CISC/CMPE-365
Fall 2019

Week 4 AND Week 5 Lab Problem: Subset Sum

Due Date: October 12, 11:59 PM

This 2-week lab deals with the Horowitz/Sahni Algorithm for solving the Subset Sum problem in better than O(2^n) time.

You work for a large not-for-profit organization with many departments, each of which proposes projects for the year. Each project has a budget requirement, expressed as a number of dollars. The government provides funding to your organization, but the funds are never adequate to cover all the projects (this is an annoying real world situation), and there is an added restriction: you must use exactly the amount of money the government is giving you, or next year they will cut your funding (unfortunately, this is also true).

Your job is to decide which projects to fund. More precisely, you are given a set of positive integers, possibly containing duplicates, and a target figure, k. Your task is to find a set of integers in the original set that sum to exactly k, if such a subset exists.

For example, if the set of project costs is S = {3,5,3,9,18,4,5,6} and the government funding is 28, then an acceptable solution set is {5,18,5} ... and another acceptable solution is {3,3,18,4} ... and there are several more. You are only required to find one solution, or report that no solution exists (which would be the case if the given set were used and the funding figure changed to 52).

The naïve algorithm simply generates all subsets and computes the sum of each one. One way to run through all the subsets is this:

```
define a Set object which has attributes:
      elements – a list of the elements in the set
      sum – the sum of the elements in the set

let empty_set be a Set in which
      empty_set.elements is an empty list
      empty_set.sum = 0

let S = {s_1, s_2, ..., s_n}   be the set whose subsets we want to evaluate

let k be the target value   (assume k != 0)

BFI_Subset_Sum(S,k):

      let subsets be an empty list of Set objects

      add empty_set to subsets

      for i = 1 to n:
            let new_subsets be an empty list of Set objects
            for each subset old_u in subsets:
                  create a new Set object new_u with
                        new_u.elements = old_u.elements with s_i appended
                        new_u.sum = old_u.sum + s_i
                  if new_u.sum == k:
                        STOP – new_u is the solution; print new_u
                  else:
                        append old_u to new_subsets
                        append new_u to new_subsets
            subsets = new_subsets

      # no solution found
      print  "no subset sums to the target value"
```

A quick demo of this in operation: suppose S = {10,20,30}

```
subsets = [  (elements = {}, sum = 0) ]       #   just the empty set

# create more subsets by including 10 in each existing subset
subsets = [  (elements = {}, sum = 0) ,
             (elements = {10}, sum = 10) ]

# create more subsets by including 20 in each existing subset
subsets = [  (elements = {}, sum = 0) ,
             (elements = {20}, sum = 20) ,
             (elements = {10}, sum = 10),
             (elements = {10,20}, sum = 30) ]

# create more subsets by including 30 in each existing subset
subsets = [  (elements = {}, sum = 0) ,
             (elements = {30}, sum = 30) ,
             (elements = {20}, sum = 20),
             (elements = {20,30}, sum = 50),
             (elements = {10}, sum = 10),
             (elements = {10,30}, sum = 40),
             (elements = {10,20}, sum = 30),
             (elements = {10,20,30}, sum = 60) ]
```

Note: the use of objects (or classes) is not a requirement – it's just a convenient way to describe the organization of the information.

The algorithm presented in class – which I call the Horowitz/Sahni algorithm since their text is the earliest record of it that I know – looks like this:

let S = $\{s_1, s_2, \ldots, s_n\}$ be the set whose subsets we want to evaluate

let k be the target value   (assume k != 0)

```
HS_Subset_Sum(S,k):

    divide S into S_left and S_right, with n/2 elements in each (or as close as possible)

    use (modified) BFI_Subset_Sum  to get a list of all the subsets and their sums
    from S_left .   Call these Subsets_Left and Sums_Left

    do the same for S_right.   Call the results Subsets_Right and Sums_Right

    if k is in Sums_Left or Sums_Right:
        print the corresponding subset that sums to k
    else:
        sort Sums_Left
        sort Sums_Right
        use the Pair_Sum algorithm (see below) to search for a value x in
        Sums_Left and a value y in Sums_Right such that x + y = k
        if found:
            print the corresponding subsets from Subsets_Left and Subsets_Right
        else:
            print  "no subset sums to the target value"
```

```
Pair_Sum(Values_1, Values_2, k):
    # Values_1 and Values_2 are sorted
    # indexing starts at 1 because I am a dinosaur
    p1 = 1
    p2 = length(Values_2)
    while (p1 <= length(Values_1) and (p2 >= 1):
        t = Values_1[p1] + Values_2[p2]
        if t == k:
            return (p1, p2)
        else if t < k:
            p1 = p1 + 1
        else:
            p2 = p2 – 1
    return (-1, -1)
```

Part 1:

Implement and test both the `BFI_Subset_Sum` and `HS_Subset_Sum` algorithms. Testing can be done on a small set such as the one used in the example at the beginning of this document. Make sure you test for cases where the target value is in the set, where the target value is the sum of the entire set, and where there is no subset that sums to the target value.

Part 2:

Conduct experiments to explore the relative efficiency of the two algorithms.

For the purposes of comparing algorithms we must always decide which operations to count. It is typical to count only operations that involve accessing, comparing or moving data (as opposed to "administrative" operations such as incrementing counters).

For the BFI_Subset_Sum algorithm, the work consists of building the subsets and computing the sums. The operations here are easy to count.

For the HS_Subset_Sum algorithm a bit more thought is needed. The count must include

- the work done by the two calls to BFI_Subset_Sum
- the sorting of Sums_Left and Sums_Right
  - you can use a built-in sort function
  - you can assume that the sort function performs 3*t*(log t) operations when sorting a set of size t
- the work done by the call to Pair_Sum
  - the data being worked on at this point are the elements of the lists of sums
  - count each operation that accesses an element of either list

Use this structure for your experiments:

```
for n = 4 to 15:              # set sizes
      for i = 1 to 20:        # number of tests
            generate a set S of n random integers
            generate a set of at least 10 target values
            for each target value k:
                  apply BFI_Subset_Sum(S,k) – count the operations
                  apply HS_Subset_Sum(S,k) – count the operations
            compute the average number of operations for BFI_Subset_Sum for this set
            compute the average number of operations for HS_Subset_Sum for this set

      compute the average number of operations for BFI_Subset_Sum for this n
      compute the average number of operations for HS_Subset_Sum for this n

Tabulate the computed average numbers of operations.
```

Part 3:

Answer the following question:

Do your observations support the theoretical predictions that BFS_Subset_Sum is in O(2^n) and HS_Subset_Sum is in O($n * 2^{\frac{n}{2}}$)

Part 4: (Optional)

Experiment with further improvements to HS_Subset_Sum, such as dividing S into more than two parts, or modifying the Pair_Sum algorithm to eliminate possible pairings more efficiently.

What to Hand In:

For Part 1:

Your code, documented and with citations for the source of any part that you did not write.

Your test evidence that shows your code executes correctly

For Part 2:

Your modified code that counts operations.

Your tabulated (or plotted on a chart) observed average number of operations for the different values of n

For Part 3:

Your conclusion, and an explanation of how you arrived at it.