Sorting and Searching

4

Many efficient algorithms are based on sorting the input data, because sorting often makes solving the problem easier. This chapter discusses the theory and practice of sorting as an algorithm design tool.

Section 4.1 first discusses three important sorting algorithms: bubble sort, merge sort, and counting sort. After this, we will learn how to use the sorting algorithm available in the C++ standard library.

Section 4.2 shows how sorting can be used as a subroutine to create efficient algorithms. For example, to quickly determine if all array elements are unique, we can first sort the array and then simply check all pairs of consecutive elements.

Section 4.3 presents the binary search algorithm, which is another important building block of efficient algorithms.

4.1 Sorting Algorithms

The basic problem in sorting is as follows: Given an array that contains n elements, sort the elements in increasing order. For example, Fig. 4.1 shows an array before and after sorting.

In this section we will go through some fundamental sorting algorithms and examine their properties. It is easy to design an $O(n^2)$ time sorting algorithm, but there are also more efficient algorithms. After discussing the theory of sorting, we will focus on using sorting in practice in C++.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_4

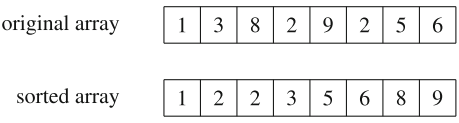

Fig. 4.1 An array before and after sorting

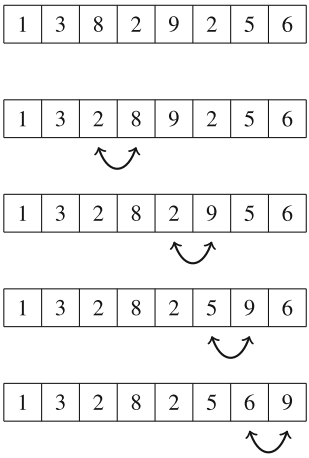

Fig. 4.2 The first round of bubble sort

4.1.1 Bubble Sort

Bubble sort is a simple sorting algorithm that works in $O(n^2)$ time. The algorithm consists of n rounds, and on each round, it iterates through the elements of the array. Whenever two consecutive elements are found that are in wrong order, the algorithm swaps them. The algorithm can be implemented as follows:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```

After the first round of bubble sort, the largest element will be in the correct position, and more generally, after k rounds, the k largest elements will be in the correct positions. Thus, after n rounds, the whole array will be sorted.

For example, Fig. 4.2 shows the first round of swaps when bubble sort is used to sort an array.

Bubble sort is an example of a sorting algorithm that always swaps *consecutive* elements in the array. It turns out that the time complexity of such an algorithm is *always* at least $O(n^2)$, because in the worst case, $O(n^2)$ swaps are required for sorting the array.

0	1	2	3	4	5	6	7
1	2	2	6	3	5	9	8

Fig. 4.3 This array has three inversions: (3, 4), (3, 5), and (6, 7)

Inversions A useful concept when analyzing sorting algorithms is an *inversion*: a pair of array indices (a, b) such that $a < b$ and $\texttt{array}[a] > \texttt{array}[b]$, i.e., the elements are in wrong order. For example, the array in Fig. 4.3 has three inversions: (3, 4), (3, 5), and (6, 7).

The number of inversions indicates how much work is needed to sort the array. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in the reverse order, the number of inversions is

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2),$$

which is the largest possible.

Swapping a pair of consecutive elements that are in the wrong order removes exactly one inversion from the array. Hence, if a sorting algorithm can only swap consecutive elements, each swap removes at most one inversion, and the time complexity of the algorithm is at least $O(n^2)$.

4.1.2 Merge Sort

If we want to create an efficient sorting algorithm, we have to be able to reorder elements that are in different parts of the array. There are several such sorting algorithms that work in $O(n \log n)$ time. One of them is *merge sort*, which is based on recursion. Merge sort sorts a subarray $\texttt{array}[a \ldots b]$ as follows:

1. If $a = b$, do not do anything, because a subarray that only contains one element is already sorted.
2. Calculate the position of the middle element: $k = \lfloor (a + b)/2 \rfloor$.
3. Recursively sort the subarray $\texttt{array}[a \ldots k]$.
4. Recursively sort the subarray $\texttt{array}[k + 1 \ldots b]$.
5. *Merge* the sorted subarrays $\texttt{array}[a \ldots k]$ and $\texttt{array}[k + 1 \ldots b]$ into a sorted subarray $\texttt{array}[a \ldots b]$.

For example, Fig. 4.4 shows how merge sort sorts an array of eight elements. First, the algorithm divides the array into two subarrays of four elements. Then, it sorts these subarrays recursively by calling itself. Finally, it merges the sorted subarrays into a sorted array of eight elements.

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step. Then, merging the sorted subarrays is possible in linear time, because they are already sorted. Since there are $O(\log n)$ recursive levels, and processing each level takes a total of $O(n)$ time, the algorithm works in $O(n \log n)$ time.

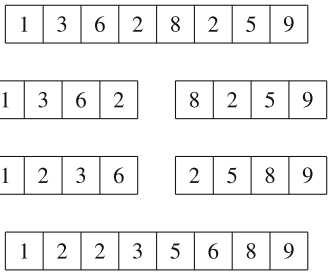

Fig. 4.4 Sorting an array using merge sort

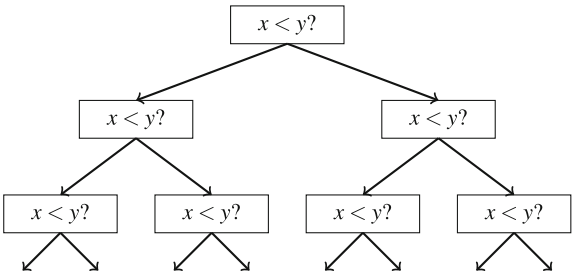

Fig. 4.5 The progress of a sorting algorithm that compares array elements

4.1.3 Sorting Lower Bound

Is it possible to sort an array faster than in $O(n \log n)$ time? It turns out that this is *not* possible when we restrict ourselves to sorting algorithms that are based on comparing array elements.

The lower bound for the time complexity can be proved by considering sorting as a process where each comparison of two elements gives more information about the contents of the array. Figure 4.5 illustrates the tree created in this process.

Here "$x < y$?" means that some elements x and y are compared. If $x < y$, the process continues to the left, and otherwise to the right. The results of the process are the possible ways to sort the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n).$$

We get a lower bound for this sum by choosing the last $n/2$ elements and changing the value of each element to $\log_2(n/2)$. This yields an estimate

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

so the height of the tree and the worst-case number of steps in a sorting algorithm is $\Omega(n \log n)$.

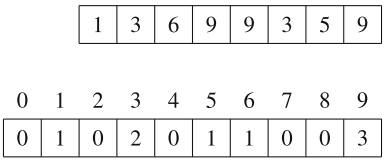

Fig. 4.6 Sorting an array using counting sort

4.1.4 Counting Sort

The lower bound $\Omega(n \log n)$ does not apply to algorithms that do not compare array elements but use some other information. An example of such an algorithm is *counting sort* that sorts an array in $O(n)$ time assuming that every element in the array is an integer between $0 \ldots c$ and $c = O(n)$.

The algorithm creates a bookkeeping array, whose indices are elements of the original array. The algorithm iterates through the original array and calculates how many times each element appears in the array. As an example, Fig. 4.6 shows an array and the corresponding bookkeeping array. For example, the value at position 3 is 2, because the value 3 appears 2 times in the original array.

The construction of the bookkeeping array takes $O(n)$ time. After this, the sorted array can be created in $O(n)$ time, because the number of occurrences of each element can be retrieved from the bookkeeping array. Thus, the total time complexity of counting sort is $O(n)$.

Counting sort is a very efficient algorithm but it can only be used when the constant c is small enough, so that the array elements can be used as indices in the bookkeeping array.

4.1.5 Sorting in Practice

In practice, it is almost never a good idea to implement a home-made sorting algorithm, because all modern programming languages have good sorting algorithms in their standard libraries. There are many reasons to use a library function: it is certainly correct and efficient, and also easy to use.

In C++, the function `sort` efficiently[1] sorts the contents of a data structure. For example, the following code sorts the elements of a vector in increasing order:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

After the sorting, the contents of the vector will be [2, 3, 3, 4, 5, 5, 8]. The default sorting order is increasing, but a reverse order is possible as follows:

[1] The C++11 standard requires that the `sort` function works in $O(n \log n)$ time; the exact implementation depends on the compiler.

```
sort(v.rbegin(),v.rend());
```

An ordinary array can be sorted as follows:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

Then, the following code sorts the string `s`:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Sorting a string means that the characters of the string are sorted. For example, the string "monkey" becomes "ekmnoy".

Comparison Operators The `sort` function requires that a *comparison operator* is defined for the data type of the elements to be sorted. When sorting, this operator will be used whenever it is necessary to find out the order of two elements.

Most C++ data types have a built-in comparison operator, and elements of those types can be sorted automatically. Numbers are sorted according to their values, and strings are sorted in alphabetical order. Pairs are sorted primarily according to their first elements and secondarily according to their second elements:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
// result: [(1,2),(1,5),(2,3)]
```

In a similar way, tuples are sorted primarily by the first element, secondarily by the second element, etc.[2]:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
// result: [(1,5,3),(2,1,3),(2,1,4)]
```

User-defined structs do not have a comparison operator automatically. The operator should be defined inside the struct as a function `operator<`, whose parameter

[2]Note that in some older compilers, the function `make_tuple` has to be used to create a tuple instead of braces (for example, `make_tuple(2,1,4)` instead of `{2,1,4}`).

is another element of the same type. The operator should return `true` if the element is smaller than the parameter, and `false` otherwise.

For example, the following struct `point` contains the x and y coordinates of a point. The comparison operator is defined so that the points are sorted primarily by the x coordinate and secondarily by the y coordinate.

```
struct point {
    int x, y;
    bool operator<(const point &p) {
        if (x == p.x) return y < p.y;
        else return x < p.x;
    }
};
```

Comparison Functions It is also possible to give an external *comparison function* to the `sort` function as a callback function. For example, the following comparison function `comp` sorts strings primarily by length and secondarily by alphabetical order:

```
bool comp(string a, string b) {
    if (a.size() == b.size()) return a < b;
    else return a.size() < b.size();
}
```

Now a vector of strings can be sorted as follows:

```
sort(v.begin(), v.end(), comp);
```

4.2 Solving Problems by Sorting

Often, we can easily solve a problem in $O(n^2)$ time using a brute force algorithm, but such an algorithm is too slow if the input size is large. In fact, a frequent goal in algorithm design is to find $O(n)$ or $O(n \log n)$ time algorithms for problems that can be trivially solved in $O(n^2)$ time. Sorting is one way to achieve this goal.

For example, suppose that we want to check if all elements in an array are unique. A brute force algorithm goes through all pairs of elements in $O(n^2)$ time:

```
bool ok = true;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        if (array[i] == array[j]) ok = false;
    }
}
```

However, we can solve the problem in $O(n \log n)$ time by first sorting the array. Then, if there are equal elements, they are next to each other in the sorted array, so they are easy to find in $O(n)$ time:

```
bool ok = true;
sort(array, array+n);
for (int i = 0; i < n-1; i++) {
    if (array[i] == array[i+1]) ok = false;
}
```

Several other problems can be solved in a similar way in $O(n \log n)$ time, such as counting the number of distinct elements, finding the most frequent element, and finding two elements whose difference is minimum.

4.2.1 Sweep Line Algorithms

A *sweep line* algorithm models a problem as a set of events that are processed in a sorted order. For example, suppose that there is a restaurant and we know the arriving and leaving times of all customers on a certain day. Our task is to find out the maximum number of customers who visited the restaurant at the same time.

For example, Fig. 4.7 shows an instance of the problem where there are four customers A, B, C, and D. In this case, the maximum number of simultaneous customers is three between A's arrival and B's leaving.

To solve the problem, we create two events for each customer: one event for arrival and another event for leaving. Then, we sort the events and go through them according to their times. To find the maximum number of customers, we maintain a counter whose value increases when a customer arrives and decreases when a customer leaves. The largest value of the counter is the answer to the problem.

Figure 4.8 shows the events in our example scenario. Each customer is assigned two events: "+" denotes an arriving customer and "−" denotes a leaving customer. The resulting algorithm works in $O(n \log n)$ time, because sorting the events takes $O(n \log n)$ time and the sweep line part takes $O(n)$ time.

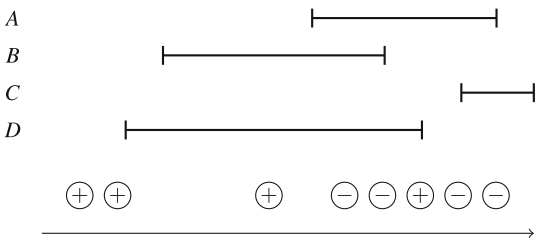

Fig. 4.7 An instance of the restaurant problem

Fig. 4.8 Solving the restaurant problem using a sweep line algorithm

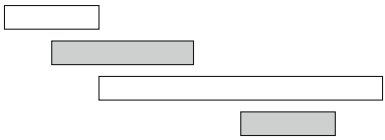

Fig. 4.9 An instance of the scheduling problem and an optimal solution with two events

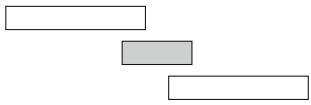

Fig. 4.10 If we select the short event, we can only select one event, but we could select both long events

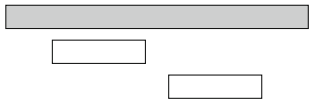

Fig. 4.11 If we select the first event, we cannot select any other events, but we could to select the other two events

4.2.2 Scheduling Events

Many scheduling problems can be solved by sorting the input data and then using a *greedy* strategy to construct a solution. A greedy algorithm always makes a choice that looks the best at the moment and never takes back its choices.

As an example, consider the following problem: Given n events with their starting and ending times, find a schedule that includes as many events as possible. For example, Fig. 4.9 shows an instance of the problem where an optimal solution is to select two events.

In this problem, there are several ways how we could sort the input data. One strategy is to sort the events according to their *lengths* and select as *short* events as possible. However, this strategy does not always work, as shown in Fig. 4.10. Then, another idea is to sort the events according to their *starting times* and always select the next possible event that *begins* as *early* as possible. However, we can find a counterexample also for this strategy, shown in Fig. 4.11.

A third idea is to sort the events according to their *ending times* and always select the next possible event that *ends* as *early* as possible. It turns out that this algorithm *always* produces an optimal solution. To justify this, consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices left how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

4.2.3 Tasks and Deadlines

Finally, consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?

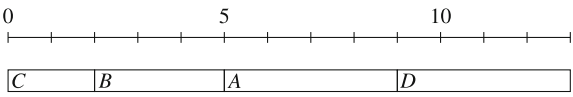

Fig. 4.12 An optimal schedule for the tasks

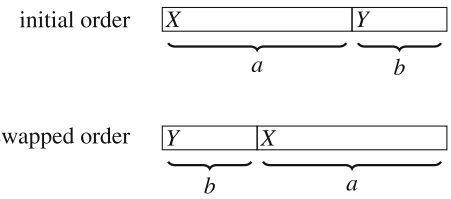

Fig. 4.13 Improving the solution by swapping tasks X and Y

For example, suppose that the tasks are as follows:

task	duration	deadline
A	4	2
B	3	10
C	2	8
D	4	15

Figure 4.12 shows an optimal schedule for the tasks in our example scenario. Using this schedule, C yields 6 points, B yields 5 points, A yields -7 points, and D yields 2 points, so the total score is 6.

It turns out that the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks.

For example, in Fig. 4.13, there are two tasks X and Y with durations a and b. Initially, X is scheduled before Y. However, since $a > b$, the tasks should be swapped. Now X gives b points less and Y gives a points more, so the total score increases by $a - b > 0$. Thus, in an optimal solution, a shorter task must always come before a longer task, and the tasks must be sorted by their durations.

4.3 Binary Search

Binary search is an $O(\log n)$ time algorithm that can be used, for example, to efficiently check whether a sorted array contains a given element. In this section, we first focus on the implementation of binary search, and after that, we will see how binary search can be used to find optimal solutions for problems.

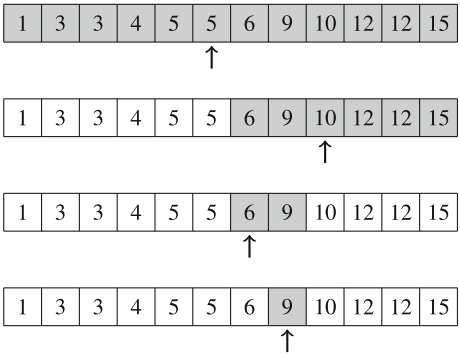

Fig. 4.14 The traditional way to implement binary search. At each step we check the middle element of the active subarray and proceed to the left or right part

4.3.1 Implementing the Search

Suppose that we are given a sorted array of n elements and we want to check if the array contains an element with a target value x. Next we discuss two ways to implement a binary search algorithm for this problem.

First Method The most common way to implement binary search resembles looking for a word in a dictionary.[3] The search maintains an active subarray in the array, which initially contains all array elements. Then, a number of steps are performed, each of which halves the search range. At each step, the search checks the middle element of the active subarray. If the middle element has the target value, the search terminates. Otherwise, the search recursively continues to the left or right half of the subarray, depending on the value of the middle element. For example, Fig. 4.14 shows how an element with value 9 is found in the array.

The search can be implemented as follows:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] < x) a = k+1;
    else b = k-1;
}
```

In this implementation, the range of the active subarray is $a \ldots b$, and the initial range is $0 \ldots n-1$. The algorithm halves the size of the subarray at each step, so the time complexity is $O(\log n)$.

[3] Some people, including the author of this book, still use printed dictionaries. Another example is finding a phone number in a printed phone book, which is even more obsolete.

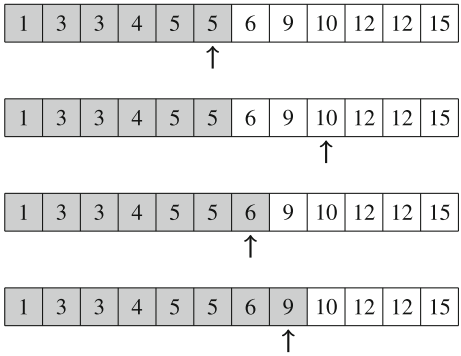

Fig. 4.15 An alternative way to implement binary search. We scan the array from left to right jumping over elements

Second Method Another way to implement binary search is to go through the array from left to right making *jumps*. The initial jump length is $n/2$, and the jump length is halved on each round: first $n/4$, then $n/8$, then $n/16$, etc., until finally the length is 1. On each round, we make jumps until we would end up outside the array or in an element whose value exceeds the target value. After the jumps, either the desired element has been found or we know that it does not appear in the array. Figure 4.15 illustrates the technique in our example scenario.

The following code implements the search:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

During the search, the variable b contains the current jump length. The time complexity of the algorithm is $O(\log n)$, because the code in the `while` loop is performed at most twice for each jump length.

4.3.2 Finding Optimal Solutions

Suppose that we are solving a problem and have a function `valid`(x) that returns `true` if x is a valid solution and `false` otherwise. In addition, we know that `valid`(x) is `false` when $x < k$ and `true` when $x \geq k$. In this situation, we can use binary search to efficiently find the value of k.

The idea is to binary search for the largest value of x for which `valid`(x) is `false`. Thus, the next value $k = x + 1$ is the smallest possible value for which `valid`(k) is `true`. The search can be implemented as follows:

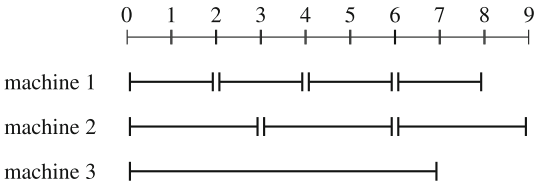

Fig. 4.16 An optimal processing schedule: machine 1 processes four jobs, machine 2 processes three jobs, and machine 3 processes one job

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!valid(x+b)) x += b;
}
int k = x+1;
```

The initial jump length z has to be an upper bound for the answer, i.e., any value for which we surely know that `valid`(z) is `true`. The algorithm calls the function `valid` $O(\log z)$ times, so the running time depends on the function `valid`. For example, if the function works in $O(n)$ time, the running time is $O(n \log z)$.

Example Consider a problem where our task is to process k jobs using n machines. Each machine i is assigned an integer p_i: the time to process a single job. What is the minimum time to process all the jobs?

For example, suppose that $k = 8$, $n = 3$ and the processing times are $p_1 = 2$, $p_2 = 3$, and $p_3 = 7$. In this case, the minimum total processing time is 9, by following the schedule in Fig. 4.16.

Let `valid`(x) be a function that finds out whether it is possible to process all the jobs using at most x units of time. In our example scenario, clearly `valid`(9) is `true`, because we can follow the schedule in Fig. 4.16. On the other hand, `valid`(8) must be `false`, because the minimum processing time is 9.

Calculating the value of `valid`(x) is easy, because each machine i can process at most $\lfloor x/p_i \rfloor$ jobs in x units of time. Thus, if the sum of all $\lfloor x/p_i \rfloor$ values is k or more, x is a valid solution. Then, we can use binary search to find the minimum value of x for which `valid`(x) is `true`.

How efficient is the resulting algorithm? The function `valid` takes $O(n)$ time, so the algorithm works in $O(n \log z)$ time, where z is an upper bound for the answer. One possible value for z is kp_1 which corresponds to a solution where only the first machine is used to process all the jobs. This is surely a valid upper bound.