A pseudo-polynomial time algorithm for Subset-Sum

Periklis A. Papakonstantinou
York University

Up to now we were discussing algorithms that no matter if they were exact or approximation algorithms, in both cases they were running in (worst-case) polynomial time with respect to the input length (e.g. when the input is measured as the number of bits). In this set of notes we introduce a Dynamic Programming algorithm that runs in exponential time (wrt the number of bits in the input) but it is the “milder” type of exponential-time algorithms we can find. When reading this document keep in mind that this algorithm is exact and it solves a computationally intractable (NP-complete) problem. Therefore we couldn’t expect to present an exact polytime algorithm for this problem. Lets give a not-so-formal definition of a pseudopolynomial time algorithm.

Definition 1. An algorithm that takes an input of \( n \) elements (some of these elements can be non-negative integers) is a pseudo-polynomial time (or pseudopolytime) algorithm if it runs in time which is bounded above by a polynomial in \( n \) and a polynomial in \( W \), where \( W \) is the sum of the elements in the input.

Note that there is no real harm instead of \( W \) to consider the largest integer in the input. The reason is that the sum of the \( n \) elements in the input is at most \( n \) times bigger than the maximum integer.

1 The problem

Problem: Subset-Sum-Equal

Input: \( S = \langle w_1, w_2, \ldots, w_n \rangle \), \( w_i \in \mathbb{N} \) and \( B \in \mathbb{N} \).

Output: Does there exist \( S' \subseteq S \) such that \( \sum_{a \in S'} a = B \)? (if yes output YES, if no output NO).

We do not prove the following theorem.

Theorem 1. There does not exist a polynomial time algorithm for Subset-Sum-Equal, unless \( P = NP \).

Shortly, we are going to show that despite the fact that Subset-Sum-Equal is a (provably) computationally hard problem there is a correct algorithm for this problem which runs in pseudopolytime. This means that for some inputs (perhaps among those there are many of practical interest) this algorithm can work in polytime. For example, suppose that for some

* This document is incomplete and potentially contains typos. Its purpose is to rapidly disseminate lecture notes for an undergraduate course in the theory of combinatorial algorithms. Please, contact papakons@cs.yorku.ca for corrections, remarks and suggestions.
reason we are interested in the following restriction of Subset-Sum: the value of every integer in the input is bounded above by \( n^3 \), where \( n \) is the number of elements. If for example, we have 1000 integers then all of them have must value less than or equal to 1000000000. Since \( n^3 \) increases with \( n \), for larger inputs we can tolerate even bigger values of the numbers involved. It turns out that there are many practical applications out there that not all of them are interested in solving the general version of the Subset-Sum (as given above). If for example the \( n^3 \) bound is true then for these specific inputs a pseudopolynomial time algorithm runs in polynomial time! This is because the pseudopolytime algorithm runs in time polynomial in \( W \), say \( W^k \) and if \( W \) is bounded by a polynomial in \( n \), say \( n^m \) then the whole algorithm (in terms of input length) runs in time \( n^{mk} \), where \( m \) and \( k \) are constants.

2 The algorithm

We are presenting a pseudopolytime algorithm for Subset-Sum-Equal (or Subset-Sum for short). We do not expect to find a polytime algorithm for an NP-complete problem. Unlike other DP algorithms the number of the subproblems is far from being polynomially many (wrt the input length).

Exercise 1. For a given input try to define “natural” subproblems that in such a way where their total number is a polynomial to the input length. After defining them, identify which exactly is the part that makes it impossible to use the optimal solutions from previous subproblems in order to efficiently build-up optimal solutions for bigger ones.

Remark 1. The reason why we insist on referring to solutions to “sub-problems” instead of saying “sub-inputs” (for the given problem) is because not every Dynamic Programming algorithm works by just truncating the input. For example, the input of the Subset-Sum consists of two parts, a sequence of integers and an integer. The following algorithm considers part of the input sequence, but as far as it concerns \( B \) it considers a “subcase” of \( B \). Notice that it seems weird to split \( B \) somehow (in a way that results in a subproblem that its solutions can be efficiently extended to solutions of bigger subproblems of the same nature).

2.1 Subproblems

\((i, w)\)-subproblem

Input: \( S = \langle w_1, w_2, \ldots, w_i \rangle \), \( w_i \in \mathbb{N} \) and \( w \in \mathbb{N} \), \( 1 \leq w \leq B \).

Output: Does there exist \( S' \subseteq S \) such that \( \sum_{a \in S'} a = w \)? (If yes output YES, if no output NO).

Therefore we have a family of \( nB \) subproblems. Potentially, this is a big number since (potentially) \( B \) is exponentially bigger than \( n \) (which is the number of elements in the input of the initially given problem).
Remark 2. We do not define subproblems for arbitrary reasons. It must be the case that as we are getting solutions to bigger subproblems the “closer” we are to answer the initially given problem. The above definition satisfies this intuitive requirement. The \((n, B)\)-th subproblem is identical to the given problem.

2.2 Table

In this case the table is a two-dimensional array containing boolean entries. The coordinates of the table are: the columns are indexed from 0 to \(B\) and the rows are indexed from 0 to \(n\). We refer to entries in the table as \((\text{row}, \text{column})\). Each entry in the table is true iff the solution to the corresponding subproblem is yes.

2.3 Recurrence relation associating entries of the table

Here is how to think about this: suppose that you are in the middle of the computation of the algorithm (that you haven’t already devised). How are you going to use solutions to previous subproblems in order to solve a bigger one (imagine that you are just trying to figure out how to solve just this bigger subproblem using the smaller ones). Thinking of this you get the following recurrence relation. Let the table be \(M\). Then,

\[
M[i, w] = M[i - 1, w] \lor M[i - 1, w - w_i]
\]

Remark 3. Note that the description of the algorithm is transparent in the above recurrence relation.

2.4 Algorithm

As every DP-algorithm, this algorithm uses the recurrence relation to fill-in the table. We start by setting everything to \textit{False} but the entry \(M[0, 0] = \text{True}\). For convenience consider references to negative indices in the array to return false. For example, \(M[-1, 10] = \text{False}\).

\textsc{Subset-Sum-Solver}[\(S = \langle w_1, w_2, \ldots, w_n \rangle, B\)]

1. Initialize \(M[0..n, 0..B]\) everywhere \textit{False} apart from \(M[0, 0] = \text{True}\)
2. for \(i \leftarrow 1 \text{ to } n\)
   do
3. for \(w \leftarrow 0 \text{ to } B\)
   do
4. \hspace{1cm} \(M[i, w] = M[i - 1, w] \lor M[i - 1, w - w_i]\)
   \hspace{1cm} (any reference outside the array returns false)
5. Output \(M[n, B]\)

The running time of this algorithm is \(O(nB)\). Therefore, in general exponential to the length of the input.
Remark 4. Note that there is no need to keep the whole $M[\cdot]$ during the execution. In each iteration $i$ we only have to keep rows $i - 1$ and $i$.

3 Proof of correctness

The algorithm contains two for-loops and thus it terminates by standard arguments.

It remains to show partial correctness. Suppose that the input is in the correct form. We can proceed by double induction. Since we haven’t discussed this method we will simulate it by (regular) induction. We do induction on the number of iterations of the two loops.

Better: we do induction on the number of the times where line 5 is executed. Consider the following predicate:

$P(j)$: at the end of the $j$-th iteration, $j$ corresponds to (unique) $i’$ and $w’$. Then, $M[i’, w’]$ contains the value of the correct solution (true or false) to the $(i’, w’)$-subproblem.

We do complete induction on $j$.

(Basis) When $j = 1$ then btwtaw $M[1, 0] = True$ which is true (since any set has as a subset the empty set which corresponds to zero sum), therefore $P(0)$ holds. (Induction step) Suppose that for every $h < j$ it holds that $P(h)$ is true. Wts (want to show) that $P(j)$ is true. Say that the $j$-th iteration corresponds to $i”, w”$. Consider the correct solution Correct for the $(i”, w”)$. If the algorithm agrees on the value with Correct (which is either true or false), i.e. $M[i”, w”] = Correct$ then $P(j)$ holds. Else, the algorithm disagrees with Correct.

Say that Correct = True and $M[i”, w”] = False$. Therefore, by the way the algorithm works $M[i” – 1, w”] = False$ and $M[i” – 1, w” – w_{i’}] = False$. But then, if the correct value is true then there exists a subset of $S'$ that sums up to $w”$. If $w_{i’} \notin S'$ then this contradicts the I.H. since $M[i” – 1, w”] = False$ and by the I.H. we know that this is a correct value.

Else, if $w_{i’} \notin S'$, removing it from $S'$ we should be able to get the sum $w – w_{i’}$ which again contradicts the induction hypothesis.

It remains to check the case that Correct = False and $M[i”, w”] = True$; and show that this is also impossible to happen. This case is the symmetric to the previous one.

Exercise 2. Complete the second case. It would be nicer to express this a bit better than the boring exhaustive enumeration of the three cases for True, False regarding and $M[i” – 1, w”] and M[i” – 1, w” – w_{i’}]$ (where the OR becomes true).

4 Extensions to the algorithm

What if we wanted to solve a different problem? What if instead of this problem we wanted to solve the problem of outputting an actual set $S'$ (if any) that sums up to $B$? There are in general two ways of accomplishing this in Dynamic Programming algorithms. Perhaps the most straightforward one is to keep a few more information in the table. That is, apart from the True/False value we keep the elements that correspond to a true value.
Remark 5. Note that the “heart” of Dynamic Programming algorithm does not deal with sets corresponding to solutions (as “opposed” to the values of solutions). The DP algorithm utilizes the values of the optimal sub solutions to make extensions, comparisons etc. The actual solutions (i.e. not only their values) come as a straightforward modification of the algorithm.

Instead of keeping in the table entries of the form (True/False, set of integers) we can just keep one more table which corresponds to sets of integers in the corresponding array entries.

\[ S = \langle w_1, w_2, \ldots, w_n \rangle, B \]

1. Initialize \( M[0..n, 0..B] \) everywhere False apart from \( M[0, 0] = True \)
2. Initialize \( M'[0..n, 0..B] \) everywhere \( \emptyset \) (i.e. each entry in the table is the empty set)
3. for \( i \leftarrow 1 \) to \( n \)
   do
4.      for \( w \leftarrow 0 \) to \( B \)
       do
5.          \( M[i, w] = M[i - 1, w] \lor M[i - 1, w - w_i] \) 
       (any reference outside the array returns false)
6.          if \( M[i, w] \neq M[i - 1, w] \)
            then
7.               \( M'[i, w] \leftarrow M'[i - 1, w - w_i] \cup \{w_i\} \)
            else
8.               \( M'[i, w] \leftarrow M'[i - 1, w] \)
4. Output \( M'[n, B] \)

Formally, we have to argue why this algorithm is correct for the new problem. But, this is mostly a technical and boring issue without much interest in terms of arguments.

Another way to accomplish the same task has to do with the following question. Given the whole \( M \) computed by the previous (not the one just above) algorithm Subset-Sum-Solver, if the output is True can we compute a set \( S' \) without doing what we did in Subset-Sum-Solver-Extended (i.e. recomputing \( M \))? Well, this question is not very well posed (since it cannot properly define what things are restricted in this new algorithm).
To avoid technicalities we will stay at this intuitive level.

Exercise 3. Devise this algorithm. You don’t need to give a proof of correctness.

Here is a sketch of how this algorithm works: it takes as input \( M \) and \( S = \langle w_1, w_2, \ldots, w_n \rangle, B \). If \( M[n, B] \) is false it outputs the empty set. If it is true then it checks to find the smaller \( j \) such that \( M[j, B] \) is true. Now we are certain that \( w_j \) has to be added. Then it continues in the same manner in the column \( B - w_j \) and so on.