Piotr Ramza
CS 401
Project 1 Report

Part 1: Source Code

Below is a section of code from my ruby implementation of the algorithm. I included the sections which actually implement the algorithm but left out functions for printing elements and reading in the input from the file, since that is not what this project is about. However I will note that the values stored in each element in the @elems array has a .value and .name variable.
Note: This section of code spans lines 45 to 183 in my .rb file, but for the purposes of explaining concepts in this report I will be using the numbers shown below.

```
1.  def solve(tgt)
2.    n = @elems.length
3.
4.    # solution has already been run on this target
5.    if @target == tgt && @done
6.      @feasible.at(n-1).at(tgt)
7.    end
8.
9.    # fill variables based on input
10.   @target = tgt
11.   @feasible = Array.new(n) { |i| Array.new(@target + 1) { |i| nil } }
12.   @lastcount = nil
13.   @count = Array.new(@target, 0)
14.   @lastmincount = nil
15.   @mincount = Array.new(@target, 0)
16.
17.   # leftmost column (column zero) is all TRUE because
18.   #    a target sum of zero is always achievable (via the
19.   #    empty set).
20.   i = 0
21.   while i < n
22.     @feasible[i][0] = Array.new(0)
23.     i += 1
24.   end
25.
26.   # populate first row
27.   x = 1
28.   while x <= @target
29.     if @elems[0].value == x
30.       @feasible[0][x] = Array.new(1, 0)
31.       @count[x] = 1
```

```ruby
    @mincount[x] = 1
  end

  x += 1
end

# save the current count arrays in the last count arrays
# This allows for the idea of a 2d array but only saves
# 2 rows at a time, saving space
@lastcount = @count.clone
@lastmincount = @mincount.clone

i = 1
while i < n
  x = 1
  while x <= tgt

    # determines if we should stick with the array that is already in the column,
    # or if there is a better option to create that number by using a previous array
    # Also counts the number of possibilities for each number (ei. column)
    if x >= @elems[i].value && !@feasible[i-1][x-@elems[i].value].nil?
      newarray = Array.new(1, i)
      oldarray = @feasible[i-1][x-@elems[i].value].clone
      combinedarray = oldarray.concat(newarray)
      placearray(combinedarray, i, x)


    elsif !@feasible[i-1][x].nil?
      @feasible[i][x] = @feasible[i-1][x]

    end
    x += 1
  end
  @lastcount = @count.clone
  @lastmincount = @mincount.clone
  i += 1
end

# return the last element in the 2d array
```

```ruby
 # This value will be the shortest lexicographically first solution
 @done = true
 @feasible.at(n-1).at(@target)

end # end solve

def placearray(combinedarray, i, x)

 # if the value is = x then we need to check if the previous array was
also
 # a single length array which held x
 if @elems[i].value == x
   if !@feasible[i-1][x].nil?
     # if it was then we keep the lexicographically first instance
     # which is already in feasible
     if combinedarray.length == @feasible[i-1][x].length
       @feasible[i][x] = @feasible[i-1][x]
       @mincount[x] +=1
       # otherwise we change the contents of feasible to be
       # the smaller array (the new array)
     elsif combinedarray.length < @feasible[i-1][x].length
       @feasible[i][x] = combinedarray
       @mincount[x] = 1
     else
       @feasible[i][x] = @feasible[i-1][x]
     end
   else
     @feasible[i][x] = Array.new(1,i)
     @mincount[x] = 1
   end
     @count[x] += 1

  elsif !@feasible[i-1][x].nil?
     # length of combination is greater than the current length
     # so we keep the current array
     if combinedarray.length > @feasible[i-1][x].length
       @feasible[i][x] = @feasible[i-1][x]
       @count[x] = @lastcount[x-@elems[i].value] + @lastcount[x]

       # the new  combination is same length and lexicographically
first
       # so we use the new array
```

```
111.      elsif combinedarray.length == @feasible[i-1][x].length &&
112.          combinedarray[0] < @feasible[i-1][x][0]
113.        @feasible[i][x] = combinedarray
114.        @count[x] = @lastcount[x-@elems[i].value] + @lastcount[x]
115.        @mincount[x] = @lastmincount[x-@elems[i].value] + @lastmincount[x]
116.
117.        # the new  combination is same length and not lexicographically first
118.        # so we use the old array
119.      elsif combinedarray.length == @feasible[i-1][x].length
120.        @feasible[i][x] = @feasible[i-1][x]
121.        @count[x] = @lastcount[x-@elems[i].value] + @lastcount[x]
122.        @mincount[x] = @lastmincount[x-@elems[i].value] + @lastmincount[x]
123.
124.        # the length of combined array is less than current array
125.        # so we need to replace it with the combined array
126.      elsif combinedarray.length < @feasible[i-1][x].length
127.        @feasible[i][x] = combinedarray
128.        @count[x] = @lastcount[x-@elems[i].value] + @lastcount[x]
129.        @mincount[x] = 1
130.      end
131.      # the previous array was nil so we use the new possible array and set
132.      # count to the count of the combined array since there is a new possibility for that number
133.    else
134.      @feasible[i][x] = combinedarray
135.      @count[x] = @lastcount[x-@elems[i].value]
136.      @mincount[x] = @lastmincount[x-@elems[i].value]
137.
138.    end
139.  end
```

<u>Part 2: Distinct Subset Count</u>

To keep track of the possible solutions I made 2 arrays named @count and @lastcount. @count represents the number of possible subarray solutions for the current row, while @lastcount represents the number of possible subarray solutions for the last row. After we iterate through each row and move on to the next, @lastcount gets replaced with @count. (Line 65)

We start with a @count that is of length @target and is populated with 0's as a starting point. As we populate row 1 of @feasible on line 28 we add 1 to @count[column number] where the value of @elems[0] = column number since there is now 1 way to make that column number. Then we set @lastcount = @count.clone (Line 41) once we are done with the first row.

As we iterate through the rest of @feasible we check if there is a new candidate subarray (Line 52), if there is not then we simply move to the next element. However if there is a new candidate subarray then we step into the placearray method. Inside the placearray method:

1. If there is a new valid candidate subarray that adds to the current column, then we first check if the column number = the value of @elems[row] since this is a special case that only adds 1 to the @count[column] (Lines 81 and 100)
2. Otherwise, number at @count[current column] = @lastcount[current column] + @lastcount[current column - @elems[current row] (Lines 114, 121, 128, 135)
    a. The logic behind this is that if there is another feasible subarray that uses the current @elems value, there can be multiple possible ways to make that combination. For example if the current value is 4 and the target is 6 there may be multiple ways to make a 2 in @elems that would result in the target sum. We need to count all these possible subarrays so we go back in last count and look at the number of ways a 2 can be made, in turn this is all the ways that a 6 can be made when there is one 4 in the subarray (this is the @lastcount[current column - @elems[current row] part of the equation). However we also have to count all the other ways to make a 6 that we found previously, so we add this value to the @lastcount[current column] value to find the total number of ways to make that column number.

When we are done @count[@target] will hold the total number of distinct subsets that equal the target

The way to find the number minimum sized subarrays is similar but a few rules are changed in placearray to reset the counter when a smaller subarray is found.

Part 3: Determining Size of Smallest Subset

This process was actually very easy since all I had to do was find the size of the subset saved in the final entry of my 2D array @feasible, since the rule of @feasible is that the lexicographically first mid-sized subarray that is feasible will be stored at each entry of the 2D array and the final entry will hold the best subarray for the given target. This will be explained further in Part 4.

Part 4: Finding Lexicographically First Min-Sized Subset

The first change I made in my code compared to the sample code that we were given was to make the 2D array @feasilbe into a 2D array that would store arrays, essentially making it into a 3D array, instead of just storing true or false. The new rule for this 2D array was that each stored array would hold the indexes of the lexicographically first min-sized subarray that could be created until that point. The 2D array had @elems.length number of rows and @target number of columns (Line 11) where the rows denoted the values of the elements and the columns denoted all numbers from 0 to target.

The idea behind this new rule is that if a new subarray is found that is shorter than the previous best subarray then we can replace the old one with the new one, but if the new subarray is longer than the previous we simply keep the old one. If they are the same length then we check the first element of each subarray to see which is lexicographically first, if they are both the same then keep the old one since we are moving left to right and top to bottom.

I accomplished this by first populating each element in @feasible with nil (Line 11), which denoted that the column number cannot be created yet. I then populated the first column of @feasible with empty arrays, since a subarray of length 0 can always be created with an empty array (Lines 20 - 24). I then populated the first row by stepping through each column, and if the value of the first element = column number then that element of @feasible was set to [0], since that column number could be created using the @elems[0] (Lines 27 - 36).

I then stepped through the remaining elements in @feasible from left to right then top to bottom (Lines 44 - 47). As I stepped through, there were 2 options:

1. If the column number was greater or equal to than the current @elems value, and the element in @feasible[row-1][col - current @elems value] is not nil (Line 52)
    a. Then create a new subarray using the subarray stored in @feasible[row-1][col - current @elems value] concatenated with the current column number (Lines 53 - 55). Since the rule of @feasible is that it holds the lexicographically first min-sized subarray to create that column number this new array will be a new candidate for the lexicographically first min-sized subarray of the current column, but we need to check if it actually is, so we send the new array to the placearray function, along with the current row and column number (Line 56).
2. Else if the element above the current element in @feasible is not null, then we copy that value down into the current element (Line 59).
    a. If we got to this branch then that would mean there is no way for the current @elems value to create the column value, so we propagate the possible array down, or leave the current element as nil if there was no way to make the possible array in the first place.

<u>The placearray function:</u>

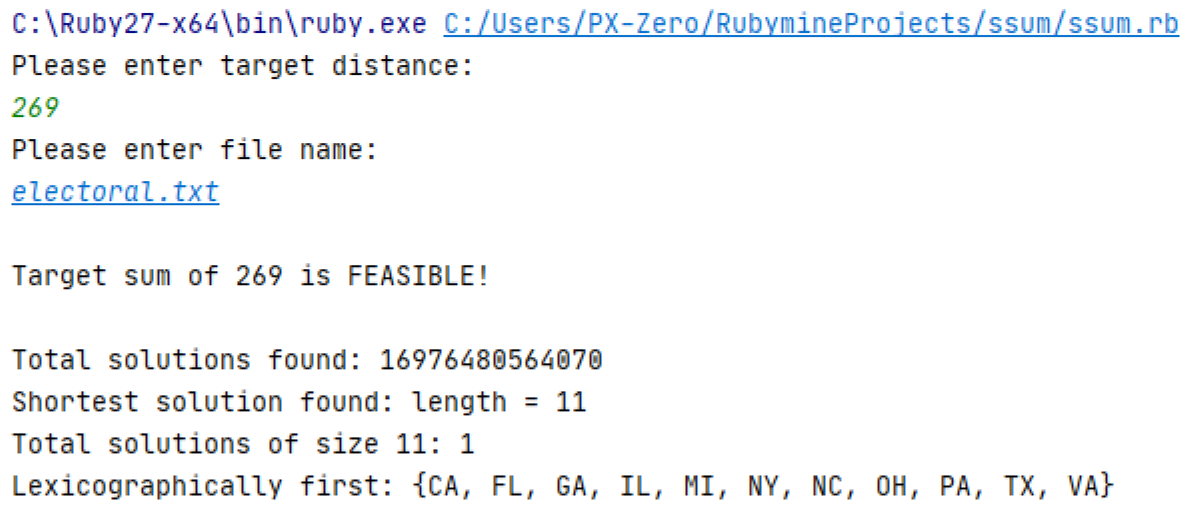
If we get to this function then we know there is a new candidate for shortest lexicographically first subarray, but we need to check if it is a better candidate than what is already in @feasible. We compare the length of the new candidate to the old subarray:

1. If the old subarray is nil, then we use the new subarray (Line 133)
2. Else if the old subarray is shorter than the new subarray then it is stored in the current element of @feasible (Line 105)
3. Else if they are the same size we find the lexicographically first by comparing the first element of the subarrays and choosing the smaller one, since the subarrays hold the indexes of the @elems (Lines 111 and 119)
4. Otherwise the old subarray is longer than the new subarray and the new array is stored (Line 126)

These checks ensure that the subarray stored at the current element in @feasible is the lexicographically first min-sized array that is possible until that point, allowing us to use these values to calculate the next.

Once we step through the entirety of @feasible, we are guaranteed to have the lexicographically first min-sized array at @feasible[@elems.length-1][@target]

Part 5:

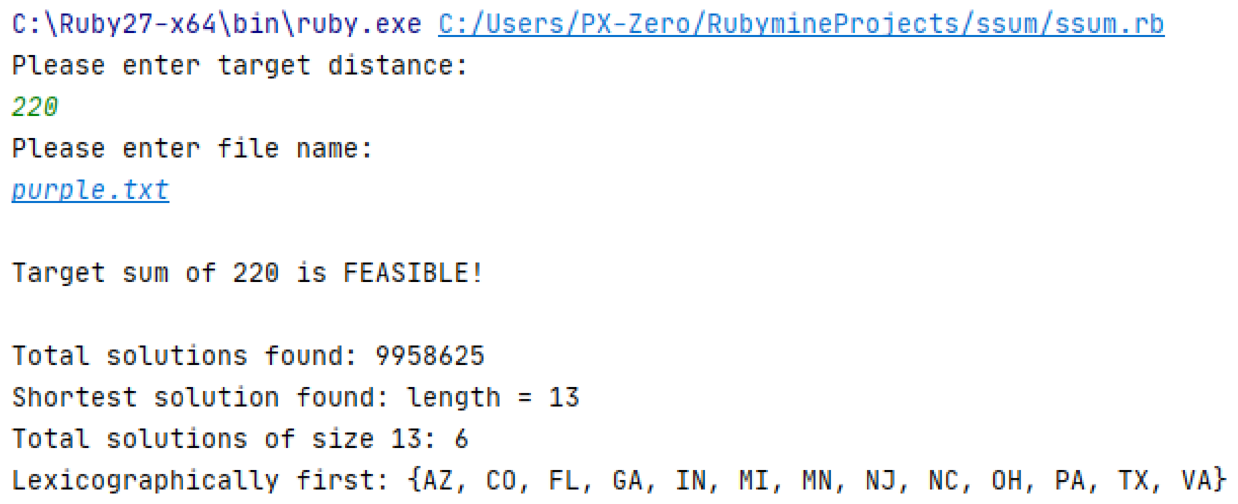


```
C:\Ruby27-x64\bin\ruby.exe C:/Users/PX-Zero/RubymineProjects/ssum/ssum.rb
Please enter target distance:
269
Please enter file name:
electoral.txt

Target sum of 269 is FEASIBLE!

Total solutions found: 16976480564070
Shortest solution found: length = 11
Total solutions of size 11: 1
Lexicographically first: {CA, FL, GA, IL, MI, NY, NC, OH, PA, TX, VA}
```

Part 6:

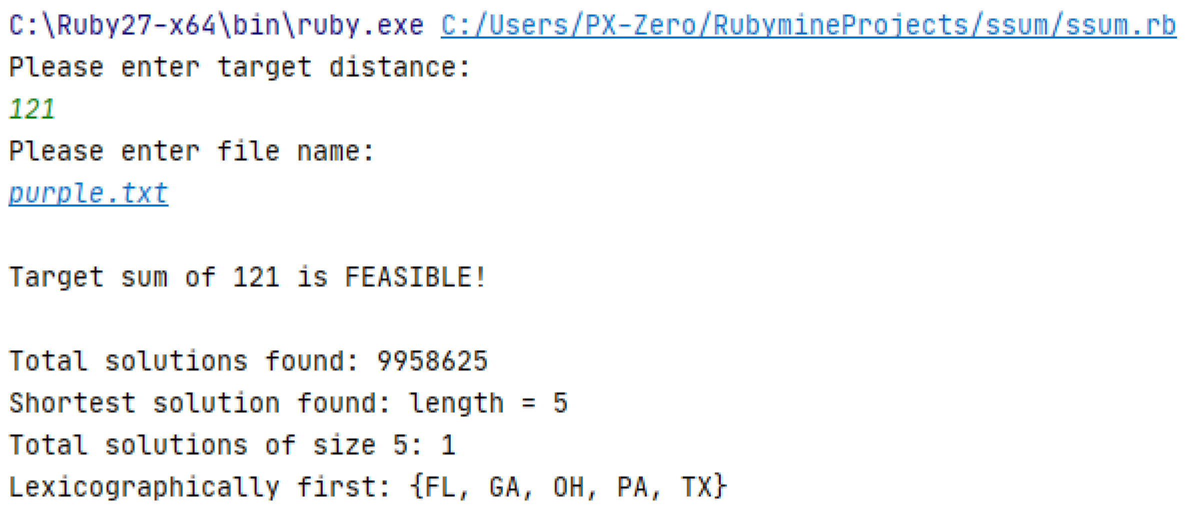


```
C:\Ruby27-x64\bin\ruby.exe C:/Users/PX-Zero/RubymineProjects/ssum/ssum.rb
Please enter target distance:
220
Please enter file name:
purple.txt

Target sum of 220 is FEASIBLE!

Total solutions found: 9958625
Shortest solution found: length = 13
Total solutions of size 13: 6
Lexicographically first: {AZ, CO, FL, GA, IN, MI, MN, NJ, NC, OH, PA, TX, VA}
```

Part 7:



```
C:\Ruby27-x64\bin\ruby.exe C:/Users/PX-Zero/RubymineProjects/ssum/ssum.rb
Please enter target distance:
121
Please enter file name:
purple.txt

Target sum of 121 is FEASIBLE!

Total solutions found: 9958625
Shortest solution found: length = 5
Total solutions of size 5: 1
Lexicographically first: {FL, GA, OH, PA, TX}
```