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.