codility

WE TEST CODERS

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 or objects. 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:

$A[i]$	5	2	8	14	1	16
i	0	1	2	3	4	5

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

$A[i]$	1	2	5	8	14	16
i	0	1	2	3	4	5

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 the first k elements will be sorted in the right order (this property is called the *invariant*).

6.1: Selection sort — $O(n^2)$.

```
def selectionSort(A):
    n = len(A)
    for k in xrange(n):
        minimal = k
        for j in xrange(k + 1, n):
            if (A[minimal] > A[j]):
                minimal = j
        A[k], A[minimal] = A[minimal], A[k]
    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 n elements takes $O(n)$ time; just repeatedly choose the lower of the first elements of the two merged parts.

The number of divisions will be $O(n)$, because 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

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.

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