Subset sum problem

From Wikipedia, the free encyclopedia

In computer science, the **subset sum problem** is an important problem in complexity theory and cryptography. The problem is this: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero? For example, given the set {−7, −3, −2, 5, 8}, the answer is *yes* because the subset {−3, −2, 5} sums to zero. The problem is NP-complete.

An equivalent problem is this: given a set of integers and an integer *s*, does any non-empty subset sum to *s*? Subset sum can also be thought of as a special case of the knapsack problem.[1] One interesting special case of subset sum is the partition problem, in which *s* is half of the sum of all elements in the set.

Contents

Complexity

The complexity of the subset sum problem can be viewed as depending on two parameters, *N*, the number of decision variables, and *P*, the precision of the problem (stated as the number of binary place values that it takes to state the problem). (Note: here the letters *N* and *P* mean something different from what they mean in the NP class of problems.)

The complexity of the best known algorithms is exponential in the smaller of the two parameters *N* and *P*. Thus, the problem is most difficult if *N* and *P* are of the same order. It only becomes easy if either *N* or *P* becomes very small.

If *N* (the number of variables) is small, then an exhaustive search for the solution is practical. If *P* (the number of place values) is a small fixed number, then there are dynamic programming algorithms that can solve it exactly.

Efficient algorithms for both small *N* and small *P* cases are given below.

Exponential time algorithm

There are several ways to solve subset sum in time exponential in *N*. The most naïve algorithm would be to cycle through all subsets of N numbers and, for every one of them, check if the subset sums to the right number. The running time is of order $O(2^N N)$, since there are 2^N subsets and, to check each subset, we need to sum at most *N* elements.

A better exponential time algorithm is known which runs in time $O(2^{N/2})$. The algorithm splits arbitrarily the N elements into two sets of $N/2$ each. For each of these two sets, it stores a list of the sums of all $2^{N/2}$ possible subsets of its elements. Each of these two lists is then sorted. Using a standard comparison sorting algorithm for this step would take time $O(2^{N/2}N)$. However, given a sorted list of sums for k elements, the list can be expanded to two sorted lists with the introduction of a $(k + 1)$st element, and these two sorted lists can be merged in time $O(2^k)$. Thus, each list can be generated in sorted form in time $O(2^{N/2})$. Given the two sorted lists, the algorithm can check if an element of the first array and an element of the second array sum up to s in time $O(2^{N/2})$. To do that, the algorithm passes through the first array in decreasing order (starting at the largest element) and the second array in increasing order (starting at the smallest element). Whenever the sum of the current element in the first array and the current element in the second array is more than s, the algorithm moves to the next element in the first array. If it is less than s, the algorithm moves to the next element in the second array. If two elements with sum s are found, it stops. Horowitz and Sahni first published this algorithm in a technical report in 1972.[2]

Pseudo-polynomial time dynamic programming solution

The problem can be solved in pseudo-polynomial time using dynamic programming. Suppose the sequence is

$$x_1, \ldots, x_N$$

sorted in the increasing order and we wish to determine if there is a nonempty subset which sums to zero. Define the boolean-valued function $Q(i, s)$ to be the value (**true** or **false**) of

"there is a nonempty subset of $x_1, \ldots, x_i$ which sums to s".

Thus, the solution to the problem "Given a set of integers, is there a non-empty subset whose sum is zero?" is the value of $Q(N, 0)$.

Let A be the sum of the negative values and B the sum of the positive values. Clearly, $Q(i, s) =$ **false**, if $s < A$ or $s > B$. So these values do not need to be stored or computed.

Create an array to hold the values $Q(i, s)$ for $1 \leqslant i \leqslant N$ and $A \leqslant s \leqslant B$.

The array can now be filled in using a simple recursion. Initially, for $A \leqslant s \leqslant B$, set

$$Q(1, s) := (x_1 == s)$$

where $==$ is a boolean function that returns true if x_1 is equal to s, false otherwise.

Then, for $i = 2, \cdots, N$, set

$$Q(i, s) := Q(i - 1, s) \text{ or } (x_i == s) \text{ or } Q(i - 1, s - x_i),\ \text{ for } A \leqslant s \leqslant B.$$

For each assignment, the values of Q on the right side are already known, either because they were stored in the table for the previous value of i or because $Q(i - 1,s - x_i) =$ **false** if $s - x_i < A$ or $s - x_i > B$. Therefore, the total number of arithmetic operations is $O(N(B - A))$. For example, if all the values are $O(N^k)$ for some k, then the time required is $O(N^{k+2})$.

This algorithm is easily modified to return the subset with sum 0 if there is one.

The dynamic programming solution has runtime of $O(sN)$ where s is the sum we want to find in set of N numbers. This solution does not count as polynomial time in complexity theory because $B - A$ is not polynomial in the *size* of the problem, which is the number of bits used to represent it. This algorithm is polynomial in the values of A and B, which are exponential in their numbers of bits.

For the case that each x_i is positive and bounded by a fixed constant C, Pisinger (http://www.diku.dk/~pisinger/) found a linear time algorithm having time complexity $O(NC)$ (note that this is for the version of the problem where the target sum is not necessarily zero, otherwise the problem would be trivial).[3] In 2015, Koiliaris and Xu found the $\tilde{O}(s\sqrt{N})$ algorithm for the subset sum problem where s is the sum we need to find.[4]

Polynomial time approximate algorithm

An approximate version of the subset sum would be: given a set of N numbers $x_1, x_2, ..., x_N$ and a number s, output

- yes, if there is a subset that sums up to s;
- no, if there is no subset summing up to a number between $(1 - c)s$ and s for some small $c > 0$;
- any answer, if there is a subset summing up to a number between $(1 - c)s$ and s but no subset summing up to s.

If all numbers are non-negative, the approximate subset sum is solvable in time polynomial in N and $1/c$.

The solution for subset sum also provides the solution for the original subset sum problem in the case where the numbers are small (again, for nonnegative numbers). If any sum of the numbers can be specified with at most P bits, then solving the problem approximately with $c = 2^{-P}$ is equivalent to solving it exactly. Then, the polynomial time algorithm for approximate subset sum becomes an exact algorithm with running time polynomial in N and 2^P (i.e., exponential in P).

The algorithm for the approximate subset sum problem is as follows:

```
initialize a list $S$ to contain one element 0.
for each $i$ from 1 to $N$ do
  let $T$ be a list consisting of $x_i + y$, for all $y$ in $S$
  let $U$ be the union of $T$ and $S$
  sort $U$
  make $S$ empty
  let $y$ be the smallest element of $U$
  add $y$ to $S$
  for each element $z$ of $U$ in increasing order do
     //trim the list by eliminating numbers close to one another
     //and throw out elements greater than $s$
    if $y + cs/N < z \leqslant s$, set $y = z$ and add $z$ to $S$
if $S$ contains a number between $(1 - c)s$ and $s$, output yes, otherwise no
```

The algorithm is polynomial time because the lists S, T and U always remain of size polynomial in N and $1/c$ and, as long as they are of polynomial size, all operations on them can be done in polynomial time. The size of lists is kept polynomial by the trimming step, in which we only include a number z into S if it is greater than the previous one by cs/N and not greater than s.

This step ensures that each element in S is smaller than the next one by at least cs/N and do not contain elements greater than s. Any list with that property consists of no more than N/c elements.

The algorithm is correct because each step introduces an additive error of at most cs/N and N steps together introduce the error of at most cs.

See also

- 3SUM
- Merkle – Hellman knapsack cryptosystem

References

1. Martello, Silvano; Toth, Paolo (1990). "4 Subset-sum problem". *Knapsack problems: Algorithms and computer interpretations*. Wiley-Interscience. pp. 105 – 136. ISBN 0-471-92420-2. MR 1086874 (https://www.ams.org/mathscinet-getitem?mr=1086874).
2. Horowitz, Ellis; Sahni, Sartaj (1974), "Computing partitions with applications to the knapsack problem", *Journal of the Association for Computing Machinery*, **21**: 277 – 292, doi:10.1145/321812.321823 (https://doi.org/10.1145%2F321812.321823), MR 0354006 (https://www.ams.org/mathscinet-getitem?mr=0354006)
3. Pisinger D (1999). "Linear Time Algorithms for Knapsack Problems with Bounded Weights". *Journal of Algorithms*, Volume 33, Number 1, October 1999, pp. 1 – 14
4. Koiliaris, Konstantinos; Xu, Chao (2015-07-08). "A Faster Pseudopolynomial Time Algorithm for Subset Sum". arXiv:1507.02318 (https://arxiv.org/abs/1507.02318).

Further reading

- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "35.5: The subset-sum problem". *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7.
- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A3.2: SP13, pg.223.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Subset_sum_problem&oldid=773867043"

Categories: Weakly NP-complete problems | Dynamic programming

- This page was last edited on 2017-04-05, at 05:48:56.
-