CS401 - Fun With Subset Sum
Due ~~Tuesday October 18~~ Thursday, Oct. 20 at 3:30PM

For this assignment you will implement a dynamic-programming algorithm for the subset-sum problem with a couple of wrinkles. Recall the formulation of the subset sum problem in its simplest form:

GIVEN: positive integers `a[1], a[2], ..., a[N]` , and target sum T

OUTPUT:

`true` if there is a subset $S \subseteq \{1..N\}$ such that $\sum_{i \in S}(a[i]) = T$

`false` otherwise.

ASSIGNMENT SUMMARY: you will write a program for the subset-sum problem which will report not only if the target sum can be formed, but also:

1. the number of distinct subsets yielding the target sum
2. the size of the smallest subset yielding the target sum
3. the number of distinct subsets yielding the target sum *and* are of minimum size.
4. the lexicographically first subset from (3)

THE BASIC ALGORITHM:
The idea behind a dynamic-programming algorithm for this problem is based on a table of booleans `S[i][y]` with dimensions

```
S[1..N][0..T]
```

The algorithm populates this table with the following interpretation:

```
S[i][y] = true if and only if a target sum of y can be
          formed by selecting elements from a[1]..a[i].

          (and false otherwise)
```

The table can be correctly populated by observing the following base-cases and recursive rules:

BASE CASES

```
S[i][0] = true   for all $i \in \{1..N\}$ (empty set has sum
zero).

S[1][a[1]] = true  (singleton)
S[1][y] = false  $\forall y \neq a[1]$
```

INDUCTIVE/RECURSIVE CASES:

```
(for i>1)
     S[i][y] = true if S[i-1][y] = true
                         OR
                (y >= a[i]) AND (S[i-1][y-a[i]] = true)

                (false otherwise)
```

Pseudo-code populating the table according to these rules follows:

```
for(i=1; i<=N; i++)
    S[i][0] = true
for(y=1; y<=T; y++) {
    if(y==a[1])
       S[1][y] = true
    else
       S[1][y] = false
}
for(i=2; i<=N; i++) {
    for(y=1; y<=T; y++) {
        if(S[i-1][y] || (a[i]<=y && S[i-1][y-a[i]]))
             S[i][y]=true
        else
             S[i][y]=false
    }
}
```

At the termination of this algorithm, there is a subset totaling `T` if and only if `S[N][T]==true`.

That’s nice, but we usually want more information that just “thumbs-up” or “thumbs-down”

MULTIPLE SOLUTIONS:

Suppose for some problem instance, there is indeed a subset totaling the target value `T`. In general, there may be many such subsets. Furthermore, this collection of subsets may vary in cardinality.

For example:

Suppose `a[] = {1, 1, 2, 3, 1, 4}` and `T=5`. There are (by my count), eight distinct subsets which add up to 5. They are enumerated in the table below (where an `'x'` indicates the element in that column is selected and a `'-'` indicating the opposite.

index i	1	2	3	4	5	6	subset size	index list
a[i]	1	1	2	3	1	4		
subset-1	x	x	x	-	x	-	4	{1, 2, 3, 5}
subset-2	x	x	-	x	-	-	3	{1, 2, 4}
subset-3	x	-	-	x	x	-	3	{1, 4, 5}
subset-4	x	-	-	-	-	x	2	{1, 6}
subset-5	-	x	-	x	x	-	3	{2, 4, 5}
subset-6	-	x	-	-	-	x	2	{2, 6}
subset-7	-	-	x	x	-	-	2	{3, 4}
subset-8	-	-	-	-	x	x	2	{5, 6}

The smallest such subsets are of size (cardinality) two and four of the eight are of this size. These are highlighted in green.

Furthermore, notice that the satisfying subsets are listed in *Lexicographic* order. See the rightmost column: each set is represented by the list of selected indices in order. We apply lexicographic ordering to these sequences.

INPUT FORMAT:

To make the program output a little more interesting, each element also has a *name* associated with it (as a string). Input files contain one element per line: the integer first, followed by an associated name (on the same line). For example, the first few lines of my file containing the number of electoral votes for each state plus the District of Columbia looks like this:

```
9 AL
3 AK
11 AZ
6 AR
55 CA
9 CO
```

NOTE: element ordering is *exactly* the order given by the input file: `a[1]=9, a[2]=3`, etc. Ordering has nothing to do with the element *names* (they just happen to be sorted in this example).

YOUR JOB:

Write a program solving the subset sum problem which reports the following (when there is at least one feasible solution):

1. The **number of distinct subsets** that yield a sum of `T` (of any size)
2. The **size of the *smallest* subse**t yielding `T`
3. The **number of distinct subsets** yielding sum `T` ***and* having minimum size** from (2).
4. Reports the ***lexicographically first* minimum-size subset** which yields the sum of T by listing its members in increasing order of the elements indices (if there is a solution).

(If there is no feasible subset, your program simply reports "INFEASIBLE").

TIP: the number of distinct subsets totaling a given target can be *very, very* large -- too large for regular old `int`s. Use **`unsigned long int`** for storing appropriate counters. (Even these will run out of gas eventually...)

RUNTIME: The runtime of your program must be asymptotically equivalent to that of the basic FEASIBLE/INFEASIBLE version described above.

EXAMPLE: The example instance from the previous page can be represented by the following file (called `toy.txt`):

```
1 dog
1 cat
2 bird
3 mouse
1 snake
4 fish
```

A sample run with a target `T=5` would produce something like this for output:

```


    Target sum of 5 is FEASIBLE!

    Number of distinct solutions:             8
    Size of smallest satisfying subset:       2
    Number of min-sized satisfying subsets:   4
    Lexicographically first min-sized solution:

                 {dog, fish}
$
```

ASSUMPTIONS:

This assignment isn't about string processing and parsing. So you may make some reasonable assumptions about the input:

- You do not need to verify that all of the element names are distinct.
- You may assume that the labels have length no greater than 10 characters.

Guiding principle: you are prototyping an algorithm implementation, not a deliverable piece of software...

PROGRAM BEHAVIOR:

Do something reasonable and include a `README` file!

Usage of my implementation is like this:

```
ssum <N_MAX> <TARGET>   <   <INPUT-FILE>
```

The first command-line argument specifies an upper-bound on N for the problem instance -- this lets me allocate my matrices before reading the data. I'm lazy and am following the above guiding principle.

The 2nd cmd-line argument is the target sum itself.

The program reads the elements `a[1]...a[N]` from std-input.

LANGUAGE:

You may implement your solution in any reasonably well known programming language of your choice.

However, we *must* be able to run it from the command-line on the CS servers (bert, ernie) and you must include instructions on compilation and usage (again, in the `README` file).

GIVEN FILES:

In the subdirectory, `src`, you have been given a C implementation of the bare-bones algorithm which simply reports `FEASIBLE` or `INFEASIBLE`. The file is named `ssum.c`. This program is completely self-contained. You are welcome to use this as a starting point or for reference.

In addition, there are three input files:

```
toy.txt
electoral.txt
purple.txt
```

`toy.txt` is the small example used previously in this handout.

`electoral.txt` contains 51 entries corresponding to the electoral votes for each state plus the District of Columbia.

purple.txt contains a subset of the 51 entries from `electoral.txt`: entries that are "safe Republican" or "safe Democrat" have been removed.

10 "safe Republican" entries have been removed totaling 49 electoral votes

10 "safe Democratic" entries have been removed totaling 121 electoral votes.

Thus, `purple.txt` contains 31 entries with a total of **`341`**

Both `electoral.txt` and `purple.txt` will play roles in your experiments and writeup.

REPORT:

You will submit a report on your work as a hardcopy in class. Your report will include the following:

1. **YOUR SOURCECODE**: If your program has gotten long, you may just include the key components needed to explain your solution (i.e., your data structures and bookkeeping, how they are populated and how solutions are extracted). You will submit your complete code electronically.

    You might format your source code with line numbers for easy reference in your subsequent explanations (below).

2. **EXPLANATION**: Explain your logic for computing the number of distinct subsets totaling the target. Include references to your actual code from (1).
3. **EXPLANATION**: Explain your logic for determining the minimum sized subset totaling the target. Including references to your actual code from (1)
4. **EXPLANATION**: Explain your logic for capturing the lexicographically first min-sized subset totaling the target. Include references to your actual code from (1).
5. **DATA**: Run your program on the electoral college data in **`electoral.txt`** with a target of **`T=269`** (exactly half of the grand total of **`538`** electoral votes). Cut and paste the results of your run (including all 4 key outputs)
6. **DATA**: Run your program on the partial electoral college data in **`purple.txt`** using a target of **`T=220.`** Recall that the "safe Republican" states totaled **`49`** electoral votes and that **`49+220=269.`**
   Cut and paste the results of your run (including all 4 key outputs)
7. **DATA:** Run your program on the partial electoral college data in **`purple.txt`** using a target of **`T=121.`** Recall that the "safe democratic" states totaled **`148`** electoral votes and that **`148+121=269.`**
   Cut and paste the results of your run (including all 4 key outputs)

SUBMISSION:

1. Create a single zip file of your source directory containing:
    - All of your source files
    - A README file explaining how to compile and run your program.
    - If compilation takes more than one step, you should also include a `makefile`

   You will submit this zip file through Blackboard.

2. In class, you will submit a hardcopy of the report described above.

FOR THE AMBITIOUS OR EASILY BORED (NOT EXTRA CREDIT):

- Try adding a feature to your program which enumerates the first **`K`** min-sized subsets totaling `T` in lexicographic order (instead of just listing the first). **`K`** could be a command-line argument to the program so the user can prevent huge large output when there are tons of satisfying subsets.
- Better yet, add a sort of “iterator” which allows the caller to walk through the min-sized subsets totaling **`T`** one-by one. For C programmers, think of how the utility function `strtok` works as a kind of model.