Codility_

Chapter 1

Iterations

In programming, iterating means repeating some part of your program. This lesson presents basic programming constructions that allow iterations to be performed: "for" and "while" loops.

1.1. For loops

If you want to repeat some operations a given number of times, or repeat them for each element in some collection, a "for" loop is the right tool to use. Its syntax is as follows:

1.1: For loop syntax

```
for some_variable in range_of_values:
    loop_body
```

The for loop repeats `loop_body` for each value in turn from the `range_of_values`, with the current value assigned to `some_variable`. In its simplest form, the range of values can be a range of integers, denoted by: `range(lowest, highest + 1)`. For example, the following loop prints every integer from 0 to 99:

```
for i in range(0, 100):
    print i
```

Looping over a range of integers starting from 0 is a very common operation. (This is mainly because arrays and Python lists are indexed by integers starting from 0; see Chapter 2 Arrays for more details.) When specifying the range of integers, if the starting value equals zero then you can simply skip it. For example, the following loop produces exactly the same result as the previous one:

```
for i in range(100):
    print i
```

Example: We are given some positive integer n. Let's compute the factorial of n and assign it to the variable `factorial`. The factorial of n is $n! = 1 \cdot 2 \cdot \ldots \cdot n$. We can obtain it by starting with 1 and multiplying it by all the integers from 1 to n.

```
factorial = 1
for i in range (1, n + 1):
    factorial *= i
```

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces. The triangle should consist of n rows, where n is a given positive integer, and consecutive rows should contain 1, 2, ..., n asterisks. For example, for $n = 4$ the triangle should appear as follows:

```
*
* *
* * *
* * * *
```

We need to use two loops, one inside the other: the outer loop should print one row in each step and the inner loop should print one asterisk in each step[2].

```
1 for i in range(1, n + 1):
2     for j in range(i):
3         print '*',
4     print
```

The `range` function can also accept one more argument specifying the step with which the iterated values progress. More formally, `range(start, stop, step)` is a sequence of values beginning with `start`, whose every consecutive value is increased by `step`, and that contains only values smaller than `stop` (for positive `step`; or greater than `stop` for negative `step`). For example, `range(10, 0, -1)` represents sequence 10, 9, 8, ..., 1. Note that we cannot omit `start` when we specify `step`.

Example: Let's print a triangle made of asterisks ('$\star$') separated by spaces and consisting of n rows again, but this time upside down, and make it symmetrical. Consecutive rows should contain $2n - 1$, $2n - 3$, ..., 3, 1 asterisks and should be indented by 0, 2, 4, ..., $2(n - 1)$ spaces. For example, for $n = 4$ the triangle should appear as follows:

```
* * * * * * *
  * * * * *
    * * *
      *
```

The triangle should have n rows, where n is some given positive integer.

This time we will use three loops: one outer and two inner loops. The outer loop in each step prints one row of the triangle. The first inner loop is responsible for printing the indentations, and the second for printing the asterisks.

```
1 for i in range(n, 0, -1):
2     for j in range(n - i):
3         print ' ',
4     for j in range(2 * i - 1):
5         print '*',
6     print
```

1.2. While loops

The number of steps in a for loop, and the values over which we loop, are fixed before the loop starts. What if the number of steps is not known in advance, or the values over which

[2]There is a clever idiom in Python denoting a string repeated a number of times. For example, `'*' * n` denotes a string comprising n asterisks. But to make our examples more instructive, we will not use this idiom here. Instead we will use the `print` statement ended with a comma. It doesn't print the newline character, but follows the output with a single space.

we loop are generated one by one, and are thus not known in advance either? In such a case, we have to use a different kind of loop, called a "while" loop. The syntax of the while loop is as follows:

1.2: While loop syntax

```
while some_condition:
    loop_body
```

Before each step of the loop, `some_condition` is computed. As long as its value is true[3], the body of the loop is executed. Once it becomes false, we exit the loop without executing `loop_body`.

Example: Given a positive integer n, how can we count the number of digits in its decimal representation? One way to do it is convert the integer into a string and count the characters. Here, though, we will use only arithmetical operations instead. We can simply keep dividing the number by ten and count how many steps are needed to obtain 0.

```
result = 0
while n > 0:
    n = n // 10
    result += 1
```

Example: The Fibonacci numbers[4] form a sequence of integers defined recursively in the following way. The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. The first few elements in this sequence are: 0, 1, 1, 2, 3, 5, 8, 13. Let's write a program that prints all the Fibonacci numbers, not exceeding a given integer n.

We can keep generating and printing consecutive Fibonacci numbers until we exceed n. In each step it's enough to store only two consecutive Fibonacci numbers.

```
a = 0
b = 1
while a <= n:
    print a
    c = a + b
    a = b
    b = c
```

1.3. Looping over collections of values[5]

We have seen how to loop over integers. Is it possible to loop over values of other types? Yes: using a "for" loop, we can loop over values stored in virtually any kind of container. The `range` function constructs a list containing all the values over which we should loop. However, we can pass a list constructed in any other way.

[3]Note that the condition can yield any value, not only `True` or `False`. A number of values other than `False` are interpreted as false, e.g. `None`, `0`, `[]` (empty list) and `''` (empty string).

[4]You can read more about the Fibonacci numbers in Chapter 13.

[5]If you are not familiar with various built-in data structures such as lists, sets and dictionaries, you can safely skip this section. Looping over lists of values will be explained in Chapter 2.

Example: The following program:

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday', 'Sunday']
for day in days:
    print day
```

prints all the days of the week, one per line.

When we use the `range` function, we first build a list of all the integers over which we will loop; then we start looping. This is memory-consuming when looping over a long sequence. In such cases, it's advisable to use – instead of `range` – an equivalent function called `xrange`. This returns exactly the same sequence of integers, but instead of storing them in a list, it returns an object that generates them on the fly.

If you loop over a set of values, the body of the loop is executed exactly once for every value in the set; however, the order in which the values are processed is arbitrary.

Example: If we modify the above program slightly, as follows:

```
days = set(['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday'])
for day in days:
    print day
```

we might get the days output in some strange order, e.g.:

```
Monday
Tuesday
Friday
Wednesday
Thursday
Sunday
Saturday
```

Looping over a dictionary means looping over its set of keys. Again, the order in which the keys are processed is arbitrary.

Example: The following program:

```
days = {'mon': 'Monday', 'tue': 'Tuesday', 'wed': 'Wednesday',
        'thu': 'Thursday', 'fri': 'Friday', 'sat': 'Saturday',
        'sun': 'Sunday'}
for day in days:
    print day, 'stands for', days[day]
```

might output e.g.:

```
wed stands for Wednesday
sun stands for Sunday
fri stands for Friday
tue stands for Tuesday
mon stands for Monday
thu stands for Thursday
sat stands for Saturday
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 2

Arrays

Array is a data-structure that can be used to store many items in one place. Imagine that we have a list of items; for example, a shopping list. We don't keep all the products on separate pages; we simply list them all together on a single page. Such a page is conceptually similar to an array. Similarly, if we plan to record air temperatures over the next 365 days, we would not create lots of individual variables, but would instead store all the data in just one array.

2.1. Creating an array

We want to create a shopping list containing three products. Such a list might be created as follows:

```
shopping = ['bread', 'butter', 'cheese']
```

(that is, `shopping` is the name of the array and every product within it is separated by a comma). Each item in the array is called an element. Arrays can store any number of elements (assuming that there is enough memory). Note that a list can be also empty:

```
shopping = []
```

If planning to record air temperatures over the next 365 days, we can create in advance a place to store the data. The array can be created in the following way:

```
temperatures = [0] * 365
```

(that is, we are creating an array containing 365 zeros).

2.2. Accessing array values

Arrays provide easy access to all elements. Within the array, every element is assigned a number called an index. Index numbers are consecutive integers starting from 0. For example, in the array `shopping = ['bread', 'butter', 'cheese']`, 'bread' is at index 0, 'butter' is at index 1 and 'cheese' is at index 2. If we want to check what value is located at some index (for example, at index 1), we can access it by specifing the index in square brackets, e.g. `shopping[1]`.

2.3. Modifying array values

We can change array elements as if they were separate variables, that is each array element can be assigned a new value independently. For example, let's say we want to record that on the 42nd day of measurement, the air temperature was 25 degrees. This can be done with a single assignment:

```
temperatures[42] = 25
```

If there was one more product to add to our shopping list, it could be appended as follows:

```
shopping += ['eggs']
```

The index for that element will be the next integer after the last (in this case, 3).

2.4. Iterating over an array

Often we need to iterate over all the elements of an array; perhaps to count the number of specified items, for example. Knowing that the array contains of N elements, we can iterate over consecutive integers from index 0 to index $N-1$ and check every such index. The length of an array can be found using the `len()` function. For example, counting the number of items in shopping list can be done quickly as follows:

```
N = len(shopping)
```

Let's write a function that counts the number of days with negative air temperature.

2.1: Negative air temperature.

```
def negative(temperatures):
    N = len(temperatures)
    days = 0
    for i in xrange(N):
        if temperatures[i] < 0:
            days += 1
    return days
```

Instead of iterating over indexes, we can iterate over the elements of the array. To do this, we can simply write:

```
for item in array:
    ...
```

For example, the above solution can be simplified as follows:

2.2: Negative air temperature — simplified.

```
def negative(temperatures):
    days = 0
    for t in temperatures:
        if t < 0:
            days += 1
    return days
```

In the above solution, for every temperature, we increase the number of days with a negative temperature if the number is lower than zero.

2.5. Basic array operations

There are a few basic operations on arrays that are very useful. Apart from the length operation:

```
len([1, 2, 3]) == 3
```

and the repetition:

```
['Hello'] * 3 == ['Hello', 'Hello', 'Hello']
```

which we have already seen, there is also concatenation:

```
[1, 2, 3] + [4, 5, 6] == [1, 2, 3, 4, 5, 6]
```

which merges two lists, and the membership operation:

```
'butter' in ['bread', 'butter', 'cheese'] == True
```

which checks for the presence of a particular item in the array.

2.6. Exercise

Problem: Given array A consisting of N integers, return the reversed array.

Solution: We can iterate over the first half of the array and exchange the elements with those in the second part of the array.

2.3: Reversing an array.

```
1 def reverse(A):
2     N = len(A)
3     for i in xrange(N // 2):
4         k = N - i - 1
5         A[i], A[k] = A[k], A[i]
6     return A
```

Python is a very rich language and provides many built-in functions and methods. It turns out, that there is already a built-in method `reverse`, that solves this exercise. Using such a method, array A can be reversed simply by:

```
1 A.reverse()
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 3

Time complexity

Use of time complexity makes it easy to estimate the running time of a program. Performing an accurate calculation of a program's operation time is a very labour-intensive process (it depends on the compiler and the type of computer or speed of the processor). Therefore, we will not make an accurate measurement; just a measurement of a certain order of magnitude.

Complexity can be viewed as the maximum number of primitive operations that a program may execute. Regular operations are single additions, multiplications, assignments etc. We may leave some operations uncounted and concentrate on those that are performed the largest number of times. Such operations are referred to as *dominant*.

The number of dominant operations depends on the specific input data. We usually want to know how the performance time depends on a particular aspect of the data. This is most frequently the data size, but it can also be the size of a square matrix or the value of some input variable.

3.1: Which is the dominant operation?

```
1 def dominant(n):
2     result = 0
3     for i in xrange(n):
4         result += 1
5     return result
```

The operation in line 4 is dominant and will be executed n times. The complexity is described in Big-O notation: in this case $O(n)$ — *linear* complexity.

The complexity specifies the order of magnitude within which the program will perform its operations. More precisely, in the case of $O(n)$, the program may perform $c \cdot n$ operations, where c is a constant; however, it may not perform n^2 operations, since this involves a different order of magnitude of data. In other words, when calculating the complexity we omit constants: i.e. regardless of whether the loop is executed $20 \cdot n$ times or $\frac{n}{5}$ times, we still have a complexity of $O(n)$, even though the running time of the program may vary. When analyzing the complexity we must look for specific, worst-case examples of data that the program will take a long time to process.

3.1. Comparison of different time complexities

Let's compare some basic time complexities.

3.2: Constant time — $O(1)$.

```
def constant(n):
    result = n * n
    return result
```

There is always a fixed number of operations.

3.3: Logarithmic time — $O(\log n)$.

```
def logarithmic(n):
    result = 0
    while n > 1:
        n //= 2
        result += 1
    return result
```

The value of n is halved on each iteration of the loop. If $n = 2^x$ then $\log n = x$. How long would the program below take to execute, depending on the input data?

3.4: Linear time — $O(n)$.

```
def linear(n, A):
    for i in xrange(n):
        if A[i] == 0:
            return 0
    return 1
```

Let's note that if the first value of array A is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array A does not contain any 0.

3.5: Quadratic time — $O(n^2)$.

```
def quadratic(n):
    result = 0
    for i in xrange(n):
        for j in xrange(i, n):
            result += 1
    return result
```

The result of the function equals $\frac{1}{2} \cdot (n \cdot (n+1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ($\frac{1}{2} \cdot n$ in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

3.6: Linear time — $O(n+m)$.

```
def linear2(n, m):
    result = 0
    for i in xrange(n):
        result += i
    for j in xrange(m):
        result += j
    return result
```

Exponential and factorial time

It is worth knowing that there are other types of time complexity such as factorial time $O(n!)$ and exponential time $O(2^n)$. Algorithms with such complexities can solve problems only for very small values of n, because they would take too long to execute for large values of n.

3.2. Time limit

Nowadays, an average computer can perform 10^8 operations in less than a second. Sometimes we have the information we need about the expected time complexity (for example, Codility specifies the expected time complexity), but sometimes we do not.

The time limit set for online tests is usually from 1 to 10 seconds. We can therefore estimate the expected complexity. During contests, we are often given a limit on the size of data, and therefore we can guess the time complexity within which the task should be solved. This is usually a great convenience because we can look for a solution that works in a specific complexity instead of worrying about a faster solution. For example, if:

- $n \leqslant 1\,000\,000$, the expected time complexity is $O(n)$ or $O(n \log n)$,
- $n \leqslant 10\,000$, the expected time complexity is $O(n^2)$,
- $n \leqslant 500$, the expected time complexity is $O(n^3)$.

Of course, these limits are not precise. They are just approximations, and will vary depending on the specific task.

3.3. Space complexity

Memory limits provide information about the expected space complexity. You can estimate the number of variables that you can declare in your programs.

In short, if you have constant numbers of variables, you also have constant space complexity: in Big-O notation this is $O(1)$. If you need to declare an array with n elements, you have linear space complexity — $O(n)$.

More specifically, space complexity is the amount of memory needed to perform the computation. It includes all the variables, both global and local, dynamic pointer data-structures and, in the case of recursion, the contents of the stack. Depending on the convention, input data may also be included. Space complexity is more tricky to calculate than time complexity because not all of these variables and data-structures may be needed at the same time. Global variables exist and occupy memory all the time; local variables (and additional information kept on the stack) will exist only during invocation of the function.

3.4. Exercise

Problem: You are given an integer n. Count the total of $1 + 2 + \ldots + n$.

Solution: The task can be solved in several ways. Some person, who knows nothing about time complexity, may implement an algorithm in which the result is incremented by 1:

3.7: Slow solution — time complexity $O(n^2)$.

```
def slow_solution(n):
    result = 0
    for i in xrange(n):
        for j in xrange(i + 1):
            result += 1
    return result
```

Another person may increment the result respectively by $1, 2, \ldots, n$. This algorithm is much faster:

3.8: Fast solution — time complexity $O(n)$.

```
def fast_solution(n):
    result = 0
    for i in xrange(n):
        result += (i + 1)
    return result
```

But the third person's solution is even quicker. Let us write the sequence $1, 2, \ldots, n$ and repeat the same sequence underneath it, but in reverse order. Then just add the numbers from the same columns:

1	2	3	$\ldots$	$n-1$	n
n	$n-1$	$n-2$	$\ldots$	2	1
$n+1$	$n+1$	$n+1$	$\ldots$	$n+1$	$n+1$

The result in each column is $n+1$, so we can easily count the final result:

3.9: Model solution — time complexity $O(1)$.

```
def model_solution(n):
    result = n * (n + 1) // 2
    return result
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility

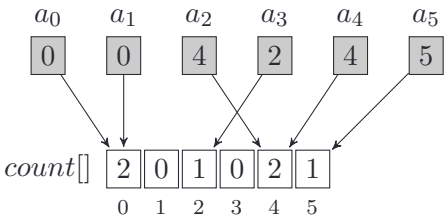
Chapter 4

Counting elements

A numerical sequence can be stored in an array in various ways. In the standard approach, the consecutive numbers $a_0, a_1, \ldots, a_{n-1}$ are usually put into the corresponding consecutive indices of the array:

$$A[0] = a_0 \quad A[1] = a_1 \quad \ldots \quad A[n-1] = a_{n-1}$$

We can also store the data in a slightly different way, by making an array of counters. Each number may be counted in the array by using an index that corresponds to the value of the given number.



Notice that we do not place elements directly into a cell; rather, we simply count their occurrences. It is important that the array in which we count elements is sufficiently large. If we know that all the elements are in the set $\{0, 1, \ldots, m\}$, then the array used for counting should be of size $m + 1$.

4.1: Counting elements — $O(n + m)$.

```
1  def counting(A, m):
2      n = len(A)
3      count = [0] * (m + 1)
4      for k in xrange(n):
5          count[A[k]] += 1
6      return count
```

The limitation here may be available memory. Usually, we are not able to create arrays of 10^9 integers, because this would require more than one gigabyte of available memory.

Counting the number of negative integers can be done in two ways. The first method is to add some big number to each value: so that, all values would be greater than or equal to zero. That is, we shift the representation of zero by some arbitrary amount to accommodate all the negative numbers we need. In the second method, we simply create a second array for counting negative numbers.

4.1. Exercise

Problem: You are given an integer m ($1 \leqslant m \leqslant 1\,000\,000$) and two non-empty, zero-indexed arrays A and B of n integers, $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$ respectively ($0 \leqslant a_i, b_i \leqslant m$).

The goal is to check whether there is a swap operation which can be performed on these arrays in such a way that the sum of elements in array A equals the sum of elements in array B after the swap. By swap operation we mean picking one element from array A and one element from array B and exchanging them.

Solution $O(n^2)$: The simplest method is to swap every pair of elements and calculate the totals. Using that approach gives us $O(n^3)$ time complexity. A better approach is to calculate the sums of elements at the beginning, and check only how the totals change during the swap operation.

4.2: Swap the elements — $O(n^2)$.

```
def slow_solution(A, B, m):
    n = len(A)
    sum_a = sum(A)
    sum_b = sum(B)
    for i in xrange(n):
        for j in xrange(n):
            change = B[j] - A[i]
            sum_a += change
            sum_b -= change
            if sum_a == sum_b:
                return True
            sum_a -= change
            sum_b += change
    return False
```

Solution $O(n + m)$: The best approach is to count the elements of array A and calculate the difference d between the sums of the elements of array A and B.

For every element of array B, we assume that we will swap it with some element from array A. The difference d tells us the value from array A that we are interested in swapping, because only one value will cause the two totals to be equal. The occurrence of this value can be found in constant time from the array used for counting.

4.3: Swap the elements — $O(n + m)$.

```
def fast_solution(A, B, m):
    n = len(A)
    sum_a = sum(A)
    sum_b = sum(B)
    d = sum_b - sum_a
    if d % 2 == 1:
        return False
    d //= 2
    count = counting(A, m)
    for i in xrange(n):
        if 0 <= B[i] - d and B[i] - d <= m and count[B[i] - d] > 0:
            return True
    return False
```

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 5

Prefix sums

There is a simple yet powerful technique that allows for the fast calculation of sums of elements in given slice (contiguous segments of array). Its main idea uses prefix sums which are defined as the consecutive totals of the first $0, 1, 2, \ldots, n$ elements of an array.

	a_0	a_1	a_2	...	a_{n-1}
$p_0 = 0$	$p_1 = a_0$	$p_2 = a_0 + a_1$	$p_3 = a_0 + a_1 + a_2$	...	$p_n = a_0 + a_1 + \ldots + a_{n-1}$

We can easily calculate the prefix sums in $O(n)$ time complexity. Notice that the total p_k equals $p_{k-1} + a_{k-1}$, so each consecutive value can be calculated in a constant time.

5.1: Counting prefix sums — $O(n)$.

```
def prefix_sums(A):
    n = len(A)
    P = [0] * (n + 1)
    for k in xrange(1, n + 1):
        P[k] = P[k - 1] + A[k - 1]
    return P
```

Similarly, we can calculate suffix sums, which are the totals of the k last values. Using prefix (or suffix) sums allows us to calculate the total of any slice of the array very quickly. For example, assume that you are asked about the totals of m slices $[x..y]$ such that $0 \leqslant x \leqslant y < n$, where the total is the sum $a_x + a_{x+1} + \ldots + a_{y-1} + a_y$.

The simplest approach is to iterate through the whole array for each result separately; however, that requires $O(n \cdot m)$ time. The better approach is to use prefix sums. If we calculate the prefix sums then we can answer each question directly in constant time. Let's subtract p_x from the value p_{y+1}.

p_{y+1}	a_0	a_1	...	a_{x-1}	a_x	a_{x+1}	...	a_{y-1}	a_y
p_x	a_0	a_1	...	a_{x-1}					
$p_{y+1} - p_x$					a_x	a_{x+1}	...	a_{y-1}	a_y

5.2: Total of one slice — $O(1)$.

```
def count_total(P, x, y):
    return P[y + 1] - P[x]
```

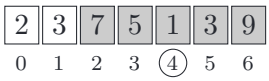
We have calculated the total of $a_x + a_{x+1} + \ldots + a_{y-1} + a_y$ in $O(1)$ time. Using this approach, the total time complexity is $O(n + m)$.

5.1. Exercise

Problem: You are given a non-empty, zero-indexed array A of n ($1 \leqslant n \leqslant 100\,000$) integers $a_0, a_1, \ldots, a_{n-1}$ ($0 \leqslant a_i \leqslant 1\,000$). This array represents number of mushrooms growing on the consecutive spots along a road. You are also given integers k and m ($0 \leqslant k, m < n$).

A mushroom picker is at spot number k on the road and should perform m moves. In one move she moves to an adjacent spot. She collects all the mushrooms growing on spots she visits. The goal is to calculate the maximum number of mushrooms that the mushroom picker can collect in m moves.

For example, consider array A such that:



The mushroom picker starts at spot $k = 4$ and should perform $m = 6$ moves. She might move to spots $3, 2, 3, 4, 5, 6$ and thereby collect $1 + 5 + 7 + 3 + 9 = 25$ mushrooms. This is the maximal number of mushrooms she can collect.

Solution $O(m^2)$: Note that the best strategy is to move in one direction optionally followed by some moves in the opposite direction. In other words, the mushroom picker should not change direction more than once. With this observation we can find the simplest solution. Make the first $p = 0, 1, 2, \ldots, m$ moves in one direction, then the next $m - p$ moves in the opposite direction. This is just a simple simulation of the moves of the mushroom picker which requires $O(m^2)$ time.

Solution $O(n+m)$: A better approach is to use prefix sums. If we make p moves in one direction, we can calculate the maximal opposite location of the mushroom picker. The mushroom picker collects all mushrooms between these extremes. We can calculate the total number of collected mushrooms in constant time by using prefix sums.

5.3: Mushroom picker — $O(n + m)$

```
def mushrooms(A, k, m):
    n = len(A)
    result = 0
    pref = prefix_sums(A)
    for p in xrange(min(m, k) + 1):
        left_pos = k - p
        right_pos = min(n - 1, max(k, k + m - 2 * p))
        result = max(result, count_total(pref, left_pos, right_pos))
    for p in xrange(min(m + 1, n - k)):
        right_pos = k + p
        left_pos = max(0, min(k, k - (m - 2 * p)))
        result = max(result, count_total(pref, left_pos, right_pos))
    return result
```

The total time complexity of such a solution is $O(n + m)$.

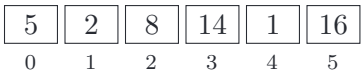
Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

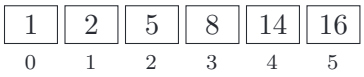
Chapter 6

Sorting

Sorting is the process of arranging data in a certain order. Usually, we sort by the value of the elements. We can sort numbers, words, pairs, etc. For example, we can sort students by their height, and we can sort cities in alphabetical order or by their numbers of citizens. The most-used orders are numerical order and alphabetical order. Let's consider the simplest set, an array consisting of integers:



We want to sort this array into numerical order to obtain the following array:



There are many sorting algorithms, and they differ considerably in terms of their time complexity and use of memory. Here we describe some of them.

6.1. Selection sort

The idea: Find the minimal element and swap it with the first element of an array. Next, just sort the rest of the array, without the first element, in the same way.

Notice that after k iterations (repetition of everything inside the loop) the first k elements will be sorted in the right order (this type of a property is called the loop *invariant*).

6.1: Selection sort — $O(n^2)$.

```
1  def selectionSort(A):
2      n = len(A)
3      for k in xrange(n):
4          minimal = k
5          for j in xrange(k + 1, n):
6              if A[j] < A[minimal]:
7                  minimal = j
8          A[k], A[minimal] = A[minimal], A[k] # swap A[k] and A[minimal]
9      return A
```

The time complexity is quadratic.

6.2. Counting sort

The idea: First, count the elements in the array of counters (see chapter 2). Next, just iterate through the array of counters in increasing order.

Notice that we have to know the range of the sorted values. If all the elements are in the set $\{0, 1, \ldots, k\}$, then the array used for counting should be of size $k+1$. The limitation here may be available memory.

6.2: Counting sort — $O(n+k)$

```
1  def countingSort(A, k):
2      n = len(A)
3      count = [0] * (k + 1)
4      for i in xrange(n):
5          count[A[i]] += 1
6      p = 0
7      for i in xrange(k + 1):
8          for j in xrange(count[i]):
9              A[p] = i
10             p += 1
11     return A
```

The time complexity here is $O(n+k)$. We need additional memory $O(k)$ to count all the elements. At first sight, the time complexity of the above implementation may appear greater. However, all the operations in lines 9 and 10 are performed not more than $O(n)$ times.

6.3. Merge sort

The idea: Divide the unsorted array into two halves, sort each half separately and then just merge them. After the split, each part is halved again.

We repeat this algorithm until we end up with individual elements, which are sorted by definition. The merging of two sorted arrays consisting of k elements takes $O(k)$ time; just repeatedly choose the lower of the first elements of the two merged parts.

The length of the array is halved on each iteration. In this way, we get consecutive levels with $1, 2, 4, 8, \ldots$ slices. For each level, the merging of the all consecutive pairs of slices requires $O(n)$ time. The number of levels is $O(\log n)$, so the total time complexity is $O(n \log n)$ (read more at http://en.wikipedia.org/wiki/Merge_sort).

6.4. Sorting functions

If the range of sorted values is unknown then there are algorithms which sort all the values in $O(n \log n)$ time. A big advantage of many programming languages are their built-in sorting functions. If you want to sort a list in Python, you can do it with only one line of code.

6.3: Built-in sort — $O(n \log n)$

```
1  A.sort()
```

The time complexity of this sorting function is $O(n \log n)$. Generally, sorting algorithms use very interesting ideas which can be used in other problems. It is worth knowing how they work, and it is also worth implementing them yourself at least once. In the future you can use the built-in sorting functions, because their implementations will be faster and they make your code shorter and more readable.

6.5. Exercise

Problem: You are given a zero-indexed array A consisting of $n > 0$ integers; you must return the number of unique values in array A.

Solution $O(n \log n)$**:** First, sort array A; similar values will then be next to each other. Finally, just count the number of distinct pairs in adjacent cells.

6.4: The number of distinct values — $O(n \log n)$.

```
1 def distinct(A):
2     n = len(A)
3     A.sort()
4     result = 1
5     for k in xrange(1, n):
6         if A[k] != A[k - 1]:
7             result += 1
8     return result
```

The time complexity is $O(n \log n)$, in view of the sorting time.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

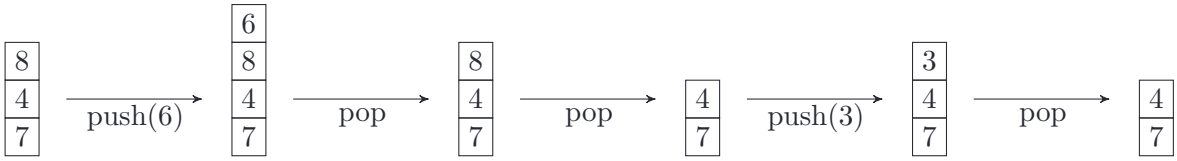
Chapter 7

Stacks and Queues

Here are described two structures used for storage of elements. The structures will provide two operations: *push* (inserting the new element to the structure) and *pop* (removing some element from the structure).

7.1. Stack

The stack is a basic data structure in which the insertion of new elements takes place at the top and deletion of elements also takes place from the top. The idea of the stack can be illustrated by plates stacked on top of one another. Each new plate is placed on top of the stack of plates (operation push), and plates can only be taken off the top of the stack (operation pop).



The stack can be represented by an array for storing the elements. Apart of the array, we should also remember the size of the stack and we must be sure to declare sufficient space for the array (in the following implementation we can store N elements).

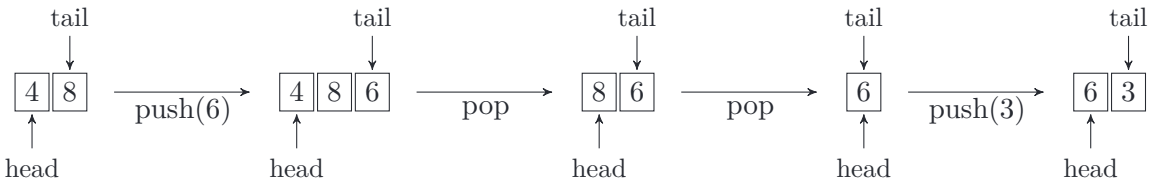
7.1: Push / pop function — $O(1)$.

```
1  stack = [0] * N
2  size = 0
3  def push(x):
4      global size
5      stack[size] = x
6      size += 1
7  def pop():
8      global size
9      size -= 1
10     return stack[size]
```

The *push* function adds an element to the stack. The *pop* function removes and returns the most recently pushed element from the stack. We shouldn't perform a pop operation on an empty stack.

7.2. Queue

The queue is a basic data structure in which new elements are inserted at the back but old elements are removed from the front. The idea of the queue can be illustrated by a line of customers in a grocery store. New people join the back of the queue and the next person to be served is the first one in the line.



The queue can be represented by an array for storing the elements. Apart of the array, we should also remember the front (head) and back (tail) of the queue. We must be sure to declare sufficient space for the array (in the following implementation we can store $N - 1$ elements).

7.2: Push / pop / size / empty function — $O(1)$.

```
queue = [0] * N
head, tail = 0, 0
def push(x):
    global tail
    tail = (tail + 1) % N
    queue[tail] = x
def pop():
    global head
    head = (head + 1) % N
    return queue[head]
def size():
    return (tail - head + N) % N
def empty():
    return head == tail
```

Notice that in the above implementation we used cyclic buffer (you can read about it more at http://en.wikipedia.org/wiki/Circular_buffer).

The *push* function adds an element to the queue. The *pop* function removes and returns an element from the front of the queue (we shouldn't perform a pop operation on an empty queue). The *empty* function check whether the queue is empty and the size function returns the number of elements in the queue.

7.3. Exercises

Problem: You are given a zero-indexed array A consisting of n integers: $a_0, a_1, \ldots, a_{n-1}$. Array A represents a scenario in a grocery store, and contains only 0s and/or 1s:

- 0 represents the action of a new person joining the line in the grocery store,
- 1 represents the action of the person at the front of the queue being served and leaving the line.

The goal is to count the minimum number of people who should have been in the line before the above scenario, so that the scenario is possible (it is not possible to serve a person if the line is empty).

Solution $O(n)$: We should remember the size of the queue and carry out a simulation of people arriving at and leaving the grocery store. If the size of the queue becomes a negative number then that sets the lower limit for the number of people who had to stand in the line previously. We should find the smallest negative number to determine the size of the queue during the whole simulation.

7.3: Model solution — $O(n)$.

```
def grocery_store(A):
    n = len(A)
    size, result = 0, 0
    for i in xrange(n):
        if A[i] == 0:
            size += 1
        else:
            size -= 1
            result = max(result, -size)
    return result
```

The total time complexity of the above algorithm is $O(n)$. The space complexity is $O(1)$ because we don't store people in the array, but only remember the size of the queue.

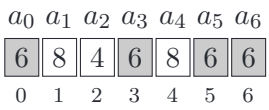
Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 8

Leader

Let us consider a sequence $a_0, a_1, \ldots, a_{n-1}$. The leader of this sequence is the element whose value occurs more than $\frac{n}{2}$ times.



In the picture the leader is highlighted in gray. Notice that the sequence can have at most one leader. If there were two leaders then their total occurrences would be more than $2 \cdot \frac{n}{2} = n$, but we only have n elements.

The leader may be found in many ways. We describe some methods here, starting with trivial, slow ideas and ending with very creative, fast algorithms. The task is to find the value of the leader of the sequence $a_0, a_1, \ldots, a_{n-1}$, such that $0 \leqslant a_i \leqslant 10^9$. If there is no leader, the result should be `-1`.

8.1. Solution with $O(n^2)$ time complexity

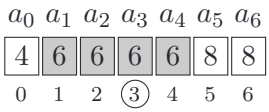
We count the occurrences of every element:

8.1: Leader — $O(n^2)$.

```
def slowLeader(A):
    n = len(A)
    leader = -1
    for k in xrange(n):
        candidate = A[k]
        count = 0
        for i in xrange(n):
            if (A[i] == candidate):
                count += 1
        if (count > n // 2):
            leader = candidate
    return leader
```

8.2. Solution with $O(n \log n)$ time complexity

If the sequence is presented in non-decreasing order, then identical values are adjacent to each other.



Having sorted the sequence, we can easily count slices of the same values and find the leader in a smarter way. Notice that if the leader occurs somewhere in our sequence, then it must occur at index $\frac{n}{2}$ (the central element). This is because, given that the leader occurs in more than half the total values in the sequence, there are more leader values than will fit on either side of the central element in the sequence.

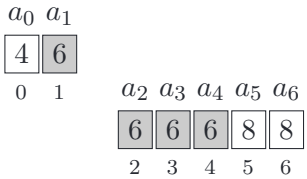
8.2: Leader — $O(n \log n)$.

```
1  def fastLeader(A):
2      n = len(A)
3      leader = -1
4      A.sort()
5      candidate = A[n // 2]
6      count = 0
7      for i in xrange(n):
8          if (A[i] == candidate):
9              count += 1
10     if (count > n // 2):
11         leader = candidate
12     return leader
```

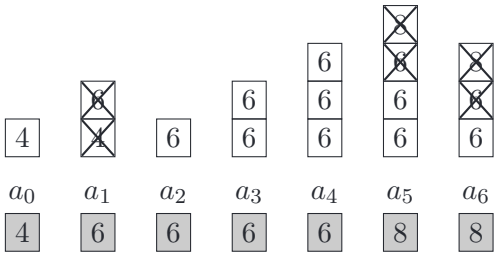
The time complexity of the above algorithm is $O(n \log n)$ due to the sorting time.

8.3. Solution with $O(n)$ time complexity

Notice that if the sequence $a_0, a_1, \ldots, a_{n-1}$ contains a leader, then after removing a pair of elements of different values, the remaining sequence still has the same leader. Indeed, if we remove two different elements then only one of them could be the leader. The leader in the new sequence occurs more than $\frac{n}{2} - 1 = \frac{n-2}{2}$ times. Consequently, it is still the leader of the new sequence of $n - 2$ elements.



Removing pairs of different elements is not trivial. Let's create an empty stack onto which we will be pushing consecutive elements. After each such operation we check whether the two elements at the top of the stack are different. If they are, we remove them from the stack. This is equivalent to removing a pair of different elements from the sequence (in the picture below, different elements being removed are highlighted in gray).



In fact, we don't need to remember all the elements from the stack, because all the values below the top are always equal. It is sufficient to remember only the values of elements and the size of the stack.

8.3: Leader — $O(n)$.

```
def goldenLeader(A):
    n = len(A)
    size = 0
    for k in xrange(n):
        if (size == 0):
            size += 1
            value = A[k]
        else:
            if (value != A[k]):
                size -= 1
            else:
                size += 1
    candidate = -1
    if (size > 0):
        candidate = value
    leader = -1
    count = 0
    for k in xrange(n):
        if (A[k] == candidate):
            count += 1
    if (count > n // 2):
        leader = candidate
    return leader
```

At the beginning we notice that if the sequence contains a leader, then after the removal of different elements the leader will not have changed. After removing all pairs of different elements, we end up with a sequence containing all the same values. This value is not necessarily the leader; it is only a candidate for the leader. Finally, we should iterate through all the elements and count the occurrences of the candidate; if it is greater than $\frac{n}{2}$ then we have found the leader; otherwise the sequence does not contain a leader.

The time complexity of this algorithm is $O(n)$ because every element is considered only once. The final counting of occurrences of the candidate value also works in $O(n)$ time.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 9

Maximum slice problem

Let's define a problem relating to maximum slices. You are given a sequence of n integers $a_0, a_1, \ldots, a_{n-1}$ and the task is to find the slice with the largest sum. More precisely, we are looking for two indices p, q such that the total $a_p + a_{p+1} + \ldots + a_q$ is maximal. We assume that the slice can be empty and its sum equals 0.

a_0	a_1	a_2	a_3	a_4	a_5	a_6
5	-7	3	5	-2	4	-1

In the picture, the slice with the largest sum is highlighted in gray. The sum of this slice equals 10 and there is no slice with a larger sum. Notice that the slice we are looking for may contain negative integers, as shown above.

9.1. Solution with $O(n^3)$ time complexity

The simplest approach is to analyze all the slices and choose the one with the largest sum.

9.1: Maximal slice — $O(n^3)$.

```
def slow_max_slice(A):
    n = len(A)
    result = 0
    for p in xrange(n):
        for q in xrange(p, n):
            sum = 0
            for i in xrange(p, q + 1):
                sum += A[i]
            result = max(result, sum)
    return result
```

Analyzing all possible slices requires $O(n^2)$ time complexity, and for each of them we compute the total in $O(n)$ time complexity. It is the most straightforward solution, however it is far from optimal.

9.2. Solution with $O(n^2)$ time complexity

We can easily improve our last solution. Notice that the prefix sum allows the sum of any slice to be computed in a constant time. With this approach, the time complexity of the whole algorithm reduces to $O(n^2)$. We assume that *pref* is an array of prefix sums ($pref_i = a_0 + a_1 + \ldots + a_{i-1}$).

9.2: Maximal slice — $O(n^2)$.

```
def quadratic_max_slice(A, pref):
    n = len(A), result = 0
    for p in xrange(n):
        for q in xrange(p, n):
            sum = pref[q + 1] - pref[p]
            result = max(result, sum)
    return result
```

We can also solve this problem without using prefix sums, within the same time complexity. Assume that we know the sum of slice (p, q), so $s = a_p + a_{p+1} + \ldots + a_q$. The sum of the slice with one more element $(p, q+1)$ equals $s + a_{q+1}$. Following this observation, there is no need to compute the sum each time from the beginning; we can use the previously calculated sum.

9.3: Maximal slice — $O(n^2)$.

```
def quadratic_max_slice(A):
    n = len(A), result = 0
    for p in xrange(n):
        sum = 0
        for q in xrange(p, n):
            sum += A[q]
            result = max(result, sum)
    return result
```

Still these solutions are not optimal.

9.3. Solution with $O(n)$ time complexity

This problem can be solved even faster. For each position, we compute the largest sum that ends in that position. If we assume that the maximum sum of a slice ending in position i equals *max_ending*, then the maximum slice ending in position $i+1$ equals $\max(0, max\_ending + a_{i+1})$.

9.4: Maximal slice — $O(n)$.

```
def golden_max_slice(A):
    max_ending = max_slice = 0
    for a in A:
        max_ending = max(0, max_ending + a)
        max_slice = max(max_slice, max_ending)
    return max_slice
```

This time, the fastest algorithm is the one with the simplest implementation, however it is conceptually more difficult. We have used here a very popular and important technique. Based on the solution for shorter sequences we can find the solution for longer sequences.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 10

Prime and composite numbers

People have been analyzing prime numbers since time immemorial, but still we continue to search for fast new algorithms that can check the primality of numbers. A prime number is a natural number greater than 1 that has exactly two divisors (1 and itself). A composite number has more than two divisors.

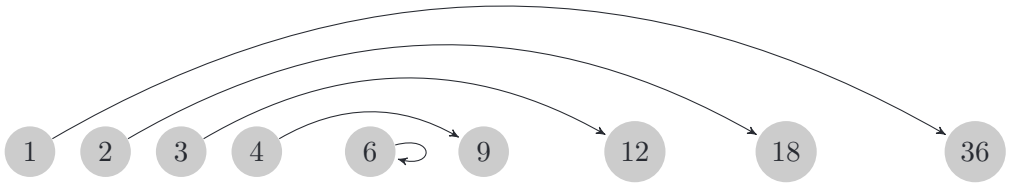
2	3	4	5	6	7	8	9	10	11	12	13	14

In the above picture the primes are highlighted in white and composite numbers are shown in gray.

10.1. Counting divisors

Let's count the number of divisors of n. The easiest approach is to iterate through all the numbers from 1 to n and check whether or not each one is a divisor. The time complexity of this solution is $O(n)$.

There is a simple way to improve the above solution. Based on one divisor, we can find the symmetric divisor. More precisely, if number a is a divisor of n, then $\frac{n}{a}$ is also a divisor. One of these two divisors is less than or equal to $\sqrt{n}$. (If that were not the case, n would be a product of two numbers greater than $\sqrt{n}$, which is impossible.)



Thus, iterating through all the numbers from 1 to $\sqrt{n}$ allows us to find all the divisors. If number n is of the form k^2, then the symmetric divisor of k is also k. This divisor should be counted just once.

10.1: Counting the number of divisors — $O(\sqrt{n})$.

```
1 def divisors(n):
2     i = 1
3     result = 0
```

```
4       while (i * i < n):
5           if (n % i == 0):
6               result += 2
7           i += 1
8       if (i * i == n):
9           result += 1
10      return result
```

10.2. Primality test

The primality test of n can be performed in an analogous way to counting the divisors. If we find a number between 2 and $n-1$ that divides n then n is a composite number. Otherwise, n is a prime number.

10.2: Primality test — $O(\sqrt{n})$.

```
1  def primality(n):
2      i = 2
3      while (i * i <= n):
4          if (n % i == 0):
5              return False
6          i += 1
7      return True
```

We assume that 1 is neither a prime nor a composite number, so the above algorithm works only for $n \geqslant 2$.

10.3. Exercises

Problem: Consider n coins aligned in a row. Each coin is showing heads at the beginning.

Then, n people turn over corresponding coins as follows. Person i reverses coins with numbers that are multiples of i. That is, person i flips coins $i, 2 \cdot i, 3 \cdot i, \ldots$ until no more appropriate coins remain. The goal is to count the number of coins showing tails. In the above example, the final configuration is:

Solution $O(n \log n)$: We can simulate the results of each person reversing coins.

10.3: Reversing coins — $O(n \log n)$.

```
1  def coins(n):
2      result = 0
3      coin = [0] * (n + 1)
4      for i in xrange(1, n + 1):
5          k = i
6          while (k <= n):
7              coin[k] = (coin[k] + 1) % 2
8              k += i
9          result += coin[i]
```

```
10      return result
```

The number of operation can be estimated by $\frac{n}{1}+\frac{n}{2}+\ldots+\frac{n}{n}$, what equals $n\cdot(\frac{1}{1}+\frac{1}{2}+\ldots+\frac{1}{n})$. The sums of multiplicative inverses (reciprocals) of the first n numbers are called harmonic numbers, which asymptotically equal $O(\log n)$. In summary, the total time complexity is $O(n \log n)$.

Solution $O(\log n)$: Notice that each coin will be turned over exactly as many times as the number of its divisors. The coins that are reversed an odd number of times show tails, meaning that it is sufficient to find the coins with an odd number of divisors.

We know that almost every number has a symmetric divisor (apart from divisors of the form $\sqrt{n}$). Thus, every number of the form k^2 has an odd number of divisors. There are exactly $\lfloor\sqrt{n}\rfloor$ such numbers between 1 and n. Finding the value of $\lfloor\sqrt{n}\rfloor$ takes logarithmic time (or constant time if we use operations on floating point numbers).

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

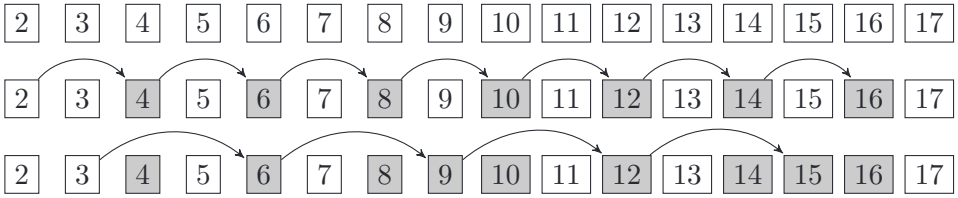
Codility_

Chapter 11

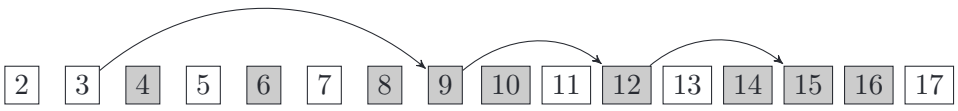
Sieve of Eratosthenes

The Sieve of Eratosthenes is a very simple and popular technique for finding all the prime numbers in the range from 2 to a given number n. The algorithm takes its name from the process of sieving—in a simple way we remove multiples of consecutive numbers.

Initially, we have the set of all the numbers $\{2, 3, \ldots, n\}$. At each step we choose the smallest number in the set and remove all its multiples. Notice that every composite number has a divisor of at most $\sqrt{n}$. In particular, it has a divisor which is a prime number. It is sufficient to remove only multiples of prime numbers not exceeding $\sqrt{n}$. In this way, all composite numbers will be removed.



The above illustration shows steps of sieving for $n = 17$. The elements of the processed set are in white, and removed composite numbers are in gray. First, we remove multiples of the smallest element in the set, which is 2. The next element remaining in the set is 3, and we also remove its multiples, and so on.



The above algorithm can be slightly improved. Notice that we needn't cross out multiples of i which are less than i^2. Such multiples are of the form $k \cdot i$, where $k < i$. These have already been removed by one of the prime divisors of k. After this improvement, we obtain the following implementation:

11.1: Sieve of Eratosthenes.

```
1  def sieve(n):
2      sieve = [True] * (n + 1)
3      sieve[0] = sieve[1] = False
4      i = 2
5      while (i * i <= n):
6          if (sieve[i]):
```

```
            k = i * i
            while (k <= n):
                sieve[k] = False
                k += i
        i += 1
    return sieve
```

Let's analyse the time complexity of the above algorithm. For each prime number $p_j \leqslant \sqrt{n}$ we cross out at most $\frac{n}{p_j}$ numbers, so we get the following number of operations:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \ldots = \sum_{p_j \leqslant \sqrt{n}} \frac{n}{p_j} = n \cdot \sum_{p_j \leqslant \sqrt{n}} \frac{1}{p_j} \tag{11.1}$$

The sum of the reciprocals of the primes $p_j \leqslant n$ equals asymptotically $O(\log \log n)$. So the overall time complexity of this algorithm is $O(n \log \log n)$. The proof is not trivial, and is beyond the scope of this article. An example proof can be found here.

11.1. Factorization

Factorization is the process of decomposition into prime factors. More precisely, for a given number x we want to find primes $p_1, p_2, \ldots, p_k$ whose product equals x.

Use of the sieve enables fast factorization. Let's modify the sieve algorithm slightly. For every crossed number we will remember the smallest prime that divides this number.

11.2: Preparing the array F for factorization.

```
def arrayF(n):
    F = [0] * (n + 1)
    i = 2
    while (i * i <= n):
        if (F[i] == 0):
            k = i * i
            while (k <= n):
                if (F[k] == 0):
                    F[k] = i;
                k += i
        i += 1
    return F
```

For example, take an array F with a value of $n = 20$:

0	0	2	0	2	0	2	3	2	0	2	0	2	3	2	0	2	0	2
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

With this approach we can factorize numbers very quickly. If we know that one of the prime factors of x is p, then all the prime factors of x are p plus the decomposition of $\frac{x}{p}$.

11.3: Factorization of x — $O(\log x)$.

```
def factorization(x, F):
    primeFactors = []
    while (F[x] > 0):
        primeFactors += [F[x]]
        x /= F[x]
    primeFactors += [x]
    return primeFactors
```

Number x cannot have more than $\log x$ prime factors, because every prime factor is $\geqslant 2$. Factorization by the above method works in $O(\log x)$ time complexity. Note that consecutive factors will be presented in non-decreasing order.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 12

Euclidean algorithm

The Euclidean algorithm is one of the oldest numerical algorithms still to be in common use. It solves the problem of computing the greatest common divisor (gcd) of two positive integers.

12.1. Euclidean algorithm by subtraction

The original version of Euclid's algorithm is based on subtraction: we recursively subtract the smaller number from the larger.

12.1: Greatest common divisor by subtraction.

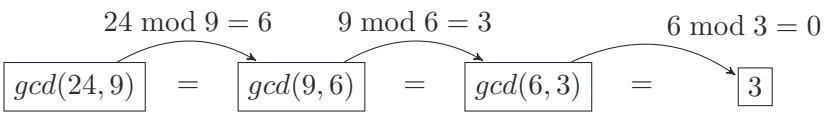
```
1 def gcd(a, b):
2     if a == b:
3         return a
4     if a > b:
5         gcd(a - b, b)
6     else:
7         gcd(a, b - a)
```

Let's estimate this algorithm's time complexity (based on $n = a+b$). The number of steps can be linear, for e.g. $gcd(x, 1)$, so the time complexity is $O(n)$. This is the worst-case complexity, because the value $x + y$ decreases with every step.

12.2. Euclidean algorithm by division

Let's start by understanding the algorithm and then go on to prove its correctness. For two given numbers a and b, such that $a \geqslant b$:

- if $b \mid a$, then $gcd(a, b) = b$,
- otherwise $\gcd(a, b) = gcd(b, a \bmod b)$.



Let's prove that $gcd(a, b) = gcd(b, r)$, where $r = a \bmod b$ and $a = b \cdot t + r$:

- Firstly, let $d = gcd(a, b)$. We get $d \mid (b \cdot t + r)$ and $d \mid b$, so $d \mid r$.
  Therefore we get $gcd(a, b) \mid gcd(b, r)$.
- Secondly, let $c = gcd(b, r)$. We get $c \mid b$ and $c \mid r$, so $c \mid a$.
  Therefore we get $gcd(b, r) \mid gcd(a, b)$.

Hence $gcd(a, b) = gcd(b, r)$. Notice that we can recursively call a function while a is not divisible by b.

12.2: Greatest common divisor by dividing.

```
1  def gcd(a, b):
2      if a % b == 0:
3          return b
4      else:
5          return gcd(b, a % b)
```

Denote by (a_i, b_i) pairs of values a and b, for which the above algorithm performs i steps. Then $b_i \geqslant Fib_{i-1}$ (where Fib_i is the i-th Fibonacci number). Inductive proof:

1. for one step, $b_1 = 0$,
2. for two steps, $b \geqslant 1$,
3. for more steps, $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$, then $a_k = b_{k+1}, a_{k-1} = b_k$, $b_{k-1} = a_k \bmod b_k$, so $a_k = q \cdot b_k + b_{k-1}$ for some $q \geqslant 1$, so $b_{k+1} \geqslant b_k + b_{k-1}$.

Fibonacci numbers can be approximated by:

$$Fib_n \approx \frac{(\frac{1+\sqrt{5}}{2})^n}{\sqrt{5}} \tag{12.1}$$

Thus, the time complexity is logarithmic based on the sum of a and b — $O(\log(a + b))$.

12.3. Binary Euclidean algorithm

This algorithm finds the *gcd* using only subtraction, binary representation, shifting and parity testing. We will use a *divide and conquer* technique.

The following function calculate `gcd(a, b, res)` $= gcd(a, b, 1) \cdot res$. So to calculate $gcd(a, b)$ it suffices to call `gcd(a, b, 1)` $= gcd(a, b)$.

12.3: Greatest common divisor using binary Euclidean algorithm.

```
1  def gcd(a, b, res):
2      if a == b:
3          return res * a
4      elif (a % 2 == 0) and (b % 2 == 0):
5          return gcd(a // 2, b // 2, 2 * res)
6      elif (a % 2 == 0):
7          return gcd(a // 2, b, res)
8      elif (b % 2 == 0):
9          return gcd(a, b // 2, res)
10     elif a > b:
11         return gcd(a - b, b, res)
12     else:
13         return gcd(a, b - a, res)
```

This algorithm is superior to the previous one for very large integers when it cannot be assumed that all the arithmetic operations used here can be done in a constant time. Due to the binary representation, operations are performed in linear time based on the length of the binary representation, even for very big integers. On the other hand, modulo applied in algorithm 10.2 has worse time complexity. It exceeds $O(\log n \cdot \log\log n)$, where $n = a + b$.

Denote by (a_i, b_i) pairs of values a and b, for which the above algorithm performs i steps. We have $a_{i+1} \geqslant a_i, b_{i+1} \geqslant b_i, b_1 = a_1 > 0$. In the first three cases, $a_{i+1} \cdot b_{i+1} \geqslant 2 \cdot a_i \cdot b_i$. In the fourth case, $a_{i+1} \cdot b_{i+1} \geqslant 2 \cdot a_{i-1} \cdot b_{i-1}$, because a difference of two odd numbers is an even number. By induction we get:

$$a_i \cdot b_i \geqslant 2^{\lfloor \frac{i-1}{2} \rfloor} \tag{12.2}$$

Thus, the time complexity is $O(\log(a \cdot b)) = O(\log a + b) = O(\log n)$. And for very large integers, $O((\log n)^2)$, since each arithmetic operation can be done in $O(\log n)$ time.

12.4. Least common multiple

The least common multiple (lcm) of two integers a and b is the smallest positive integer that is divisible by both a and b. There is the following relation:

$$lcm(a, b) = \frac{a \cdot b}{gcd(a,b)}$$

Knowing how to compute the $gcd(a, b)$ in $O(\log(a+b))$ time, we can also compute the $lcm(a, b)$ in the same time complexity.

12.5. Exercise

Problem: Michael, Mark and Matthew collect coins of consecutive face values a, b and c (each boy has only one kind of coins). The boys have to find the minimum amount of money that each of them may spend by using only their own coins.

Solution: It is easy to note that we want to find the least common multiple of the three integers, i.e. $lcm(a, b, c)$. The problem can be generalized for the lcm of exactly n integers. There is the following relation:

$$lcm(a_1, a_2, \ldots, a_n) = lcm(a_1, lcm(a_2, a_3, \ldots, a_n))$$

We simply find the lcm n times, and each step works in logarithmic time.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 13

Fibonacci numbers

The Fibonacci numbers form a sequence of integers defined recursively in the following way. The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two.

$$F_n = \begin{cases} 0 & \text{for } n = 0, \\ 1 & \text{for } n = 1, \\ F_{n-1} + F_{n-2} & \text{for } n > 1. \end{cases}$$

The first twelve Fibonacci numbers are:

0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	2	3	5	8	13	21	34	55	89

Notice that recursive enumeration as described by the definition is very slow. The definition of F_n repeatedly refers to the previous numbers from the Fibonacci sequence.

13.1: Finding Fibonacci numbers recursively.

```
def fibonacci(n):
    if (n <= 1):
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

The above algorithm performs F_n additions of 1, and, as the sequence grows exponentially, we get an inefficient solution.

Enumeration of the Fibonacci numbers can be done faster simply by using a basis of dynamic programming. We can calculate the values $F_0, F_1, \ldots, F_n$ based on the previously calculated numbers (it is sufficient to remember only the last two values).

13.2: Finding Fibonacci numbers dynamically.

```
def fibonacciDynamic(n):
    fib = [0] * (n + 2)
    fib[1] = 1
    for i in xrange(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

The time complexity of the above algorithm is $O(n)$.

13.1. Faster algorithms for Fibonacci numbers

Fibonacci numbers can be found in $O(\log n)$ time. However, for this purpose we have to use matrix multiplication and the following formula:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}, \text{ for } n \geqslant 1.$$

Even faster solution is possible by using the following formula:

$$Fib_n = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}} \tag{13.1}$$

These algorithms are not trivial and it will be presented in the future lessons.

13.2. Exercise

Problem: For all the given numbers $x_0, x_1, \ldots, x_{n-1}$, such that $1 \leqslant x_i \leqslant m \leqslant 1\,000\,000$, check whether they may be presented as the sum of two Fibonacci numbers.

Solution: Notice that only a few tens of Fibonacci numbers are smaller than the maximal m (exactly 31). We consider all the pairs. If some of them sum to $k \leqslant m$, then we mark index k in the array to denote that the value k can be presented as the sum of two Fibonacci numbers.

In summary, for each number x_i we can answer whether it is the sum of two Fibonacci numbers in constant time. The total time complexity is $O(n + m)$.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility

Chapter 14

Binary search algorithm

The binary search is a simple and very useful algorithm whereby many linear algorithms can be optimized to run in logarithmic time.

14.1. Intuition

Imagine the following game. The computer selects an integer value between 1 and 16 and our goal is to guess this number with a minimum number of questions. For each guessed number the computer states whether the guessed number is equal to, bigger or smaller than the number to be guessed.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The iterative check of all the successive values $1, 2, \ldots, 16$ is linear, because with each question the set of the candidates is reduced by one.

The goal is to ask a question that reduces the set of candidates maximally. The best option is to choose the middle element, as doing so causes the set of candidates to be halved each time. With this approach, we ask the logarithmic number of questions at maximum.

14.2. Implementation

In a binary search we use the information that all the elements are sorted. Let's try to solve the task in which we ask for the position of a value x in a sorted array $a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n-1}$. Let's see how the number of candidates is reduced, for example for the value $x = 31$.

12	15	15	19	24	31	53	59	60
0	1	2	3	4	5	6	7	8

12	15	15	19	24	31	53	59	60
0	1	2	3	(4)	5	6	7	8

12	15	15	19	24	31	53	59	60
0	1	2	3	4	5	(6)	7	8

For every step of the algorithm we should remember the beginning and the end of the remaining slice of the array (respectively, variables *beg* and *end*). The middle element of the slice can easily be calculated as $mid = \lfloor \frac{beg+end}{2} \rfloor$.

14.1: Binary search in $O(\log n)$.

```
def binarySearch(A, x):
    n = len(A)
    beg = 0
    end = n - 1
    result = -1
    while (beg <= end):
        mid = (beg + end) / 2
        if (A[mid] <= x):
            beg = mid + 1
            result = mid
        else:
            end = mid - 1
    return result
```

The above algorithm will find the largest element which is less than or equal to x. In subsequent iterations the number of candidates is halved, so the time complexity is $O(\log n)$. It is noteworthy that the above implementation is universal; it is enough to modify only the condition inside the while loop.

14.3. Binary search on the result

In many tasks, we should return some integer that is both optimal and that meets certain conditions. We can often find this number using a binary search. We guess some value and then check whether the result should be smaller or bigger. At the start we have a certain range in which we can find the result. After each attempt the range is halved, so the number of questions can be estimated by $O(\log n)$.

Thus, the problem of finding the optimal value reduces to checking whether some value is valid and optimal. The latter problem is often much simpler, and the binary search adds only a $\log n$ factor to the overall time complexity.

14.4. Exercise

Problem: You are given n binary values $x_0, x_1, \ldots, x_{n-1}$, such that $x_i \in \{0, 1\}$. This array represents holes in a roof (1 is a hole). You are also given k boards of the same size. The goal is to choose the optimal (minimal) size of the boards that allows all the holes to be covered by boards.

Solution: The size of the boards can be found with a binary search. If size x is sufficient to cover all the holes, then we know that sizes $x+1, x+2, \ldots, n$ are also sufficient. On the other hand, if we know that x is not sufficient to cover all the holes, then sizes $x-1, x-2, \ldots, 1$ are also insufficient.

14.2: Binary search in $O(\log n)$.

```
def boards(A, k):
    n = len(A)
    beg = 1
    end = n
    result = -1
    while (beg <= end):
        mid = (beg + end) / 2
        if (check(A, mid) <= k):
            end = mid - 1
            result = mid
```

```
11          else:
12              beg = mid + 1
13      return result
```

There is the question of how to check whether size x is sufficient. We can go through all the indices from the left to the right and greedily count the boards. We add a new board only if there is a hole that is not covered by the last board.

14.3: Greedily check in $O(n)$.

```
1  def check(A, k):
2      n = len(A)
3      boards = 0
4      last = -1
5      for i in xrange(n):
6          if A[i] == 1 and last < i:
7              boards += 1
8              last = i + k - 1
9      return boards
```

The total time complexity of such a solution is $O(n \log n)$ due to the binary search time.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 15

Caterpillar method

The Caterpillar method is a likeable name for a popular means of solving algorithmic tasks. The idea is to check elements in a way that's reminiscent of movements of a caterpillar. The caterpillar crawls through the array. We remember the front and back positions of the caterpillar, and at every step either of them is moved forward.

15.1. Usage example

Let's check whether a sequence $a_0, a_1, \ldots, a_{n-1}$ $(1 \leqslant a_i \leqslant 10^9)$ contains a contiguous subsequence whose sum of elements equals s. For example, in the following sequence we are looking for a subsequence whose total equals $s = 12$.

a_0	a_1	a_2	a_3	a_4	a_5	a_6
6	2	7	4	1	3	6

Each position of the caterpillar will represent a different contiguous subsequence in which the total of the elements is not greater than s. Let's initially set the caterpillar on the first element. Next we will perform the following steps:

- if we can, we move the right end ($front$) forward and increase the size of the caterpillar;
- otherwise, we move the left end ($back$) forward and decrease the size of the caterpillar.

In this way, for every position of the left end we know the longest caterpillar that covers elements whose total is not greater than s. If there is a subsequence whose total of elements equals s, then there certainly is a moment when the caterpillar covers all its elements.

15.1: Caterpillar in $O(n)$ time complexity.

```
def caterpillarMethod(A, s):
    n = len(A)
    front, total = 0, 0
    for back in xrange(n):
        while (front < n and total + A[front] <= s):
            total += A[front]
            front += 1
        if total == s:
            return True
        total -= A[back]
```

```
11      return False
```

Let's estimate the time complexity of the above algorithm. At the first glance we have two nested loops, what suggest quadratic time. However, notice that at every step we move the front or the back of the caterpillar, and their positions will never exceed n. Thus we actually get an $O(n)$ solution.

The above estimation of time complexity is based on amortized cost, which will be explained more precisely in future lessons.

15.2. Exercise

Problem: You are given n sticks (of lengths $1 \leqslant a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n-1} \leqslant 10^9$). The goal is to count the number of triangles that can be constructed using these sticks. More precisely, we have to count the number of triplets at indices $x < y < z$, such that $a_x + a_y > a_z$.

Solution $O(n^2)$: For every pair x, y we can find the largest stick z that can be used to construct the triangle. Every stick k, such that $y < k \leqslant z$, can also be used, because the condition $a_x + a_y > a_k$ will still be true. We can add up all these triangles at once.

If the value z is found every time from the beginning then we get a $O(n^3)$ time complexity solution. However, we can instead use the caterpillar method. When increasing the value of y, we can increase (as far as possible) the value of z.

15.2: The number of triangles in $O(n^2)$.

```
1  def triangles(A):
2      n = len(A)
3      result = 0
4      for x in xrange(n):
5          z = x + 2
6          for y in xrange(x + 1, n):
7              while (z < n and A[x] + A[y] > A[z]):
8                  z += 1
9              result += z - y - 1
10     return result
```

The time complexity of the above algorithm is $O(n^2)$, because for every stick x the values of y and z increase $O(n)$ number of times.

Every lesson will provide you with programming tasks at http://codility.com/programmers.

Codility_

Chapter 16

Greedy algorithms

We consider problems in which a result comprises a sequence of steps or choices that have to be made to achieve the optimal solution. Greedy programming is a method by which a solution is determined based on making the locally optimal choice at any given moment. In other words, we choose the best decision from the viewpoint of the current stage of the solution.

Depending on the problem, the greedy method of solving a task may or may not be the best approach. If it is not the best approach, then it often returns a result which is approximately correct but suboptimal. In such cases dynamic programming or brute-force can be the optimal approach. On the other hand, if it works correctly, its running time is usually faster than those of dynamic programming or brute-force.

16.1. The Coin Changing problem

For a given set of denominations, you are asked to find the minimum number of coins with which a given amount of money can be paid. That problem can be approached by a greedy algorithm that always selects the largest denomination not exceeding the remaining amount of money to be paid. As long as the remaining amount is greater than zero, the process is repeated.

A correct algorithm should always return the minimum number of coins. It turns out that the greedy algorithm is correct for only some denomination selections, but not for all. For example, for coins of values 1, 2 and 5 the algorithm returns the optimal number of coins for each amount of money, but for coins of values 1, 3 and 4 the algorithm may return a suboptimal result. An amount of 6 will be paid with three coins: 4, 1 and 1 by using the greedy algorithm. The optimal number of coins is actually only two: 3 and 3.

Consider n denominations $0 < m_0 \leqslant m_1 \leqslant \ldots \leqslant m_{n-1}$ and the amount k to be paid.

16.1: The greedy algorithm for finding change.

```
1 def greedyCoinChanging(M, k):
2     n = len(M)
3     result = []
4     for i in xrange(n - 1, -1, -1):
5         result += [(M[i], k // M[i])]
6         k %= M[i]
7     return result
```

The function returns the list of pairs: denomination, number of coins. The time complexity of the above algorithm is $O(n)$ as the number of coins is added once for every denomination.

16.2. Proving correctness

If we construct an optimal solution by making consecutive choices, then such a property can be proved by induction: if there exists an optimal solution consistent with the choices that have been made so far, then there also has to exist an optimal solution consistent with the next choice (including the situation when the first choice is made).

16.3. Exercise

Problem: There are $n > 0$ canoeists weighing respectively $1 \leqslant w_0 \leqslant w_1 \leqslant \ldots \leqslant w_{n-1} \leqslant 10^9$. The goal is to seat them in the minimum number of double canoes whose displacement (the maximum load) equals k. You may assume that $w_i \leqslant k$.

Solution A $O(n)$: The task can be solved by using a greedy algorithm. The heaviest canoeist is called *heavy*. Other canoeists who can be seated with *heavy* in the canoe are called *light*. All the other remaining canoeists are also called *heavy*.

The idea is that, for the heaviest *heavy*, we should find the heaviest *light* who can be seated with him/her. So, we seat together the heaviest *heavy* and the heaviest *light*. Let us note that the lighter the heaviest *heavy* is, the heavier *light* can be. Thus, the division between *heavy* and *light* will change over time — as the heaviest *heavy* gets closer to the pool of *light*.

16.2: Canoeist in $O(n)$ solution.

```
def greedyCanoeistA(W, k):
    N = len(W)
    light = deque()
    heavy = deque()
    for i in xrange(N - 1):
        if W[i] + W[-1] <= k:
            light.append(W[i])
        else:
            heavy.append(W[i])
    heavy.append(W[-1])
    canoes = 0
    while (light or heavy):
        if len(light) > 0:
            light.pop()
        heavy.pop()
        canoes += 1
        if (not heavy and light):
            heavy.append(light.pop())
        while (len(heavy) > 1 and heavy[-1] + heavy[0] <= k):
            light.append(heavy.popleft())
    return canoes
```

Proof of correctness: There exists an optimal solution in which the heaviest *heavy* h and the heaviest *light* l are seated together. If there were a better solution in which h sat alone then l could be seated with him/her anyway. If *heavy* h were seated with some *light* $x \leqslant l$, then x and l could just be swapped. If l has any companion y, x and y would fit together, as $y \leqslant h$.

The solution for the first canoe is optimal, so the problem can be reduced to seat the remaining canoeists in the minimum number of canoes.

The total time complexity of this solution is $O(n)$. The outer *while* loop performs $O(n)$ steps since in each step one or two canoeists are seated in a canoe. The inner *while* loop in each step changes a *heavy* into a *light*. As at the beginning there are $O(n)$ *heavy* and with each step at the outer *while* loop only one *light* become a *heavy*, the overall total number of steps of the inner *while* loop has to be $O(n)$.

Solution B $O(n)$: The heaviest canoeist is seated with the lightest, as long as their weight is less than or equal to k. If not, the heaviest canoeist is seated alone in the canoe.

16.3: Canoeist in $O(n)$ solution.

```
1  def greedyCanoeistB(W, k):
2      canoes = 0
3      j = 0
4      i = len(W) - 1
5      while (i >= j):
6          if W[i] + W[j] <= k:
7              j += 1;
8          canoes += 1;
9          i -= 1
10     return canoes
```

The time complexity is $O(n)$, because with each step of the loop, at least one canoeist is seated.

Proof of correctness: Analogically to solution A. If *light* l were seated with some *heavy* $x < h$, then x and h could just be swapped.

If the heaviest canoeist is seated alone, it is not possible to seat anybody with him/her. If there exists a solution in which the heaviest canoeist h is seated with some other x, we can swap x with the lightest canoeist l, because l can sit in place of x since $x \geqslant l$. Also, x can sit in place of l, since if l has any companion y, we have $y \leqslant h$.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.

Codility_

Chapter 17

Dynamic programming

Dynamic programming is a method by which a solution is determined based on solving successively similar but smaller problems. This technique is used in algorithmic tasks in which the solution of a bigger problem is relatively easy to find, if we have solutions for its sub-problems.

17.1. The Coin Changing problem

For a given set of denominations, you are asked to find the minimum number of coins with which a given amount of money can be paid. Assume that you can use as many coins of a particular denomination as necessary. The greedy algorithmic approach is always to select the largest denomination not exceeding the remaining amount of money to be paid. As long as the remaining amount is greater than zero, the process is repeated. However, this algorithm may return a suboptimal result. For instance, for an amount of 6 and coins of values $1, 3, 4$, we get $6 = 4 + 1 + 1$, but the optimal solution here is $6 = 3 + 3$.

A dynamic algorithm finds solutions to this problem for all amounts not exceeding the given amount, and for increasing sets of denominations. For the example data, it would consider all the amounts from 0 to 6, and the following sets of denominations: $\varnothing, \{1\}, \{1, 3\}$ and $\{1, 3, 4\}$. Let $dp[i, j]$ be the minimum number of coins needed to pay the amount j if we use the set containing the i smallest denominations. The number of coins needed must satisfy the following rules:

- no coins are needed to pay a zero amount: $dp[i, 0] = 0$ (for all i);
- if there are no denominations and the amount is positive, there is no solution, so for convenience the result can be infinite in this case: $dp[0, j] = \infty$ (for all $j > 0$);
- if the amount to be paid is smaller than the highest denomination c_i, this denomination can be discarded: $dp[i, j] = dp[i - 1, j]$ (for all $i > 0$ and all j such that $c_i > j$);
- otherwise, we should consider two options and choose the one requiring fewer coins: either we use a coin of the highest denomination, and a smaller amount to be paid remains, or we don't use coins of the highest denomination (and the denomination can thus be discarded): $dp[i, j] = \min(dp[i, j - c_i] + 1, dp[i - 1, j])$ (for all $i > 0$ and all j such that $c_i \leqslant j$).

The following table shows all the solutions to sub-problems considered for the example data.

$dp[i,j]$	0	1	2	3	4	5	6
$\varnothing$	0	∞	∞	∞	∞	∞	∞
$\{1\}$	0	1	2	3	4	5	6
$\{1,3\}$	0	1	2	1	2	3	2
$\{1,3,4\}$	0	1	2	1	1	2	2

Implementation

Consider n denominations, $0 < c_0 \leqslant c_1 \leqslant \ldots \leqslant c_{n-1}$. The algorithm processes the respective denominations and calculates the minimum number of coins needed to pay every amount from 0 to k. When considering each successive denomination, we use the previously calculated results for the smaller amounts.

17.1: The dynamic algorithm for finding change.

```
def dynamic_coin_changing(C, k):
    n = len(C)
    # create two-dimensional array with all zeros
    dp = [[0] * (k + 1) for i in xrange(n + 1)]
    dp[0] = [0] + [MAX_INT] * k
    for i in xrange(1, n + 1):
        for j in xrange(C[i - 1]):
            dp[i][j] = dp[i - 1][j]
        for j in xrange(C[i - 1], k + 1):
            dp[i][j] = min(dp[i][j - C[i - 1]] + 1, dp[i - 1][j])
    return dp[n]
```

Both the time complexity and the space complexity of the above algorithm is $O(n \cdot k)$. In the above implementation, memory usage can be optimized. Notice that, during the calculation of dp, we only use the previous row, so we don't need to remember all of the rows.

17.2: The dynamic algorithm for finding change with optimized memory.

```
def dynamic_coin_changing(C, k):
    n = len(C)
    dp = [0] + [MAX_INT] * k
    for i in xrange(1, n + 1):
        for j in xrange(C[i - 1], k + 1):
            dp[j] = min(dp[j - C[i - 1]] + 1, dp[j])
    return dp
```

The time complexity is $O(n \cdot k)$ and the space complexity is $O(k)$.

17.2. Exercise

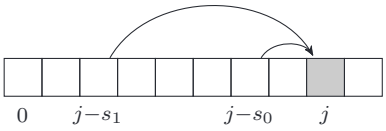
Problem: A small frog wants to get from position 0 to k ($1 \leqslant k \leqslant 10\,000$). The frog can jump over any one of n fixed distances $s_0, s_1, \ldots, s_{n-1}$ ($1 \leqslant s_i \leqslant k$). The goal is to count the number of different ways in which the frog can jump to position k. To avoid overflow, it is sufficient to return the result modulo q, where q is a given number.

We assume that two patterns of jumps are different if, in one pattern, the frog visits a position which is not visited in the other pattern.

Solution $O(n \cdot k)$: The task can be solved by using dynamic programming. Let's create an array dp consisting of k elements, such that $dp[j]$ will be the number of ways in which the frog can jump to position j.

We update consecutive cells of array dp. There is exactly one way for the frog to jump to position 0, so $dp[0] = 1$. Next, consider some position $j > 0$.

The number of ways in which the frog can jump to position j with a final jump of s_i is $dp[j - s_i]$. Thus, the number of ways in which the frog can get to position j is increased by the number of ways of getting to position $j - s_i$, for every jump s_i.



More precisely, $dp[j]$ is increased by the value of $dp[j - s_i]$ (for all $s_i \leqslant j$) modulo q.

17.3: Solution in time complexity $O(n \cdot k)$ and space complexity $O(k)$.

```
1  def frog(S, k, q):
2      n = len(S)
3      dp = [1] + [0] * k
4      for j in xrange(1, k + 1):
5          for i in xrange(n):
6              if S[i] <= j:
7                  dp[j] = (dp[j] + dp[j - S[i]]) % q;
8      return dp[k]
```

The time complexity is $O(n \cdot k)$ (all cells of array dp are visited for every jump) and the space complexity is $O(k)$.

Every lesson will provide you with programming tasks at http://codility.com/programmers.