Smallest number that can not be formed from sum of numbers from array

Asked 8 years, 2 months ago Modified 10 months ago Viewed 18k times

30

23

This problem was asked to me in Amazon interview -

Given a array of positive integers, you have to find the smallest positive integer that can not be formed from the sum of numbers from array.

Example:

```
Array:[4 13 2 3 1]
result= 11 { Since 11 was smallest positive number which can not be formed from the given array elements }
```

What i did was :

1. sorted the array
2. calculated the prefix sum
3. Treverse the sum array and check if next element is less than 1 greater than sum i.e. A[j]<=(sum+1). If not so then answer would be **sum+1**

But this was nlog(n) solution.

Interviewer was not satisfied with this and asked a solution in less than O(n log n) time.

arrays algorithm data-structures dynamic-programming subset-sum

Share Improve this question Follow

edited Mar 12, 2016 at 23:18

templatetypedef

342k 96 849 1024

asked Jan 12, 2014 at 17:19

user3187810

303 1 3 6

1 Are you saying that the interviewer asked for a O(logn) solution? That's obviously not possible because you have to look at each array value once, which would take at least O(n). – interjay Jan 12, 2014 at 17:23

4 Probably need to be more specific here: Smallest integer greater than zero which can not be created by summing any combination of the elements of the array, perhaps? – DavidO Jan 12, 2014 at 17:24

1 Are the array elements all positive integers? Can there be duplicates? – interjay Jan 12, 2014 at 17:29

1 Does the problem's spec guarantee a maximum possible integer value substantially less than INT_MAX? – DavidO Jan 12, 2014 at 17:35

2 Isn't this coincidently very similar to this question that was asked yesterday? stackoverflow.com/questions/21060873/... – Abhishek Bansal Jan 12, 2014 at 17:41

4 Answers

Active | Oldest | Votes

53

There's a beautiful algorithm for solving this problem in time O(n + Sort), where Sort is the amount of time required to sort the input array.

The idea behind the algorithm is to sort the array and then ask the following question: what is the smallest positive integer you cannot make using the first k elements of the array? You then scan forward through the array from left to right, updating your answer to this question, until you find the smallest number you can't make.

Here's how it works. Initially, the smallest number you can't make is 1. Then, going from left to right, do the following:

- If the current number is bigger than the smallest number you can't make so far, then you know the smallest number you can't make - it's the one you've got recorded, and you're done.
- Otherwise, the current number is less than or equal to the smallest number you can't make. The claim is that you can indeed make this number. Right now, you know the smallest number you can't make with the first k elements of the array (call it `candidate`) and are now looking at value `A[k]`. The number `candidate - A[k]` therefore must be some number that you can indeed make with the first k elements of the array, since otherwise `candidate - A[k]` would be a smaller number than the smallest number you allegedly can't make with the first k numbers in the array. Moreover, you can make any number in the range `candidate` to `candidate + A[k]`, inclusive, because you can start with any number in the range from 1 to `A[k]`, inclusive, and then add `candidate - 1` to it. Therefore, set `candidate` to `candidate + A[k]` and increment `k`.

In pseudocode:

```
Sort(A)
candidate = 1
for i from 1 to length(A):
   if A[i] > candidate: return candidate
   else: candidate = candidate + A[i]
return candidate
```

Here's a test run on `[4, 13, 2, 1, 3]`. Sort the array to get `[1, 2, 3, 4, 13]`. Then, set `candidate` to 1. We then do the following:

- A[1] = 1, `candidate` = 1:
  - A[1] ≤ `candidate`, so set `candidate = candidate + A[1] = 2`
- A[2] = 2, `candidate` = 2:
  - A[2] ≤ `candidate`, so set `candidate = candidate + A[2] = 4`
- A[3] = 3, `candidate` = 4:
  - A[3] ≤ `candidate`, so set `candidate = candidate + A[3] = 7`
- A[4] = 4, `candidate` = 7:
  - A[4] ≤ `candidate`, so set `candidate = candidate + A[4] = 11`
- A[5] = 13, `candidate` = 11:
  - A[4] > `candidate`, so return `candidate` (11).

So the answer is 11.

The runtime here is O(n + Sort) because outside of sorting, the runtime is O(n). You can clearly sort in O(n log n) time using heapsort, and if you know some upper bound on the numbers you can sort in time O(n log U) (where U is the maximum possible number) by using radix sort. If U is a fixed constant, (say, 10^9), then radix sort runs in time O(n) and this entire algorithm then runs in time O(n) as well.

Hope this helps!

Share Improve this answer Follow

edited Jan 4, 2019 at 16:18

answered Jan 12, 2014 at 17:52

templatetypedef
342k 96 849 1024

1 It should be `candidate = candidate + A[i]` in the `else`, without the `-1`. This is exactly the same algorithm as given by OP, but the explanation is very helpful. – interjay Jan 12, 2014 at 18:00

1 @user3187810- This solution is pretty fast - it runs in no worse than O(n log n) time and possibly a lot better if you can sort the integers using something like radix sort. – templatetypedef Jan 12, 2014 at 18:02

1 @interjay: I updated the answer. I didn't realize when I was writing this that it ended up being identical to the OP's answer. Now that I realize this, I think the answer is still useful in that it provides a justification for the answer and also shows how to speed it up (namely, improving the sorting step). If you think this isn't necessary, though, I can delete this answer. – templatetypedef Jan 12, 2014 at 18:04

5 @user3187810- If the integers have some fixed upper bound (say, 10^9), you can sort them in time O(n) by using counting sort or radix sort. That would then drop the total runtime to O(n). – templatetypedef Jan 12, 2014 at 18:10

1 If the numbers in the array are randomly generated, a statistically significant improvement can be made by simply checking if 1 exists before performing the rest of the algorithm. – neeKo Jan 12, 2014 at 21:15

9

Use bitvectors to accomplish this in linear time.

Start with an empty bitvector b. Then for each element k in your array, do this:

b = b | b << k | 2^(k-1)

To be clear, the i'th element is set to 1 to represent the number i, and `| k` is setting the k-th element to 1.

After you finish processing the array, the index of the first zero in b is your answer (counting from the right, starting at 1).

1. b=0
2. process 4: b = b | b<<4 | 1000 = 1000
3. process 13: b = b | b<<13 | 1000000000000 = 10001000000001000
4. process 2: b = b | b<<2 | 10 = 1010101000000101010
5. process 3: b = b | b<<3 | 100 = 1011111101000101111110

6. process 1: b = b | b<<1 | 1 = 11111111111001111111111

First zero: position 11.

Share Improve this answer Follow

edited Nov 1, 2015 at 13:56

answered Jan 12, 2014 at 20:50

Dave
5,700 1 22 35

2 Note that this is linear time IF the bitvector operations are constant time, which they might not be. – Dave Jan 12, 2014 at 21:34

1 To the best of my knowledge, there aren't any computers that support bitwise operations on arbitrary-width numbers in constant time. This is definitely a cool idea, but I don't think it's really O(n). – templatetypedef Jan 12, 2014 at 21:58

1 @templatetypedef: Fair point. OP answered in comments that the integers were guaranteed to be in the range of [1,10^9], so a sufficiently large bitvector to occupy that entire space could be reserved in constant time at the start. Even without that allowance, doubling reserved size every time allocated space was exceeded should constrain you to O(lg n) allocations. – Conspicuous Compiler Jan 15, 2014 at 8:54

1 @DaveGalvin Is `>>` a shift? Cause that's a shift right not a shift left. Even if it is a shift left, I must not be understanding something, cause in your step 3: `1|8192|1` does not equal 8209. – Jonathan Mee Jan 9, 2015 at 14:59

1 @JonathanMee I had written a mirror-universe version of the algorithm! Amazing that nobody else caught that or mentioned it. It's correct now. Thanks! – Dave Nov 1, 2015 at 13:55

7

Consider all integers in interval $[2^i .. 2^{i+1} - 1]$. And suppose all integers below 2^i can be formed from sum of numbers from given array. Also suppose that we already know C, which is sum of all numbers below 2^i. If $C >= 2^{i+1} - 1$, every number in this interval may be represented as sum of given numbers. Otherwise we could check if interval $[2^i .. C + 1]$ contains any number from given array. And if there is no such number, C + 1 is what we searched for.

Here is a sketch of an algorithm:

1. For each input number, determine to which interval it belongs, and update corresponding sum: `S[int_log(x)] += x`.
2. Compute prefix sum for array S: `foreach i: C[i] = C[i-1] + S[i]`.
3. Filter array C to keep only entries with values lower than next power of 2.

4. Scan input array once more and notice which of the intervals [2^i .. C + 1] contain at least one input number: `i = int_log(x) - 1; B[i] |= (x <= C[i] + 1)`.
5. Find first interval that is not filtered out on step #3 and corresponding element of `B[]` not set on step #4.

If it is not obvious why we can apply step 3, here is the proof. Choose any number between 2^i and C, then sequentially subtract from it all the numbers below 2^i in decreasing order. Eventually we get either some number less than the last subtracted number or zero. If the result is zero, just add together all the subtracted numbers and we have the representation of chosen number. If the result is non-zero and less than the last subtracted number, this result is also less than 2^i, so it is "representable" and none of the subtracted numbers are used for its representation. When we add these subtracted numbers back, we have the representation of chosen number. This also suggests that instead of filtering intervals one by one we could skip several intervals at once by jumping directly to int_log of C.

Time complexity is determined by function `int_log()`, which is integer logarithm or index of the highest set bit in the number. If our instruction set contains integer logarithm or any its equivalent (count leading zeros, or tricks with floating point numbers), then complexity is O(n). Otherwise we could use some bit hacking to implement `int_log()` in O(log log U) and obtain O(n * log log U) time complexity. (Here U is largest number in the array).

If step 1 (in addition to updating the sum) will also update minimum value in given range, step 4 is not needed anymore. We could just compare C[i] to Min[i+1]. This means we need only single pass over input array. Or we could apply this algorithm not to array but to a stream of numbers.

Several examples:

```
Input:       [ 4 13  2  3  1]    [ 1  2  3  9]    [ 1  1  2  9]
int_log:       2  3  1  1  0       0  1  1  3       0  0  1  3

int_log:     0  1  2  3          0  1  2  3       0  1  2  3
S:           1  5  4 13          1  5  0  9       2  2  0  9
C:           1  6 10 23          1  6  6 15       2  4  4 13
filtered(C): n  n  n  n          n  n  n  n       n  n  n  n
number in
[2^i..C+1]:  2  4  -             2  -  -          2  -  -
C+1:            11                  7                5
```

For multi-precision input numbers this approach needs O(n * log M) time and O(log M) space. Where M is largest number in the array. The same time is needed just to read all the numbers (and in the worst case we need every bit of them).

Still this result may be improved to O(n * log R) where R is the value found by this algorithm (actually, the output-sensitive variant of it). The only modification needed for this optimization is instead of processing whole numbers at once, process them digit-by-digit: first pass processes the low order bits of each number (like bits 0..63), second pass - next bits (like 64..127), etc. We could ignore all higher-order bits after result is found. Also this decreases space requirements to O(K) numbers, where K is number of bits in machine word.

Share Improve this answer Follow

edited Jan 12, 2015 at 17:54

answered Jan 12, 2014 at 20:24

Evgeny Kluev
23.8k 7 52 93

Can you please explain how this works for { 1 2 3 9 } and { 1 1 2 9 } – user3187810 Jan 13, 2014 at 7:02

OK. Several examples added. – Evgeny Kluev Jan 13, 2014 at 7:37

@EvgenyKluev I'm looking at your examples I can't figure out how your "S:" line is being calculated. In your description you mention prefix sum, but that is certainly not prefix sum. – Jonathan Mee Jan 9, 2015 at 15:21

@JonathanMee: actually, "C" is prefix sum, not "S". "S[i]" is sum of values from input array having integer logarithm equal to "i". And "C[i]" is sum of values having integer logarithm less or equal to "i". – Evgeny Kluev Jan 9, 2015 at 16:10

1 @EvgenyKluev Thanks for the explanation I now understand `C` and `S`. But I'm stuck again on Step 3. I don't understand what you mean by "next power of 2". – Jonathan Mee Jan 9, 2015 at 16:28

0

If you sort the array, it will work for you. Counting sort could've done it in `O(n)`, but if you think in a practically large scenario, range can be pretty high.

Quicksort `O(n*logn)` will do the work for you:

```
def smallestPositiveInteger(self, array):
    candidate = 1
    n = len(array)
    array = sorted(array)
    for i in range(0, n):
        if array[i] <= candidate:
            candidate += array[i]
        else:
```

```
            break
    return candidate
```

Share Improve this answer Follow

answered Apr 19, 2021 at 12:31

LITDataScience
124 10