The topic of this and the following two chapters is the design of efficient algorithms. We aim for algorithms that execute efficiently even when the input size n is big.

A common situation in algorithm design is one where it is easy to design a straightforward algorithm that solves the problem with two nested loops in $O(n^2)$ time. This kind of an algorithm can be called a *brute force* algorithm. But if n is big, a more efficient algorithm may be necessary.

In practice, an efficient algorithm is often required to have a time complexity $O(n)$ or $O(n \log n)$. We will first take a look at $O(n)$ time algorithms that scan through the input in one loop while maintaining some additional data. The time complexity $O(n \log n)$ often arises from the use of sorting, which will be covered in Chapter 5.

Outline of an efficient algorithm

A typical efficient algorithm might be structured something like this:

```
# define variables
for ...
    # efficient code
# return answer
```

An efficient algorithm typically has a single for-loop that goes through the input from left to right. The code inside the loop is efficient so that each round in the loop takes $O(1)$ time. Then the time complexity of the whole algorithm is $O(n)$.

A loop in an efficient algorithm may contain the following:

- updates of variable values using other variables or individual elements of the input
- arithmetic expressions related to variable updates
- if-commands that affect the variable updates

But the loop may not contain:

- other loops that go through the input
- slow operations that process the input (for example, `count` or the slice operation `[:]`)
- slow function calls (for example, `sum`, `min` or `max` applied to the whole input)

A major challenge in the design of many algorithms is to figure out how to implement the algorithm so that the loop contains only efficient code. We will next see examples of how to achieve this.

Example: Stock trading

Task

You are given the price of a stock for n days. Your task is figure out the highest profit you could have made if you had bought the stock on one day and sold it on another day.

Consider the following situation:

Day	0	1	2	3	4	5	6	7
Price	3	7	5	1	4	6	2	3

Here the highest profit is 6 – 1 = 5, achieved by buying on day 3 and selling on day 5.

A straightforward algorithm for solving this problem iterates through all combinations of buying and selling days. The following function `best_profit` implements the algorithm:

```python
def best_profit(prices):
    n = len(prices)
    best = 0
    for i in range(n):
        for j in range(i + 1, n):
            best = max(best, prices[j] - prices[i])
    return best
```

The variable `i` indicates the buying day and the variable `j` the selling day. The profit is computed for each combination of days, and the variable `best` remembers the highest profit encounted so far. This is a correct algorithm but its time complexity is $O(n^2)$, which makes it slow for big n. We would like to have a more efficient algorithm that has only one loop.

Let us consider how a single loop algorithm might work. When the loop reaches a given day, what is the highest profit possible if we sell on that day? The profit is maximized if we bought the stock at the lowest price on any of the preceding days. Thus each possible selling day should be paired with lowest buying price on the preceding days.

This idea is implemented in the following algorithm:

```
def best_profit(prices):
    n = len(prices)
    best = 0
    for i in range(n):
        min_price = min(prices[0:i+1])
        best = max(best, prices[i] - min_price)
    return best
```

The variable `i` now indicates the selling day. The algorithm computes the lowest price up to day `i` into the variable `min_price`. This is implemented with the `min` function over the beginning part `prices[0:i+1]` of the list. Then the highest profit possible when selling on day `i` can be computed as `prices[i] - min_price`.

This is again a correct algorithm and has only one loop, but it is still not efficient. The problem is that computing `min_price` takes too much time, because the function `min` has to scan through all the preceding elements, which takes $O(n)$ time. In essence, there is a hidden second loop inside the function `min`. Thus the total time complexity is still $O(n^2)$.

We can fix the problem as follows:

```
def best_profit(prices):
    n = len(prices)
    best = 0
    min_price = prices[0]
    for i in range(n):
        min_price = min(min_price, prices[i])
```

```python
        best = max(best, prices[i] - min_price)
    return best
```

Now the value of the variable `min_price` is not computed from scratch each time, but instead each new value is computed efficiently from the previous one. With this modification, each round of the loop needs only $O(1)$ time and the time complexity of the whole algorithm is $O(n)$, making it efficient.

Notice that the function `min` can be slow or fast. Computing the smallest value on a long list is slow, but computing the smaller of two values is fast.

Is the algorithm correct?

The operating logic of an efficient algorithm is often more complicated than that of a straightforward brute force algorithm. This can make it more difficult to determine if the algorithm works correctly.

A useful way to test the correctness of an algorithm is to *compare* its output with a simpler correct algorithm. This can be automated so that the algorithms are tested on a large number of random inputs. For example, the above algorithms can be tested as follows:

```python
import random

def best_profit_brute(prices):
    ...

def best_profit_fast(prices):
    ...

while True:
    n = random.randint(1, 20)
    prices = [random.randint(1, 10) for _ in range(n)]

    result_brute = best_profit_brute(prices)
    result_fast = best_profit_fast(prices)

    print(prices, result_brute, result_fast)

    if result_brute != result_fast:
        print("ERROR")
        break
```

Here the function `best_profit_brute` implements the brute force algorithm and the function `best_profit_fast` implements the efficient algorithm. The main program generates random lists with the length n in the range $1 \dots 20$ and the prices in the range $1 \dots 10$. After each test run, the program prints out the list used in the test and the outputs of the two functions. The output might look like this:

```
[2, 4, 5, 4, 2, 4, 8, 7, 5] 6 6
[8, 8, 8, 3, 6, 4, 9, 3, 2, 5, 4, 5, 2] 6 6
[9, 3, 1, 5, 8, 9, 3] 8 8
[3, 6, 7] 4 4
[6, 8, 7, 10, 8, 6, 1, 1, 2, 2, 8, 9, 10] 9 9
[4, 5, 3, 4, 5] 2 2
[3, 6, 2] 3 3
[4, 3, 8, 10, 7, 3, 4, 7, 5, 1, 7, 8, 7] 7 7
...
```

The matching outputs provide some assurance of the correctness of the efficient algorithm. If the program finds an input, where the outputs differ, it prints an error and exits. We can then try to figure out why the algorithm produced an incorrect output.

Example: Bit string

Task

You are given a bit string consisting of the characters `0` and `1`. How many ways can you select two positions in the bit string so that the left position contains the bit `0` and the right position contains the bit `1`?

Consider the following situation:

Position	0	1	2	3	4	5	6	7
Bit	0	1	0	0	1	0	1	1

Here there are 12 such pairs of positions.

A straightforward solution is to iterate through all possible pairs of positions and count the number of times with `0` on the left and `1` on the right:

```python
def count_ways(bits):
    n = len(bits)
    result = 0
    for i in range(n):
        for j in range(i + 1, n):
            if bits[i] == '0' and bits[j] == '1':
                result += 1
    return result
```

Again, the algorithm is too slow as its time complexity is $O(n^2)$.

Let us think about how we could solve the task with a single loop. As with the stock trading problem, a good approach is to consider all pairs ending at the current position simultaneously. More precisely, at a position `i`, we need an efficient way of counting the pairs with a bit `1` at position `i` and a bit `0` at a position before `i`.

If the bit at position `i` is `0`, the count is obviously 0. If the bit at position `i` is `1`, we need to know how many of the preceding positions contain a `0` bit. We get this number efficiently by keeping track of the number of `0` bits seen so far. Here is an implementation of this idea:

```python
def count_ways(bits):
    n = len(bits)
    result = 0
    zeros = 0
    for i in range(len(bits)):
        if bits[i] == '0':
            zeros += 1
        if bits[i] == '1':
            result += zeros
    return result
```

The code executed within the loop depends on whether the bit at the current position is `0` or `1`. If the bit is `0`, we increment the variable `zeros` that stores the number of zeros seen so far. If the bit is `1`, we add the value `zeros` to the variable `result`, corresponding to the desired pairs with `i` as the right position.

The algorithm has a single loop that scans through the input, and the code inside the loop needs $O(1)$ time. Thus the algorithm is efficient as it runs in $O(n)$ time.

Example: List splitting

Task

You are given a list containing n integers. Your task is to count how many ways one can split the list into two parts so that both parts have the same total sum of elements.

Consider the following example list:

Position	0	1	2	3	4	5	6	7
Number	1	-1	1	-1	1	-1	1	-1

Here the number of ways is 3. We can split the list between positions 1 and 2, between positions 3 and 4, and between positions 5 and 6.

Here is a straightforward algorithm for the task:

```python
def count_splits(numbers):
    n = len(numbers)
    result = 0
    for i in range(n - 1):
        left_sum = sum(numbers[0:i+1])
        right_sum = sum(numbers[i+1:])
        if left_sum == right_sum:
            result += 1
    return result
```

The algorithm goes through all ways of splitting the list, and computes the sums of the resulting parts into the variables `left_sum` and `right_sum`. If the sums are the same, the count stored in the variable `result` is incremented by one. The time complexity of the algorithm is $O(n^2)$, because computing the two sums takes $O(n)$ time.

Since the algorithm goes through the elements from left to right, we can compute `left_sum` more efficiently by incrementing the previous value at each step:

```python
def count_splits(numbers):
    n = len(numbers)
    result = 0
    left_sum = 0
    for i in range(n - 1):
        left_sum += numbers[i]
        right_sum = sum(numbers[i+1:])
        if left_sum == right_sum:
            result += 1
    return result
```

This is still not fast enough, because computing `right_sum` is still slow, and the trick we used for `left_sum` does not work for `right_sum`, because the list is processed in left-to-right order. Even with the faster computation of `left_sum`, the time complexity is still $O(n^2)$.

For further improvement, we can utilize the following observation: If we know the sum of the *whole* list in addition to `left_sum`, we can compute `right_sum` efficiently:

```
def count_splits(numbers):
    n = len(numbers)
    result = 0
    left_sum = 0
    total_sum = sum(numbers)
    for i in range(n - 1):
        left_sum += numbers[i]
        right_sum = total_sum - left_sum
        if left_sum == right_sum:
            result += 1
    return result
```

Since the total sum does not change during the loop, we can compute it into the variable `total_sum` before the loop. This takes $O(n)$ time but it is done only once. Then, inside the loop, `right_sum` can be computed as the diffirence `total_sum - left_sum`. The total time complexity of the obtained algorithm is $O(n)$.

Example: Sublists

Task

You are given a list containing n integers. How many ways can we choose a

> sublist that contains exactly two distinct integers?
>
> For example, the list $[1, 2, 3, 3, 2, 2, 4, 2]$ has 14 such sublists.

This task is harder than the ones above, but the same approach works here too: Go through the list, and at each position, compute how many solutions end at the current position.

With the example list, we should obtain the following counts at each position:

Index	0	1	2	3	4	5	6	7
Number	1	2	3	3	2	2	4	2
Count	0	1	1	1	3	3	2	3

For example, the count at the position 5 is 3, because the valid sublists ending at the position 5 are $[2, 3, 3, 2, 2]$, $[3, 3, 2, 2]$ and $[3, 2, 2]$.

We can compute the count of sublists ending at a position `i` efficiently with two variables: `a` points to the nearest preceding position that contains a different value than the one in the position `i`, and `b` points to the nearest preceding position whose value differs from both of the values at the positions `i` and `a`. These two values are useful, because a valid sublist ending at `i` must start after the position `b` and before or at the position `a`. Thus the number of valid sublists ending at `i` can be counted with the formula `a - b`.

For example, when `i` is at the position 5, we have the following situation:

Index	0	1	2	3	4	5	6	7
Number	1	2	3	3	2	2	4	2
	`b`			`a`		`i`		

Here `a` points to the position 3 containing the value 3, and `b` points to the position `0` containing the value `1`. The count of sublists is obtained as 3 – 0 = 3.

The variables of `a` and `b` must be updated every time when the value at the position `i` is different from the value at the position `i - 1`. There are two cases to consider:

1. If the value at `i` is different from the value at `a`, `b` moves to the position `a` and `a` moves to the position `i - 1`.
2. If the value at `i` is equal to the value at `a`, `a` again moves to `i - 1` but `b` does not move.

Let us consider what happens next in our example. When `i` moves from the position 5 to the position 6, the values at `i` and `a` are different and we have the case 1. Thus `a` moves to the position 6 – 1 = 5 and `b` moves to the position 3:

Index	0	1	2	3	4	5	6	7
Number	1	2	3	3	2	2	4	2
				`b`		`a`	`i`	

When `i` moves from the position 6 to the position 7, the values at `i` and `a` are equal and we have the case 2. Thus `a` moves to the position 7 – 1 = 6 and `b` does not move:

Index	0	1	2	3	4	5	6	7
Number	1	2	3	3	2	2	4	2
				`b`			`a`	`i`

This idea leads to an efficient algorithm that can be implemented as follows:

```python
def count_lists(numbers):
    n = len(numbers)
    a = b = -1
    result = 0
    for i in range(1, n):
        if numbers[i] != numbers[i - 1]:
            if numbers[i] != numbers[a]:
                b = a
            a = i - 1
        result += a - b
    return result
```

Initially, both `a` and `b` have the value `-1`, which indicates that they are not yet pointing to any list position. It is straightforward to verify that the algorithm computes the correct sublist counts in the beginning while `a` or `b` still has the value `-1`.