All possible sum of each array combination

Asked 3 months ago Modified 3 months ago Viewed 88 times

3

Is there a name for this algorithm?

I have an array {1,2,3} and all my possible sums are

{1},{2},{3},{1+2},{1+3},{2+3}, {1+2+3} = {1},{2},{3},{4},{5},{6}

{1,1,2} => {1}, {2}, {3}, {4}

I tried to find another algorithm rather than brute-force combinations, is there any formal name for algorithm that solves this problem or any better implementation like using dynamic programming?

Constraints - array can have n positive integers, repeated or not

- input - array of integers
- output - array of every possible sum you could make with the input array

  {1,1,2} - {1} {2} {3} {4}

{1,2,2} - {1} {2} {3} {4} {5}

algorithms time-complexity combinatorics

Share Cite Improve this question Follow

edited Nov 30, 2021 at 16:04

asked Nov 29, 2021 at 19:46

Seven to sun
31 3

This is not a power set, and the OP said he didn't want to brute-force the combinations (which is what using a power set will do) – nir shahar Nov 29, 2021 at 20:23

What other constraints do you have on the input? is the array always of the form $[n] := \{k \mid k \in \mathbb{N}, 1 \leq k \leq n\}$? – nir shahar Nov 29, 2021 at 20:24

1 *Is there a name for this algorithm*: You are not describing an *algorithm*, but a *problem*. You are *looking* for an (output-)efficient algorithm to solve your problem, presumably one whose efficiency is measured against the size of both input and output. – Yuval Filmus Nov 30, 2021 at 7:51

@nirshahar I updated the title, your form is right. – Seven to sun Nov 30, 2021 at 15:59

@YuvalFilmus, yes, I updated the title to better explain my idea – Seven to sun Nov 30, 2021 at 16:02

2 Answers

Active | Oldest | Votes

4

Sadly, the worst case scenario is exponential in time, no matter what algorithm you use.

To see why, consider the following set of numbers: $A := \{2^k \mid 0 \leq k \leq n\}$. There are n numbers in this set, each taking at most n bits to represent. Hence the input is of size $m := n^2$.

Now, notice that for any two subsets $S_1, S_2 \subseteq A$, if $S_1 \neq S_2$ then there must be some i such that (without loss of generality) $2^i \in S_1$ but $2^i \notin S_2$.

We can think of the numbers in A as binary numbers, and then 2^i will be the binary number with 1 at the i'th spot and 0 everywhere else. You can easily show that you cannot construct 2^i as a sum of numbers from $A \setminus \{2^i\}$ (try to prove this!), and in particular this means that for any subset of $A \setminus \{2^i\}$, the i'th bit in the summation will always be 0 (again, show this!).

Since $2^i \notin S_2$, then $S_2 \subseteq A \setminus \{2^i\}$ and therefore $s_2 := \sum_{x \in S_2} x$ will have 0 at the i'th spot in its binary representation. But $s_1 := \sum_{x \in S_1} x = 2^i + \sum_{x \in S_1 \setminus \{2^i\}} x$.

Denote $s_1' := \sum_{x \in S_1 \setminus \{2^i\}} x$ and since $S_1 \setminus \{2^i\} \subseteq A \setminus \{2^i\}$ then the i'th bit at s_1' must be 0.

Thus, the i'th bit at s_1 has to be 1, since it is a sum of 2^i with another number that has 0 at the i'th bit.

But this means that $s_1 \neq s_2$, since s_2 had 0 at the i'th bit!

Thus the summation operation must be a one-to-one (injective) function, and hence the number of different summations has to be at least the number os subsets of A (and its not hard to show it actually equals that), which is $|P(A)| = 2^{|A|} = 2^n = 2^{\sqrt{m}}$.

Thus the output in this case is of size at least $2^{\sqrt{m}}$ when the input was of size (in bits) m. One can even improve upon this bound to get that the output size in bits has to be $\Omega(2^{2\sqrt{m}})$, which is achieved by observing that all values in the output are distinct natural numbers.

Notice that this is result shows that any algorithm will have to "brute-force" in the specific case of this A.

That being said, one *can* construct a dynamic programming solution for this problem when we know that the size of the output (i.e, the number of different summations) is small. Say, the output has k elements - then one can construct a dynamic programming solution with $O(n \cdot k)$:

- For any $1 \leq i \leq n$, define sum_i to be the *sorted* array of all possible summations of the first i items in A.
- To compute sum_{i+1}, do the following:
  1. create a copy of sum_i called sum_i', and you will need to add x_i (the i'th element in A) to each of the values in sum_i'.
  2. Now, merge sum_i and sum_i' together, **while removing any duplicates** (this step can take an additional $O(|sum_i| + |sum_i'|) = O(k)$ time, if we use the merging algorithm from MergeSort).
  3. If we also add k to the resulting array, it will be exactly sum_{i+1} (prove this!).

This will be computed n times, for a total of $O(n \cdot k)$.

Share Cite Improve this answer Follow

Share Cite Improve this answer Follow

edited Nov 30, 2021 at 20:22

answered Nov 30, 2021 at 17:20

nir shahar
10.5k 3 10 33

2 If you keep the arrays sorted, merge takes linear time. – Yuval Filmus Nov 30, 2021 at 17:32

Thanks @YuvalFilmus, I have edited the answer and improved it :) – nir shahar Nov 30, 2021 at 17:42

0

If you have n random real numbers, then there are 2^n possible and different sums. So you need 2^n operations

For something like your example, many small integers, you calculate all possible sums involving the first number, then the first two, etc. You will have many sums that are equal, so that saves time. And you will have complete ranges. For example “all integers from 12 to 17”. If the next number to add is 6, you get a range “all integers from 12 to 23” in constant time.

Share Cite Improve this answer Follow

answered Dec 1, 2021 at 9:33

gnasher729
24.6k 28 39