How to devise this solution to Non-Constructible Change challenge from Algoexpert.io

Asked 11 months ago Modified 3 months ago Viewed 2k times

13

4

I'm working through algoexpert.io coding challenges and I'm having trouble undersatnding the suggested solution to one of the questions titled **Non-Constructible Change**

Here's the challenge question:

Given an array of positive integers representing the values of coins in your possession, write a function that returns the minimum amount of change (the minimum sum of money) that you **cannot** create. The given coins can have any positive integer value and aren't necessarily unique (i.e., you can have multiple coins of the same value).

For example, if you're given coins = [1, 2, 5], the minimum amount of change that you can't create is 4. If you're given no coins, the minimum amount of change that you can't create is 1.

```
// O(nlogn) time, O(n) size.
function nonConstructibleChange(coins) {
  coins = coins.sort((a, b) => a - b); // O(nlogn) time operation
  let change = 0;

  for (coin of coins) {
    if (coin > change + 1) return change + 1;
    change += coin;
  }

  return change + 1;
}
```

My problem

I am not completely sure how did the author of the solution come up with the intuition that

```
if the current coin is greater than `change + 1`, the smallest impossible change is
equal to `change + 1`.
```

I can see how it tracks, and indeed the algorithm passes all tests, but I'd like to know more about a process I could use to devise this rule.

Thank you for taking the time to read the question!

javascript algorithm

Share Improve this question Follow

edited Apr 18, 2021 at 15:22
RK-
11.8k 22 88 153

asked Apr 14, 2021 at 22:40
Daniel Kaczmarczyk
305 1 8

I'm afraid there's no formula for creativity. – Scott Sauyet Apr 18, 2021 at 18:28

1 ... and I don't mean that previous comment to be snide. This is an elegant answer to the problem, and I'm not sure I would have seen it. I'd suggest trying to figure out how *you* would solve this and then iterate upon it to see if you can improve it. Sometimes an "aha* moment will sneak up on you. And other times you simply toil away until you get something that works without elegant insights. Both can get the job done, although the leaps of intuition are more satisfying. – Scott Sauyet Apr 18, 2021 at 18:32

4 Answers

Active Oldest Votes

15

This one took me a while too, but this was how I made sense of it:

Assume you've proven you can make 1-8 cents.

You go to the next iteration and want to know if you can make 9 cents. So you iterate to the next new coin in the sorted list.

if newCoin < 9:

- You know for a fact that you can make 9 cents.

- Example if the new coin is 5: Use that new coin and subtract it from the total you're trying to make. 9 - 5 = 4. Then however way you made 4 previously just do that again. (You've already proven you can make 1-8 cents)

if newCoin == 9:

- You know for a fact that you can make 9 cents.
- Just use the 9 cent coin

if newCoin > 9:

- You know for a fact that you CANNOT make 9 cents
- This is because **you can't use the new coin**. For example, a 10 cent coin is useless to you when you're trying to make 9 cents since it's too big (can't make 9)
- **You're also screwed if you don't use the new coin** because if you add up all the coins you've seen so far, you've only been able to make 8 (can't make 9)

And that's where the change + 1 comes from. (your variable change = 8)

```
if the current coin is greater than `change + 1`, the smallest impossible change is
equal to `change + 1`.
```

Share Improve this answer Follow

answered Apr 26, 2021 at 6:35
Jack Windensky
151 2

This is helpful. It's also important a) that the coins are sorted first, so that you know that any remaining coins are the same as or bigger than the next one. And it's *also* important b) to consider the issue of gaps. If you have a set of *constructible* values with gaps in, and you add a new coin (allowed by the `<= max+1` rule), then you get a new set of values, also with gaps. So (imho, after trying to get my head round it too!) it's important to see that the algorithm also ensures that there will never be gaps in the list of values, at any step. C.f. @HemantJoshi's also useful answer. – MikeBeaton Nov 19, 2021 at 14:38

When integers are sorted, we can always track the highest constructible value by doing a cumulative sum of the sorted integers.

5

For example, `coins = [1, 2, 5] ==> 1,2,3,5,6,7`

At any point, if the next integer is larger than max constructible + 1, then there is no way to construct max constructible + 1.

```
[1]       mc=1     ---> 2
[1, 2]    mc=3     ---> 4
[1, 2, 5] mc=7     ---> 8
```

If the current integer is greater than `mc + 1`, the smallest impossible change is equal to `mc + 1`.

So in this case, for `[1, 2, 5]` the smallest value is 4.

It can be done like this (i am using go)

```
    func NonConstructibleChange(array []int) int {
        sort.Ints(array)
        ncc :=1
        for i:=0; i< len(array) && array[i]<=ncc; i++{
            ncc +=array[i]
        }
        return ncc
    }
```

Share Improve this answer Follow

edited Apr 24, 2021 at 13:51

answered Apr 24, 2021 at 13:30
Hemant Joshi
51 3

3 "When integers are sorted, we can always track the highest constructible value by doing a cumulative sum of the sorted integers." How do you know for sure that knowing this cumulative sum of sorted integers means that we can construct the value for all values between 1 and the cumulative sum? None of the solutions I've seen explain how we came to that conclusion, they kinda just give it as something that is evident. – FrostyStraw Aug 11, 2021 at 22:46

If you re-read a couple of times (I had to!) @HemantJoshi is not saying you can construct every value up to the highest constructible value. He is *just* saying that it is the highest value you can construct. There may be gaps - and in the first example, there *is* a gap, at 4. So looking for the highest *non-*constructible value is basically looking for the first gap! :-) – MikeBeaton Nov 19, 2021 at 14:25

2

After a while thinking about that, I thought it was a probability problem. I mean, how many changes are possible given N coins?

This mean that I should combine every possibility. For example, given `coins = [1, 5, 9]`, what's the minimum amount of change that can be made?

```
1 -> 1, 1+5, 1+9, 1+5+9.
5 -> 5, 5+9.
9 -> 9.
```

resulting in `possible changes = [1, 6, 10, 15, 14, 9]`.
Then, the minimum amount of change that can be created using this 3 coins `[1, 5, 9]` should be `2`.

By the way, if that was the case, this is a `O(2^n) time` because for every new coin (every new possibility) the number of calculations double.

Unfortunately, that is not the case...

Share Improve this answer Follow

answered Jul 8, 2021 at 16:01
Guilherme Caraciolo
104 1 6

2 *is* indeed the minimal amount of change that *cannot* be created with these three coins - so it is indeed the correct answer! And this is a correct (but inefficient) way of solving the problem. (A *problem* is O(n) if an O(n) algorithm is the fastest solution - this does not mean that there cannot be a different, *correct*, but less efficient, e.g. O(n^2) solution, as here.) – MikeBeaton Nov 19, 2021 at 14:42

0

I managed to tackle it without the + 1 thing, by tracking the current minimum impossible change, like so:

```
function nonConstructibleChange(coins) {
  let currentMinImpossibleChange = 1;

  const sortedCoins = coins.sort((a,b) => a-b)

  for(let i = 0; i < sortedCoins.length ; i += 1) {
    if(currentMinImpossibleChange < sortedCoins[i]) return currentMinImpossibleChange
    currentMinImpossibleChange += sortedCoins[i];
  }

  return currentMinImpossibleChange;
}
```

Share Improve this answer Follow

answered Dec 2, 2021 at 20:13

Murilo Botelho

15 4