Subset Sum Algorithms Report

Author: Dylen Greenenwald

Source Code

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

// Constant for infinity; used to initialize table entries
// which may not be feasible for min cardinality
const unsigned long long int INF = 99999999999999;

struct ssum_elem
{
  unsigned int x;   // value of element
  std::string name; // label for element
};

// Data corresponding to an entry in the ssum_table
// Each entry encodes this data for a target sum of i, x
// where i is the index of the last element considered and
// x is the target sum
struct ssum_data
{
  bool feasible;                             // feasibility of entry achieving target sum
  bool include;                              // flag indicating inclusion of entry in lexi first sset
  unsigned long long int no_v_ssets;          // number of valid distinct subsets for entry
  unsigned long long int min_card;            // min card of valid distinct subsets for entry
  unsigned long long int no_v_ssets_min_card; // number of valid distinct subsets of min card for entry
};

class ssum_instance
{
private:
  unsigned int target = 0; // Overall target sum for this instance
  int done = false;        // Flag indicating whether table has already been populated

  // dynamic programming table which stores interesting ssum data
  std::vector<std::vector<ssum_data>> ssum_table;

public:
  std::vector<ssum_elem> elems;

  // Function:  read_elems
  // Description:  reads elements from stdin, returns num els;
  // Format:  sequence of <number> <name> pairs where
  //      <number> is non-negative int and
  //      <name> is a string associated with element
  void read_elems(std::istream &stream)
  {
    ssum_elem e;

    elems.clear();
    while (stream >> e.x && stream >> e.name)
    {
      elems.push_back(e);
    }
    done = false;
  }

  // Function:  solve
  // Desc:  populates dynamic programming table of
  //        calling object for specified target sum.
  //        Returns true/false depending on whether the
  //        target sum is achievable or not.
  //        Table remains intact after call.
  //        Object can be reused for alternative target sums.
  ssum_data solve(unsigned int tgt)
  {
    unsigned int n = elems.size();
    unsigned int i, x;

    // Reverse the elements to extract the lexicographically first subset
    std::reverse(elems.begin(), elems.end());

    // If target sum is previously requested one and table has already been populated, ret result
    if (target == tgt && done)
      return ssum_table[n - 1][tgt];

    // Otherwise, reassign class members to prepare for new table population
    target = tgt;
    ssum_table = std::vector<std::vector<ssum_data>>(n, std::vector<ssum_data>(target + 1, {false, false, 0, INF, 0}));

    // Column base cases
    for (i = 0; i < n; i++)
    { // N runtime

      // All TRUE because a target sum of zero is
      // always acheivable (via the empty set).
      ssum_table[i][0].feasible = true;

      // All FALSE because no element (in Z+) can be
      // included to compose a target some of 0.
      ssum_table[i][0].include = false;

      // The empty set is a set, so there is 1 valid
      // subset which compose a target sum of 0.
      ssum_table[i][0].no_v_ssets = 1;

      // The empty set has a cardinality of 0.
      ssum_table[i][0].min_card = 0;

      // The empty set is unique, so it is the only
      // valid subset of cardinality 0.
      ssum_table[i][0].no_v_ssets_min_card = 1;
    } // otherwise, ssum_table[i][0] init vals remain

    // Row base cases
    for (x = 1; x <= target; x++)
    { // T runtime

      // We only care about the case where the first
      // element is exactly the target sum.
      if (elems[0].x == x)
      {
        // The first element IS the target sum, so it is feasible.
        ssum_table[0][x].feasible = true;
```

```cpp

        // The first element IS the target sum, so has the potential
        // to be included in the lexicographically first min card subset.
        ssum_table[0][x].include = true;

        // The set containing the first element is a set, so
        // there is 1 valid subset which composes the target sum.
        ssum_table[0][x].no_v_ssets = 1;

        // The set containing the first element has a cardinality of 1.
        ssum_table[0][x].min_card = 1;

        // The set containing the first element is unique, so it is the
        // only valid subset of cardinality 1.
        ssum_table[0][x].no_v_ssets_min_card = 1;
      }
    } // otherwise, ssum_table[0][x] init vals remain

    // Populate the rest of the table using the recursive cases
    for (i = 1; i < n; i++) // NT runtime
    {
      for (x = 1; x <= tgt; x++)
      {
        // Exclude case
        if (ssum_table[i - 1][x].feasible)
        {
          // If the target sum can be composed without the new element, values carry over
          ssum_table[i][x].feasible = true;
          ssum_table[i][x].no_v_ssets = ssum_table[i - 1][x].no_v_ssets;
          ssum_table[i][x].min_card = ssum_table[i - 1][x].min_card;
          ssum_table[i][x].no_v_ssets_min_card = ssum_table[i - 1][x].no_v_ssets_min_card;
        }
        // Include case
        if (x >= elems[i].x && ssum_table[i - 1][x - elems[i].x].feasible)
        {
          ssum_table[i][x].feasible = true;

          // # of valid ssets is the sum of the # of valid ssets in the exclude and include cases// Note that # valid ssets in exclude case is already stored in exclude case, so we accumulate
          ssum_table[i][x].no_v_ssets = ssum_table[i - 1][x].no_v_ssets
                                      + ssum_table[i - 1][x - elems[i].x].no_v_ssets;

          // min card is min btween exclude and include cases -- if one is infeasible, other is min (INF const)
          ssum_table[i][x].min_card = std::min(ssum_table[i - 1][x].min_card,
                                               ssum_table[i - 1][x - elems[i].x].min_card + 1);

          // Smallest valid size has either not changed, decreased, or increased bc of new el
          if (ssum_table[i - 1][x].min_card == ssum_table[i - 1][x - elems[i].x].min_card + 1)
          {
            // # of valid ssets of min card is the sum of the # of valid ssets in the exclude and include cases
            // since they are both of min card
            ssum_table[i][x].no_v_ssets_min_card = ssum_table[i - 1][x].no_v_ssets_min_card
                                                 + ssum_table[i - 1][x - elems[i].x].no_v_ssets_min_card;

            // new el may be included since min card would not change as a result of its inclusion
            ssum_table[i][x].include = true;
          }
          else if (ssum_table[i - 1][x].min_card > ssum_table[i - 1][x - elems[i].x].min_card + 1)
          {
            // # of valid ssets of min card is the # of valid ssets in the include case since it is the
            // only one of min card
            ssum_table[i][x].no_v_ssets_min_card = ssum_table[i - 1][x - elems[i].x].no_v_ssets_min_card;

            // new el may be included since min card would decrease as a result of its inclusion
            ssum_table[i][x].include = true;
          }
          else if (ssum_table[i - 1][x].min_card < ssum_table[i - 1][x - elems[i].x].min_card + 1)
          {
            // # of valid ssets of min card is the # of valid ssets in the exclude case since it is the
            // only one of min card
            ssum_table[i][x].no_v_ssets_min_card = ssum_table[i - 1][x].no_v_ssets_min_card;

            // NOTE: new el may not be included since min card would increase as a result of its inclusion
          }
        }
      }
    }
    done = true;
    return ssum_table[n - 1][target];
  }

  // Function:  extract
  // Desc:  populates first satisfying subset of minimum
  //        cardinality which occurs lexicographically first
  //        according to the indices of the input elements.
  std::vector<int> extract()
  {
    std::vector<int> lexi_first;

    // start the extraction from the last element in the table
    int i = elems.size() - 1;
    int x = target;

    // until we reach a base case ...
    while (x > 0)
    {
      if (ssum_table[i][x].include)
      {
        // add first occurring indices that have been marked
        // for min card inclusion
        lexi_first.push_back(i);

        // locate the next element to extract by following the
        // include case
        x -= elems[i].x;
      }
      i--;
    }

    // manually reverse elements according to mapping offset
    for (int &ind : lexi_first)
      ind = elems.size() - 1 - ind;

    // automatically reverse elements to restore original order
    // for class user
    std::reverse(elems.begin(), elems.end());

    return lexi_first;
  }
}; // end class

/**
* usage:  ssum  <target> < <input-file>
*
*
* input file format:
*
*     sequence of non-negative-int, string pairs
*
*     example:

    12 alice
    9  bob
    22 cathy
    12 doug

* such a file specifies a collection of 4 integers: 12, 9, 22, 12
* "named" alice, bob, cathy and doug.
*/
int main(int argc, char *argv[])
{
  unsigned int target;
  ssum_instance ssi;

  // Ensure proper command line usage
```

```cpp
262   if (argc != 2)
263   {
264     fprintf(stderr, "one cmd-line arg expected: target sum\n");
265     return 0;
266   }
267   if (sscanf(argv[1], "%u", &target) != 1)
268   {
269     fprintf(stderr, "bad argument '%s'\n", argv[1]);
270     fprintf(stderr, "   Expected unsigned integer\n");
271     return 0;
272   }
273
274   ssi.read_elems(std::cin);                             // Read in elements from stdin
275   ssum_data result = ssi.solve(target);                 // Get reference to last entry in dp ssum table
276   std::vector<int> lexi_first_min_card = ssi.extract(); // Extract lexi first min card subset
277
278   // Report all discovered data
279   std::cout << "### REPORT ###" << std::endl;
280   std::cout << "  NUM ELEMS            : " << ssi.elems.size() << std::endl;
281   std::cout << "  TARGET               : " << target << std::endl;
282   std::cout << "  NUM-FEASIBLE         : " << result.no_v_ssets << std::endl;
283   std::cout << "  MIN-CARD-FEASIBLE    : " << result.min_card << std::endl;
284   std::cout << "  NUM-MIN-CARD-FEASIBLE: " << result.no_v_ssets_min_card << std::endl;
285
286   std::cout << "Lex-First Min-Card Subset Totaling " << target << ": " << std::endl;
287   std::cout << " {" << std::endl;
288   for (int i = 0; i < lexi_first_min_card.size(); i++)
289   {
290     // Aliases for legibility
291     int id = lexi_first_min_card[i];
292     int val = ssi.elems[id].x;
293
294     std::string name = ssi.elems[id].name;
295     std::cout << "  " << name << "  ( id: " << id << "; val: " << val << ")"
296               << std::endl;
297   }
298   std::cout << " }" << std::endl;
299 }
```

Explanations

I would like to preface all of these explanations by stating that my overall approach to the entire problem was to simply extend the table to accommodate more data. As such, for the raw statistics (the first 2/3 below), there are both base cases and recursive cases to consider in their computation.

Additionally, note that each of the table's entries are initialized with values that can only be overwritten; in other words, they will never go "in the other direction" for whatever metric is currently being examined.

Distinct Subset Quantity Computation

Base cases:

- The first **column** of the dynamic programming table (as we established with the simple T/F feasibility) corresponds to a target sum of 0 for some prefix of input elements up to i for i <= n (with 1 based indexing). In all of these cases, the number of distinct subsets which can compose the target sum is 1. This is because the only set (the empty set) which can compose a target sum of 0 (from an input of positive integers) is independent of the values of the elements themselves. See `line 98` from section 1 and the comments above.
- The first **row** of the dynamic programming table corresponds to a varying target and an array consisting of the first element provided by the user. The target sum can only be composed by a singleton element if it is itself equal to the target. If this is the case, then there is exactly one valid subset which composes that target sum: the set containing the index of that element. See `line 125` from section 1 and the comments above.

Recursive cases:

- If it is feasible to compose a given target sum without the newest element, then it must be the case that the number of distinct subsets totaling the target is at least as large as it was without that element. In other words, if the newest element doesn't contribute in any way to the feasibility of the target sum (the "exclude" case), the number of distinct subsets that total the target sum in consideration the new element will remain equal to the number before consideration. See `line 146`; the number of distinct subsets (the value, in this case, as indicated by the above comment) will carry over from the prefix that excludes that element.

- It follows straightforwardly from this notion that if the newest element *does* contribute in some way to the feasibility of the target sum (the "include" case), that there will be more distinct subsets that result from its inclusion. In fact, there will be exactly as many more distinct subsets as can be computed by looking up previously populated values in the table that take into account the value at the newest index. Refer to `line 157` to see this computation. Notice that this computation only occurs if it's feasible to include the newest element in the input sequence, as defined by `line 151`. To explain this intuitively, since the newest element has a new index, any valid subset that includes its index will be distinct by our definition -- regardless of whether or not the values in these new sets match previously existing subsets that achieve the target sum.

Minimum Sized Subset Computation

Base cases:

- The first **column** refers to prefixes that compose a sum of 0. Since this can only be done by the empty set and the empty set has a cardinality of 0, any such prefix must have a minimum cardinality of 0 for a target sum of 0. See `line 101`, its comment and its enclosing loop.
- The first **row** refers to varying target sums, and seeing if they can be composed by the first element in the sequence. If this is the case, then the set containing such a singleton has a cardinality of 1. Otherwise, the minimum cardinality remains at its initial value of "infinity" to indicate that the set cannot be composed by any number of elements. See `line 128`, its comment, and its enclosing branch and loop.

Recursive cases:

- Since we have initialized all values of the table to infinity (see `line 11` and `line 82` for my implementation of this concept), we do not have to clutter our code with additional branching based on feasibility of a particular case.
- If we are dealing with the exclude case, the new element does not contribute to any of the data, so the minimum cardinality remains the same as the input sequence right up to before the new element occurs. See `line 147`.
- If we are dealing with the include case (but also, generally), there are 3 possibilities:
    1. The "previous" (excluding the new element) minimum cardinality of any satisfying subset is exactly equal to the minimum cardinality of any satisfying subset which includes the element. See the conditional on `line 164`.
    2. The "previous" (excluding the new element) minimum cardinality of any satisfying subset is strictly greater than the minimum cardinality of any satisfying subset which includes the element. See the conditional on `line 174`.
    3. The "previous" (excluding the new element) minimum cardinality of any satisfying subset is strictly less than the minimum cardinality of any satisfying subset which includes the element. See the conditional on `line 174`.

The new minimum cardinality is simply the minimum between the two cases. See `line 160`.

Lexicographically First Minimum Size Subset Extraction

The approach to this problem was different than the raw statistics above, so I will preface it before diving in.

The first thing we might observe is that if we start at the "end" entry (the entry corresponding to the full input sequence and whole target sum) and trace back through the table while biasing ourselves toward only choosing elements whose indices exist in minimum cardinality subsets, then we will always have a set of indices which correspond to the lexicographically *last* minimum sized subset. It follows that if we reverse the order of the input sequence that we will reverse the lexicographic ordering of indices which compose minimum cardinality subsets as well, thus yielding the lexicographically *first* subset of the original input sequence. This reversal will be the first operation of the `solve` member function of the class; see `line 74`. This is the big picture idea, the rest is left to implementation details.

The data which this approach cruxes on is a flag which will indicate to our extraction algorithm whether or not a particular index should be included in the set it returns. We can add this as a property of the existing `ssum_data` structure -- call it `includes`. It is initialized to false and changed as necessary. See `line 82`.

Dynamic programming table flag population...

Base cases:

- Since no positive integer can be included in a set which composes a sum of 0, it must be the case that the entire left column of the table has this flag set to false. This is redundant in the code, but good for legibility. See `line 94` and related comment.
- If the first element in the reversed sequence is exactly the target sum, then that must be included in the lexicographically first subset since its minimum cardinality is 1. Else, the flag remains false. See `line 121`.

Recursive cases:

- The main question here is, "when do we want to include an element?" The answer is when it belongs to a set of overall minimum cardinality. Thus, we only want to set this flag in the include case of the nested loop. However, we don't want to set it *any* time we can include a particular element in a satisfying subset since it has to be of minimum cardinality. Thus, the solution is to only set this flag to be true when the new element's inclusion in the subset will result in either a new, smaller minimum cardinality compared to what has previously been discovered **or** when its inclusion will result in a maintained minimum cardinality. In other words, we only want to include the new element when its inclusion in a particular subset results in its cardinality being less than or equal to what it previously was before this element's consideration. See `line 164` to `line 190`, specifically taking note of `line 172`, `line 181`, `line 189` and related comments.

At this point, we have everything we need to extract the lexicographically first subset from the original sequence. However, because of my chosen implementation, there are some annoying kinks to work out.

Now that these flags are populated, we can perform our traceback. We start at the bottom right entry, referring to the result of the overall problem. We will "pop off" any elements marked with the include flag, starting at the original target and continuing until the target sum is exactly 0. Note that it would have been equivalent on `line 211` to put `while (x != 0)`, but the existing version is easier to follow based on the decrementation of the intermediate target. We will shorten the reversed input sequence element by element (see `line 223`), until we have pushed all of our indices into our vector that represents the lexicographically first subset.

After the while loop ends, we will be left with a vector of indices that encode the lexicographically first starting from the end of the reversed sequence. In order to normalize the indices for our report, we must take care of the offset imposed by the reversal. To do this, we subtract whatever index we got from the size and then also subtract 1 from that (to take care of the 0 based indexing implementation). Finally (although, in retrospect, might have been more appropriate elsewhere), we reverse the input sequence to its initial order before returning the lexicographically first subset.

Data

Run 1

```
→ src (main) ✗ ./ssum 269 < electoral.txt
### REPORT ###
  NUM ELEMS              : 51
  TARGET                 : 269
  NUM-FEASIBLE           : 16976480564070
  MIN-CARD-FEASIBLE      : 11
  NUM-MIN-CARD-FEASIBLE: 1
Lex-First Min-Card Subset Totaling 269:
 {
  CA  ( id: 4; val: 55)
  FL  ( id: 9; val: 29)
  GA  ( id: 10; val: 16)
  IL  ( id: 13; val: 20)
  MI  ( id: 22; val: 16)
  NY  ( id: 32; val: 29)
  NC  ( id: 33; val: 15)
  OH  ( id: 35; val: 18)
  PA  ( id: 38; val: 20)
  TX  ( id: 43; val: 38)
  VA  ( id: 46; val: 13)
 }
```

Run 2

```
→ src (main) ✗ ./ssum 220 < purple.txt
### REPORT ###
  NUM ELEMS               : 31
  TARGET                  : 220
  NUM-FEASIBLE            : 9958625
  MIN-CARD-FEASIBLE       : 13
  NUM-MIN-CARD-FEASIBLE: 57
Lex-First Min-Card Subset Totaling 220:
 {
  AZ  ( id: 0; val: 11)
  CO  ( id: 2; val: 9)
  FL  ( id: 5; val: 29)
  GA  ( id: 6; val: 16)
  IN  ( id: 7; val: 11)
  MI  ( id: 12; val: 16)
  MN  ( id: 13; val: 10)
  NJ  ( id: 19; val: 14)
  NC  ( id: 20; val: 15)
  OH  ( id: 21; val: 18)
  PA  ( id: 23; val: 20)
  TX  ( id: 26; val: 38)
  VA  ( id: 27; val: 13)
 }
```

Run 3

```
→ src (main) ✗ ./ssum 121 < purple.txt
### REPORT ###
  NUM ELEMS               : 31
  TARGET                  : 121
  NUM-FEASIBLE            : 9958625
  MIN-CARD-FEASIBLE       : 5
  NUM-MIN-CARD-FEASIBLE: 2
Lex-First Min-Card Subset Totaling 121:
 {
  FL  ( id: 5; val: 29)
  GA  ( id: 6; val: 16)
  OH  ( id: 21; val: 18)
  PA  ( id: 23; val: 20)
  TX  ( id: 26; val: 38)
 }
```