Chapter 3

Sorting

Sorting is a fundamental algorithm design problem. Many efficient algorithms use sorting as a subroutine, because it is often easier to process data if the elements are in a sorted order.

For example, the problem ”does an array contain two equal elements?” is easy to solve using sorting. If the array contains two equal elements, they will be next to each other after sorting, so it is easy to find them. Also, the problem ”what is the most frequent element in an array?” can be solved similarly.

There are many algorithms for sorting, and they are also good examples of how to apply different algorithm design techniques. The efficient general sorting algorithms work in $O(n \log n)$ time, and many algorithms that use sorting as a subroutine also have this time complexity.

Sorting theory

The basic problem in sorting is as follows:

> Given an array that contains n elements, your task is to sort the elements in increasing order.

For example, the array

1	3	8	2	9	2	5	6

will be as follows after sorting:

1	2	2	3	5	6	8	9

$O(n^2)$ algorithms

Simple algorithms for sorting an array work in $O(n^2)$ time. Such algorithms are short and usually consist of two nested loops. A famous $O(n^2)$ time sorting algorithm is **bubble sort** where the elements ”bubble” in the array according to their values.

Bubble sort consists of n rounds. On each round, the algorithm iterates through the elements of the array. Whenever two consecutive elements are found that are not in correct order, the algorithm swaps them. The algorithm can be implemented as follows:

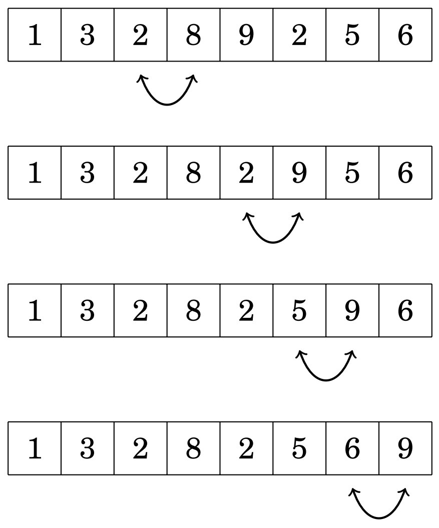
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```

After the first round of the algorithm, the largest element will be in the correct position, and in general, 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, in the array

1	3	8	2	9	2	5	6

the first round of bubble sort swaps elements as follows:



Inversions

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.

A useful concept when analyzing sorting algorithms is an **inversion**: a pair of array elements ($\texttt{array}[a], \texttt{array}[b]$) such that $a < b$ and $\texttt{array}[a] > \texttt{array}[b]$, i.e., the elements are in the wrong order. For example, the array

1	2	2	6	3	5	9	8

has three inversions: (6,3), (6,5) and (9,8). 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 the largest possible:

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

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 $O(n^2)$.

$O(n \log n)$ algorithms

It is possible to sort an array efficiently in $O(n \log n)$ time using algorithms that are not limited to swapping consecutive elements. One such algorithm is **merge sort**[1], which is based on recursion.

Merge sort sorts a subarray `array`$[a \dots b]$ as follows:

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

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step. The recursion consists of $O(\log n)$ levels, and processing each level takes $O(n)$ time. Merging the subarrays `array`$[a \dots k]$ and `array`$[k+1 \dots b]$ is possible in linear time, because they are already sorted.

For example, consider sorting the following array:

1	3	6	2	8	2	5	9

The array will be divided into two subarrays as follows:

1	3	6	2

8	2	5	9

Then, the subarrays will be sorted recursively as follows:

1	2	3	6

2	5	8	9

[1]According to [47], merge sort was invented by J. von Neumann in 1945.

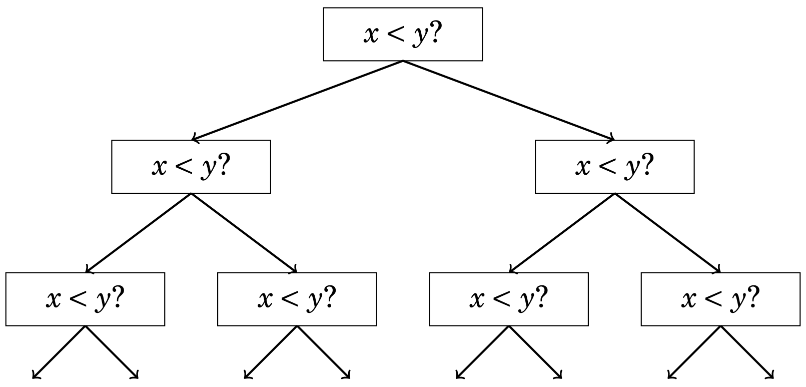
Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

1	2	2	3	5	6	8	9

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. The process creates the following tree:



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

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

so the height of the tree and the minimum possible number of steps in a sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ 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.

For example, the array

1	3	6	9	9	3	5	9

corresponds to the following bookkeeping array:

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

For example, the value at position 3 in the bookkeeping array is 2, because the element 3 appears 2 times in the original array.

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.

Sorting in C++

It is almost never a good idea to use a home-made sorting algorithm in a contest, because there are good implementations available in programming languages. For example, the C++ standard library contains the function `sort` that can be easily used for sorting arrays and other data structures.

There are many benefits in using a library function. First, it saves time because there is no need to implement the function. Second, the library implementation is certainly correct and efficient: it is not probable that a home-made sorting function would be better.

In this section we will see how to use the C++ `sort` function. The following code sorts 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:

```
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);
```

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 function `sort` 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. For example, numbers are sorted according to their values and strings are sorted in alphabetical order.

Pairs (`pair`) are sorted primarily according to their first elements (`first`). However, if the first elements of two pairs are equal, they are sorted according to their second elements (`second`):

```
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());
```

After this, the order of the pairs is $(1,2)$, $(1,5)$ and $(2,3)$.

In a similar way, tuples (`tuple`) 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());
```

After this, the order of the tuples is $(1,5,3)$, $(2,1,3)$ and $(2,1,4)$.

User-defined structs

User-defined structs do not have a comparison operator automatically. The operator should be defined inside the struct as a function `operator<`, whose parameter 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 `P` contains the x and y coordinates of a point. The comparison operator is defined so that the points are sorted primarily by the

[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}`).

x coordinate and secondarily by the y coordinate.

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

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.size() < b.size();
    return a < b;
}
```

Now a vector of strings can be sorted as follows:

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

Binary search

A general method for searching for an element in an array is to use a `for` loop that iterates through the elements of the array. For example, the following code searches for an element x in an array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, this is also the best possible approach, because there is no additional information available where in the array we should search for the element x.

However, if the array is *sorted*, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Method 1

The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region.

At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element.

The above idea 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) b = k-1;
    else a = k+1;
}
```

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

Method 2

An alternative method to implement binary search is based on an efficient way to iterate through the elements of the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to right, and the initial jump length is $n/2$. At each step, the jump length will be halved: first $n/4$, then $n/8$, $n/16$, etc., until finally the length is 1. After the jumps, either the target element has been found or we know that it does not appear in the array.

The following code implements the above idea:

```
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.

C++ functions

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- `lower_bound` returns a pointer to the first array element whose value is at least x.
- `upper_bound` returns a pointer to the first array element whose value is larger than x.
- `equal_range` returns both above pointers.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element. For example, the following code finds out whether an array contains an element with value x:

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Then, the following code counts the number of elements whose value is x:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

Using `equal_range`, the code becomes shorter:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

Finding the smallest solution

An important use for binary search is to find the position where the value of a *function* changes. Suppose that we wish to find the smallest value k that is a valid solution for a problem. We are given a function ok(x) that returns `true` if x is a valid solution and `false` otherwise. In addition, we know that ok(x) is `false` when $x < k$ and `true` when $x \geq k$. The situation looks as follows:

x	0	1	$\cdots$	$k-1$	k	$k+1$	$\cdots$
ok(x)	`false`	`false`	$\cdots$	`false`	`true`	`true`	$\cdots$

Now, the value of k can be found using binary search:

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

The search finds the largest value of x for which ok(x) is `false`. Thus, the next value $k = x+1$ is the smallest possible value for which ok(k) is `true`. The initial jump length z has to be large enough, for example some value for which we know beforehand that ok(z) is `true`.

The algorithm calls the function ok $O(\log z)$ times, so the total time complexity depends on the function ok. For example, if the function works in $O(n)$ time, the total time complexity is $O(n \log z)$.

Finding the maximum value

Binary search can also be used to find the maximum value for a function that is first increasing and then decreasing. Our task is to find a position k such that

- $f(x) < f(x+1)$ when $x < k$, and
- $f(x) > f(x+1)$ when $x \geq k$.

The idea is to use binary search for finding the largest value of x for which $f(x) < f(x+1)$. This implies that $k = x+1$ because $f(x+1) > f(x+2)$. The following code implements the search:

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

Note that unlike in the ordinary binary search, here it is not allowed that consecutive values of the function are equal. In this case it would not be possible to know how to continue the search.