Sorting and greedy algorithms (Programming Club 1)

Tomasz Kosciuszko

22.01.2018

1 Introduction

Imagine that you are a pirate and you just got into a secret vault, full of treasures. You are very experienced so you can immediately tell how much any artifact is worth. Assuming that you can just take two - one in each hand, which artifacts should you choose? Obviously, the most expansive ones:

Algorithm 1.1 (Pirate's) *Sort the items by how much they are worth. Then choose the two most expensive and take them.*

Example 1.1 *For example, say the treasures have values:* $5, 9, 3, 5, 1, 4$*. The sorted list looks* $1, 3, 4, 5, 5, 9$*. We decide on the last two artifacts, so* 5 *and* 9*.*

Such approach is called **greedy**, because we don't consider all possible solutions but take the one that looks good. In this trivial example the algorithm is straightforward and obviously correct. But as we will see that is not always the case.

Let's introduce a small modification. Apart of price each item will now have its weight. The pirate is allowed to take as many items as he wants, but not exceeding total limit of 10 kilograms. Which ones should he choose now?

Example 1.2 *For example, take the following pairs (weight, value):*
$(2, 2), (4, 3), (6, 3), (9, 4)$*. The optimal total value here is* 6*.*

Even though the problem seems similar to the previous one, there is no easy greedy solution. If the pirate decides to take the most valuable artifacts, or the lightest, we can always find a counter-example in which his strategy is not optimal.

We will see at the next meeting that a good approach to solve problems like this is **dynamic programming**, but for now let's try to solve a more interesting problem with the **greedy** approach.

2 Problem example - "Squares and not squares"

Let's have a look at this problem's statement:

`http://codeforces.com/problemset/problem/898/E`

Read through the description to make sure that you understand what it is asking for. We get to make operations of subtraction and addition of candles to piles, in such way that at the end $n/2$ piles are perfect squares and $n/2$ are not.

Example 2.1 *Say the piles have the sizes* $120, 110, 23, 34, 25, 45$*. The optimal solution is to change* 120 *to* 121 *and* 34 *to* 36*, thus adding* 3 *candles. After this* $121, 36, 25$ *are perfect squares and* $110, 23, 45$ *are not.*

Let's try to sort our numbers by how close they are to a perfect square. The following function written in JAVA will tell us how many operations on number x we need to bring it to a perfect square:

```
private static int distance(int x) {
    int sqrtFloor = (int)Math.sqrt(x);
    int low  = (int)Math.pow(sqrtFloor, 2);
    int high = (int)Math.pow(sqrtFloor + 1, 2);
    return Math.min(high - x, x - low);
}
```

Now let's remember how many operations we need to solve each pile in array *op*[]. In most cases the result will be just the sum of the $n/2$ smallest numbers from the array *op*[], let's sort the array and add the first $n/2$ numbers:

```
for (int i = 0; i < n; i++) {
    piles[i] = in.nextInt();
    op[i]    = distance(piles[i]);
}
Arrays.sort(op);
long result = 0;
for (int i = 0; i < n/2; i++) {
    result += op[i];
}
```

There is just one nasty case left. What if more than $n/2$ piles are perfect sqares at the beginning? Obviously we should spoil the appriopriate amount of squares. In first order we should spoil any squares different to 0, adding 1. Then, if needed, we should spoil piles with 0 candles, adding 2. The following code takes care of the edge case:

```
int perfectPositive = 0;
int perfectZero = 0;
for (int i = 0; i < n; i++) {
    if (distance(piles[i]) == 0) {
        if (piles[i] == 0) {
            perfectZero ++;
        } else {
            perfectPositive ++;
        }
    }
}
while (perfectPositive + perfectZero > n / 2) {
    if (perfectPositive > 0) {
        perfectPositive --;
        result ++;
    } else {
        perfectZero --;
        result += 2;
    }
}
```

Now try to put the example code together and compile it. Or, write your own if you prefer. All popular programming languages are supported on codeforces! Once you are convinced it works submit it via codeforces to check if it passes all the tests.

3 Important remarks

- Why is the presented solution greedy? Because we are sorting our objects by some feature and then building the solution from there.
- Why is it correct? Often, it is totally not obvious that a greedy solution is correct. Have a go on the proof!
- Is it fast enough? Always pay attention to the size of data you will have to process. In this problem we are told that $n \leq 200000$. Notice, that in the code above I used the Arrays.sort function implemented in the java.util package. The complexity of it is $O(n \log n)$ and because it is the slowest part of my code I should fit in the time limit. But if I used the bubble sort algorithm instead ($O(n^2)$) I would get ”Time limit exceeded” error and my solution would not be accepted!
- If you are not sure how to deal with **standard input and output** in your favourite language, have a look at some accepted solutions by other people: `http://codeforces.com/problemset/status`.

4 Practice problems

- Hungry Student Problem (EASY): `http://codeforces.com/problemset/problem/903/A`
- Cards (EASY): `http://codeforces.com/problemset/problem/701/A`
- Maxim and Discounts (MEDIUM): `http://codeforces.com/problemset/problem/261/A`
- Group Photo (HARD): `http://codeforces.com/problemset/problem/529/B`

5 Hints

- Hungry Student Problem: What is the biggest number for which the answer is "NO"?
- Cards: Can you compute how much points exactly each player should get?
- Maxim and Discounts: Is it ever beneficial for Maxim to get a more expensive product rather tham a cheaper one? Is it ever beneficial to use a bigger promotion, rather than a smaller one?
- Group Photo: Notice that n is small, you can try an $O(n^2)$ or even an $O(n^2 \log n)$ algorithm. Perhaps you could check all possible heights of the picture and for each of them, minimize the width.