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.

```
def greedyCoinChanging(M, k):
    n = len(M)
    result = []
    for i in xrange(n - 1, -1, -1):
        result += [(M[i], k // M[i])]
        k %= M[i]
    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.

```
def greedyCanoeistB(W, k):
    canoes = 0
    j = 0
    i = len(W) - 1
    while (i >= j):
        if W[i] + W[j] <= k:
            j += 1;
        canoes += 1;
        i -= 1
    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.