements, so that without loss of generality we may regard it as one of the n! permutations of the sequence (1, 2, … , n). A network that sorts a duplicate-free configuration will also sort a configuration containing duplicates.

The comparator network above correctly sorts many of its 4! = 24 input configurations, but it fails on the sequence (4, 1, 3, 2). Hence it is not a sorting network. It is evident that a network with a sufficient number of comparators in the right places will sort correctly, but as the example above shows, it is not immediately evident what number suffices or how the comparators should be placed. The network in Exhibit 17.16 shows that five comparators, arranged judiciously, suffice to sort four elements.

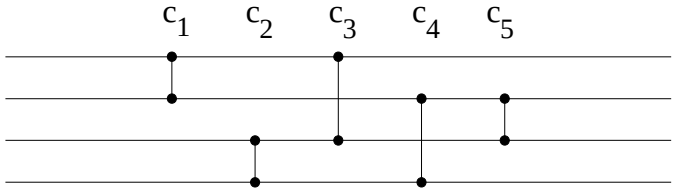

Exhibit 17.16: Five comparators suffice to sort four elements.

How can we tell if a given network sorts successfully? Exhaustive testing is feasible for small networks such as the one above, where we can trace the flow of all 4! = 24 input configurations. Networks with a regular structure

usually admit a simpler correctness proof. For this example, we observe that c_1, c_2, and c_3 place the smallest element on the top wire. Similarly, c_1, c_2, and c_4 place the largest on the bottom wire. This leaves the middle two elements on the middle two wires, which c_5 then puts into place.

What design principles might lead us to create large sorting networks guaranteed to be correct? Sorting algorithms designed for a sequential machine cannot, in general, be mapped directly into network notation, because the network is a more restricted model of computation: Whereas most sequential sorting algorithms make comparisons based on the outcome of previous comparisons, a sorting network makes the same comparisons for all input configurations. The same fundamental algorithm design principles useful when designing sequential algorithms also apply to parallel algorithms.

Divide-and-conquer. Place two sorting networks for n wires next to each other, and combine them into a sorting network for $2 \cdot n$ wires by appending a *merge network* to merge their outputs. In sequential computation merging is simple because we can choose the most useful comparison depending on the outcome of previous comparisons. The rigid structure of comparator networks makes merging networks harder to design.

Incremental algorithm.We place an n-th wire next to a sorting network with $n - 1$ wires, and either precede or follow the network by a "ladder" of comparators that tie the extra wire into the existing network, as shown in the following figures. This leads to designs that mirror the straight insertion and selection algorithms in the section "Simple sorting algorithms that work in time $\Theta(n^2)$

Insertion sort. With the top $n - 1$ elements sorted, the element on the bottom wire trickles into its correct place. Induction yields the expanded diagram on the right in Exhibit 17.17.

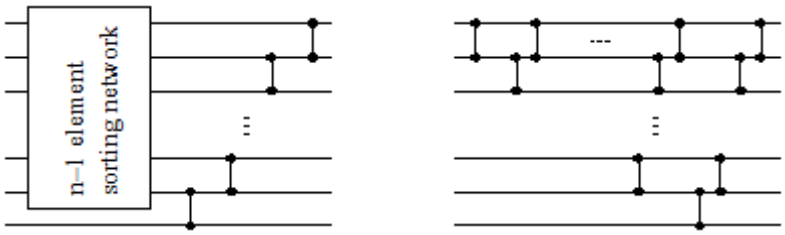

Exhibit 17.17: Insertion sort leads by induction to the sorting network on the right.

Selection sort. The maximum element first trickles down to the bottom, then the remaining elements are sorted. The expanded diagram is on the right in Exhibit 17.18.

Exhibit 17.18: Selection sort leads by induction to the sorting network on the right.

Comparators can be shifted along their pair of wires so as to reduce the number of stages, provided that the topology of the network remains unchanged. This compression reduces both insertion and selection sort to the triangular network shown in Exhibit 17.19. Thus we see that the distinction between insertion and selection was more a distinction of sequential order of operations rather than one of data flow.

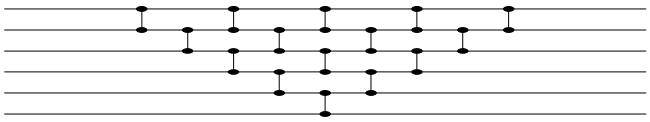

Exhibit 17.19: Shifting comparators reduces the number of stages.

Any number of comparators that are aligned vertically require only a single unit of time. The compressed triangular network has $O(n^2)$ comparators, but its time complexity is $2 \cdot n - 1 \in O(n)$. There are networks with better asymptotic behavior, but they are rather exotic [Knu 73b].

Exercises and programming projects

1. Implement insertion sort, selection sort, merge sort, and quicksort and animate the sorting process for each of these algorithms: for example, as shown in the snapshots in "Algorithm animation". Compare the number of comparisons and exchange operations needed by the algorithms for different input configurations.
2. What is the smallest possible depth of a leaf in a decision tree for a sorting algorithm?
3. Show that $2 \cdot n - 1$ comparisons are necessary in the worst case to merge two sorted arrays containing n elements each.
4. The most obvious method of systematically interchanging the out-of-order pairs of elements in an array

   ```
   var  A: array[1 .. n] of elt;
   ```

   is to scan adjacent pairs of elements from bottom to top (imagine that the array is drawn vertically, with A[1] at the top and A[n] at the bottom) repeatedly, interchanging those found out of order:

   ```
   for  i := 1  to  n - 1  do
         for  j := n  downto  i + 1  do
           if  A[j - 1] > A[j]  then  A[j - 1] :=: A[j];
   ```

   This technique is known as *bubble sort,* since smaller elements "bubble up" to the top.

   (a) Explain by words, figures, and an example how bubble sort works. Show that this algorithm sorts correctly.

   (b) Determine the exact number of comparisons and exchange operations that are performed by bubble sort in the best, average, and worst case.

   (c) What is the worst-case time complexity of this algorithm?
5. A sorting algorithm is called *stable* if it preserves the original order of equal elements. Which of the sorting algorithms discussed in this chapter is stable?
6. Assume that quicksort chooses the threshold m as the first element of the sequence to be sorted. Show that the running time of such a quicksort algorithm is $\Theta(n^2)$ when the input array is sorted in nonincreasing or nondecreasing order.
7. Find a worst-case input configuration for a quicksort algorithm that chooses the threshold m as the median of the first, middle, and last elements of the sequence to be sorted.
8. Array A contains m and array B contains n different integers which are not necessarily ordered:

   ```
   const  m = … ;  { length of array A }
     n = … ;  { length of array B }

   var  A: array[1 .. m] of integer;
     B: array[1 .. n] of integer;
   ```

A *duplicate* is an integer that is contained in both A and B. *Problem:* How many duplicates are there in A and B?

(a) Determine the time complexity of the brute-force algorithm that compares each integer contained in one array to all integers in the other array.

(b) Write a more efficient

```
function duplicates: integer;
```

Your solution may rearrange the integers in the arrays.

(c) What is the worst-case time complexity of your improved algorithm?