Homework 8

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

21 September 2016

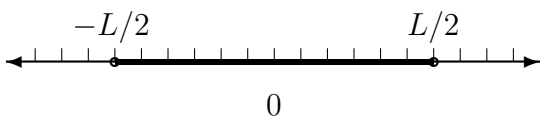
Problem 13:

The algorithm in order to solve this problem first requires pruning rules in order to decrease the size of the binary tree that is generated from the subsets of $v_1,\ldots,v_n$.

The two pruning rules are:

1) If the absolute value of a node v is greater than (L/2), then we can prune that subtree rooted at the node v.
2) If two nodes have the same value and are at the same depth in the binary tree, then we can arbitrarily select one of the nodes and prune the subtree rooted at that node.

Below is a line representation of the valid area that a node can occupy in terms of the value it holds.



Now we will derive an iterative, array-based algorithm.
Let A[k, S] hold the bitstring of the subset k with sum S, where a 1 in the bitstring indicates $x_i = 1$ and a 0 indicates $x_i = 0$. We compute A[k, S]:

Function: *Sum*

Input: *int* $v_1, \ldots, v_n$

for $k \leftarrow 1$ *to* n **do**

  for $S \leftarrow 0$ *to* L **do**

    $A[k][S] = A[k][S] :: 1$

    $A[k][S] = A[k][S - v_k] :: 0$

return $A[k][0]$

At A[k][0], the bitstring is stored for the subset k with a sum of 0, if a solution exists.

Problem 14:

The goal of this problem to to find a solution to the polynomial

$$\left(\sum_{i=1}^{n} x_i v_i\right) \bmod n = \ L \bmod n$$

given a series of positive values for v and L. Also, x can be 0 or 1.

This problem is similar to the subset sum problem, except that values larger than L can not be immediately pruned. Instead, at a given level, if two nodes have the same value *modulo* n, one can be arbitrarily pruned.

Additionally, since the goal is to find a solution to the polynomial, a bit string can be created by appending a 0 or a 1 to consider the possibilities in which a value was included or discluded in the sum.

Enumerating all possibilities and then considering the above pruning rule gives way to the following algorithm.

Function: *Modular Subset Sum*
Input: *Positive Integers* $v_1, \ldots, v_n, L$
for $k = 0$ *to* n **do**
  for $s = 0$ *to* $L \bmod n$ **do**
    if *A[k,s] is defined* **then**
      `/* :: is the concatentation operator.` $a :: c \rightarrow ac$ `*/`
      $A[k+1, s] = A[k, s] :: 0$
      $A[k+1, (s + v_{k+1} \bmod n) \bmod n] = A[k, s] :: 1$
Return: $A[n, L \bmod n]$

When this algorithm completes, the answer, if it exists, will be the bit string at $A[n, L \mod n]$. This bit string starting, read from left to right, will represent the coefficient of each v in the polynomial.

Problem 15:

This problem is very similar to the Knapsack problem from class, except that it can be conceptualized slightly differently: instead of having a single list of items where you can only take an item once with no repeats, now you have a class of identical items, which you can select from repeatedly. This additional complexity adds a single inner loop to our otherwise fairly standard computation. The new loop represents choosing an item one, twice, thrice, *etc.* Below, *max* takes the pairwise maximum of scalar values, while *maxElement* returns the highest scalar value in the array argument.

```
Function: Repeated Knapsack
Input: Positive Integers v_1, ..., v_n, w_1, ... w_n, W
Globals: A zeroed array A[n][W]
for k = 1 to n do
    /* generate level k+1 from level k                    */
    for s = 1 to W do
        /* generate solutions for each of the weights     */
        if A[k, s] is defined] then
            for p = 1 to W do
                A[k + 1, s + pw_k] =
                max(A[k + 1, s + pw_k], maxElement(A[k, s]) + pv_k)
Return: maxElement(A[n, W]
```

Problem 16:

For this problem, we maintain a tuple representing a subset's current values and it's potential to catch to the other sequence. Once a term is not able to catch up, we prune that node from the tree. Lastly, we represent the subset as a bitstring, where each bit represents the inclusion or exclusion of the corresponding set element. Note that the number of inner loop iterations sums to L since our "catch up" factor, which is monotone decreasing, can cross no more than an L-length distance (just like with the real line shown in #13), which is obvious when the inner list is represented as a linked list instead of as an array.

Function: *Repeated Knapsack*
Input: *Positive Integers* $v_1, \ldots, v_n$
Globals: *A array A[n] holding linked lists*
Let $S = \sum_{i=1}^{n} v_i^3$ and $P = \prod_{i=1}^{n} v_i$
$A[1] \leftarrow (0, 1, S, P, emptyString)$
for $i = 1$ *to* n **do**

```
/* generate level i+1 from level i                                    */
```

foreach $(sum, prod, Rsum, Rprod, str) \in A[i]$ **do**
  if $sum = prod$ **then**
    Return: *str*
  if $sum + RSum < prod$ **then**
    Continue
  if $prod * Rprod < sum$ **then**
    Continue
  $A[i+1]$.append($(sum, prod, Rsum, Rprod, str :: 0)$)
  $A[i+1]$.append(
  $(sum + v_i^3, prod * v_i, Rsum - v_i^3, Rprod/v_i, str :: 1)$)