Sorting is a classical algorithmic problem where the goal is to reorder the elements according to their value. There are efficient algorithms for sorting in $O(n \log n)$ time.

In this chapter, we will see how to do sorting in Python and how sorting can be utilized in implementing efficient algorithms. We will also take a look at a bit of the theory of sorting and some common sorting algorithms.

Sorting in Python

A Python list can be sorted with the method `sort`:

```
numbers = [4, 2, 1, 2]
numbers.sort()
print(numbers) # [1, 2, 2, 4]
```

In addition, Python has the function `sorted` that returns a sorted list:

```
numbers = [4, 2, 1, 2]
print(sorted(numbers)) # [1, 2, 2, 4]
```

The difference between the two ways of sorting a list is that the method `sort` modifies the list while the function `sorted` creates a new list and leaves the original list unmodified.

The time complexity of both the method `sort` and the function `sorted` is $O(n \log n)$ allowing their use within efficient algorithms.

Example: Smallest difference

Task

You are given a list of numbers. What is the smallest difference between two elements?

For example, whe the list is $[4, 1, 7, 3, 9]$, the desired answer is 1, because the smallest difference is between the numbers 3 and 4.

In a sorted list, the smallest difference is always between two adjacent elements. Thus we can solve the problem efficiently by sorting the list and going through all pairs of adjacent elements. For example, sorting the list $[4, 1, 7, 3, 9]$ produces the list $[1, 3, 4, 7, 9]$, where the elements 3 and 4 with the smallest difference are adjacent.

The following function `min_diff` implements the algorithm:

```python
def min_diff(numbers):
    numbers = sorted(numbers)

    result = numbers[1] - numbers[0]
    for i in range(2, len(numbers)):
        result = min(result, numbers[i] - numbers[i - 1])

    return result
```

The algorithm first sorts the list using the function `sorted` and then computes the smallest difference of adjacent elements into the the variable `result`.

The sorting takes $O(n \log n)$ time and going through the list takes $O(n)$ time, which gives $O(n \log n)$ as the total time complexity of the algorithm.

Avoiding side effects

Notice that we could have used the method `sort` instead of the function `sorted`:

```python
def min_diff(numbers):
    numbers.sort()
    ...
```

The trouble with this, however, is the side effect of modifying the list, which affects the list outside the function `min_diff` too. The following code segments illustrate why this is undesirable:

Using `sort`

```
numbers = [4, 1, 7, 3,9]
print(min_diff(numbers)) # 1
print(numbers) # [1, 3, 4, 7, 9]
```

Using `sorted`

```
numbers = [4, 1, 7, 3,9]
print(min_diff(numbers)) # 1
print(numbers) # [4, 1, 7, 3,9]
```

When using the method `sort`, the ordering of the list changes, which may come as a surprise for someone who knows what the function `min_diff` does (returns the smallest difference) but not how it does it. It is better to avoid this side effect by using the function `sorted`.

Hashing vs. sorting

Many tasks have two possibilities for an efficient solution: hashing and sorting. Let us consider an example that we solved with hashing earlier:

Task

You are given a list of numbers. How many distinct numbers does it contain?

For example, when the list is $[3, 1, 2, 1, 5, 2, 2, 3]$, the desired answer is 4, because the distinct numbers are 1, 2, 3 and 5.

Below is the earlier solution using hashing:

Algorithm 1

```
def count_distinct(numbers):
    seen = set()

    for x in numbers:
        seen.add(x)
    return len(seen)
```

An alternative solution uses sorting. Since equal numbers are contiguous in a sorted list, we can go through the sorted list and increment a counter whenever the number changes.

Below is the algorithm that uses sorting:

Algorithm 2

```
def count_distinct(numbers):
    numbers = sorted(numbers)

    result = 1
    for i in range(1, len(numbers)):
        if numbers[i] != numbers[i - 1]:
            result += 1

    return result
```

The time complexity is $O(n)$ when using hashing and $O(n \log n)$ when using sorting, but how fast are they in practice?

Let us do a comparison test. Since both algorithms are efficient, we can use large test inputs ($n = 10^7$). The input list contains random numbers in the range $1 \dots k$ for some integer k. By varying the value of k, we can control how many of the numbers are equal, which might affect the running time.

Here are the test results for the test computer:

Upper bound k	**Algorithm 1 (hashing)**	**Algorithm 2 (sorting)**
10^3	0.46 s	3.18 s
10^4	0.56 s	4.50 s
10^5	1.16 s	5.74 s
10^6	2.56 s	6.38 s
10^7	2.56 s	6.48 s

In this case, the algorithm using hashing appears to be more efficient. It is also slightly shorter and simpler.

More about sorting

Sorting into reverse order

The parameter `reverse` changes the ordering from ascending to descending:

```python
numbers = [2, 4, 3, 5, 6, 1, 8, 7]
numbers.sort(reverse=True)
print(numbers) # [8, 7, 6, 5, 4, 3, 2, 1]
```

Multipart elements

If the elements to be sorted are tuples or lists, the first element of the tuple or list is the primary sorting key, the second element is the secondary key, etc..

For example, a list of pairs gets sorted as follows:

```python
pairs = [(3, 5), (1, 3), (1, 2), (2, 4)]
pairs.sort()
print(pairs) # [(1, 2), (1, 3), (2, 4), (3, 5)]
```

Element comparisons

The parameter `key` can be used for defining a function that is applied to each element before any comparison. Below is an example of using the parameter:

```python
numbers = [4, -1, 6, 2, -7, 8, 3, -5]
numbers.sort(key=abs)
print(numbers) # [-1, 2, 3, 4, -5, 6, -7, 8]
```

Here the key function is `abs` (absolute value), which causes the sorting to ignore the minus signs.

Own class

In Python, objects of a class can be sorted if the class defines sufficient methods for comparing objects. For example, it is enough to define the methods `__eq__` and `__lt__`, which are called when the objects are compared with the operators

`==` and `<`. The following code illustrates this:

```python
class Location:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)

    def __lt__(self, other):
        return (self.x, self.y) < (other.x, other.y)

    def __repr__(self):
        return str((self.x, self.y))

locations = []
locations.append(Location(1, 4))
locations.append(Location(4, 5))
locations.append(Location(2, 2))
locations.append(Location(1, 2))

locations.sort()

print(locations) # [(1, 2), (1, 4), (2, 2), (4, 5)]
```

Example: Restaurant

Task

A restaurant is visited by n customers during a given day, and you know the arrival and departure time of each customer. If a customer departs at the same moment as another arrives, they are both considered to be in the restaurant at that moment. Your task is to find out the highest number of customers that are in the restaurant at the same time.

Consider the following example case:

Customer	Arrival time	Departure time
#1	6	8

Customer	Arrival time	Departure time
#2	3	7
#3	6	9
#4	1	5
#5	2	8

Here the highest number of customers is 4: The customers #1, #2, #3 and #5 are all in the restaurant during the time period 6–7.

A useful approach in this kind of a task is to process events in order of time. There are two types of events: customer arrivals and customer departures. With each event the number of customers increases or decreases by one.

In our example case, the events are:

Time	Event	Customer count
1	Customer #4 arrives	1
2	Customer #5 arrives	2
3	Customer #2 arrives	3
5	Customer #4 departs	2
6	Customer #1 arrives	3
6	Customer #3 arrives	4
7	Customer #2 departs	3
8	Customer #1 departs	2
8	Customer #5 departs	1
9	Customer #3 departs	0

We can solve the task by creating a list of the customer events and sorting it by time. Then we can go through the list while maintaining a counter of customers.

In the following function, the lists `arrivals` and `departures` contain the customer arrival and departure times. In our example case, `arrivals` is `[6, 3, 6, 1, 2]` and `departures` is `[8, 7, 9, 5, 8]`.

```python
def max_customers(arrivals, departures):
    events = []
    for time in arrivals:
        events.append((time, 1))
    for time in departures:
        events.append((time, 2))

    events.sort()

    counter = 0
    result = 0
    for event in events:
        if event[1] == 1:
            counter += 1
        if event[1] == 2:
            counter -= 1
        result = max(result, counter)

    return result
```

The function creates the list `events` and adds the customer arrival and departure events to the list as pairs. An arrival is represented as a pair `(time, 1)` and a departure as a pair `(time, 2)`. Then the list of events is sorted using the time as the primary key.

Next the function goes through the list of events while keeping track of the customer count in the variable `counter`. The variable `result` stores the highest customer count encountered so far.

The resulting algorithm consists of three parts. Creating the event list takes $O(n)$ time, since there are two events for each customer. Sorting the events list takes $O(n \log n)$ time, and finally, iterating through the events takes $O(n)$ time. Thus the total time complexity of the algorithm is $O(n \log n)$.

How is sorting done?

A sorting algorithm is given a list of elements and the goal is to reorder them by their value. A typical algorithm can compare elements to each other and move elements to a different position in the list.

Simple sorting algorithms compare adjacent elements and swap them when needed. Their time complexity is $O(n^2)$. One such algorithm is insertion sort that processes the elements from left to right and moves each element to its correct position among the already processed elements.

There are also efficient algorithms with time complexity $O(n \log n)$. One such algorithm is merge sort that starts by sorting the first half and the second half of the list separately using recursion, and then merges the two halfs into a single sorted list.

The Python sorting algorithm, called *Timsort*, is a variant of merge sort but utilizes insertion sort too. Its time complexity is $O(n \log n)$, and it is designed to operate particularly fast in certain situations that are common in real life data.

There can be no general purpose sorting algorithm with a time complexity better than $O(n \log n)$. This can be shown by proving a lower bound for how many comparisons is needed for sorting in the worst case.

What happens during sorting?

The following code can be used for finding out what happens when Python sorts a list:

```python
import functools

def cmp(a, b):
    print("compare", a, b)
    return a - b

numbers = [4, 1, 3, 2]
numbers.sort(key=functools.cmp_to_key(cmp))
print(numbers)
```

When the method `sort` is called as above, it compares the elements on the list by calling the function `cmp`. The function `cmp` takes the elements `a` and `b` as parameters and must return

- a negative value if `a` is smaller than `b`,
- a positive value if `a` is bigger than `b`, and
- zero if `a` and `b` are equal.

Here `cmp` has been implemented so that it returns the value of the expression `a - b`.

The method `sort` changes the ordering of the elements based on the calls to the function `cmp`. Here the function `cmp` prints out all the comparisons it makes, which provides us with information on comparisons made by the sorting algorithm. The output of the code is as follows:

```
compare 1 4
compare 3 1
compare 3 4
compare 3 1
compare 2 3
compare 2 1
[1, 2, 3, 4]
```

This means that the method `sort` first compares the numbers 1 and 4, then the numbers 3 and 1, and so on. The method needs six comparisons to complete the sorting.

Sorting in other languages

Different programming languages have different ways of doing sorting. The sorting can be based on different the algorithms, but typically built-in implementations use efficient algorithms and are well tuned for fast performance.

C++ has the function `std::sort` that is given an iterator to the beginning and to the end of the interval to be sorted. The function is used as follows:

```
std::vector<int> numbers;
...
std::sort(numbers.begin(), numbers.end());
```

Java has the method `sort` of the `Collections` class:

```
ArrayList<Integer> numbers = new ArrayList<>();
...
Collections.sort(numbers);
```

In JavaScript, an array is sorted using the method `sort`. Using the method can produce a surprise:

```
numbers = [2, 11, 1, 3];
numbers.sort();
console.log(numbers); // [1, 11, 2, 3]
```

Here a list of integers gets sorted so that the number 11 is before the number 2. This is because the method `sort` treats the elements as strings by default, and the string `11` is smaller than the string `2`. The method can be given an integer comparison function as a parameter as follows:

```
numbers = [2, 11, 1, 3];
numbers.sort((a, b) => a - b);
console.log(numbers); // [1, 2, 3, 11]
```

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI