Homework 12

Brian Knotten, Brett Schreiber, Brian Falkenstein

September 24, 2018

13

Define S to be the running sum of values, and L to be the total sum of values in the input.
The decision tree can be constructed as follows: for $0 \leq i \leq n$, at depth i, have the left branch add $-v_i$ to S (that is, setting x_i to 1), and the right branch add $+v_i$ (setting x_i to 0). For both branches, subtract (positive) v_i from L. A solution can then be found at any leaf containing an S value of 0. This will check all possible 2^n possibilities. The tree can then be pruned to run in linear time by following the rules:

1. If two nodes at the same depth have the same value, prune one of them.
2. If $|S + (-1)^{x_i} v_i| > L$ for a depth i

Note that if $|S + (-1)^{x_i} v_i| = L$, that sub-tree will contain a solution (if S is negative, add the remaining values, and if S is positive, subtract the remaining values).
The first rule works because two nodes at the same depth still have the same remaining values to decide over, so one can arbitrarily be pruned. The second rule works, because if a node has an $|S|$ value larger than the sum of the remaining values, there is no way any combination of additions or subtractions could bring $|S|$ to zero.
This pruned decision tree can be visualized in a table that is of size $n \times L$, and can thus be traversed in time nL. Because L is a constant, we can say that this algorithm runs in linear time.

14

Define S to be the running sum of values, and V to be the set containing the input values.
The decision tree can be constructed as follows: for $0 \leq i \leq n$, at depth i have the left branch subtract v_i from S and the right branch add v_i to S. A solution can be found at any branch where $S\%n = L$. This tree, as in the previous example, will list all possible 2^n possibilities. It can be pruned by the following rules:

1. If two nodes at the same depth have the same value, prune one of them.
2.

The first rule works for the same reason as it does in question 13: all nodes at the same depth have the same remaining values to decide over, so if two have the same S value, one can arbitrarily be pruned.

15

The algorithm works by considering every possible capacity from 1 to W and, for each capacity, considering the best possible value if each item was the last one selected. The algorithm utilizes two one-dimensional arrays: $M[W]$ (stores the max value for each capacity) and CVL (CurrentValueLast - stores the values of each item if it was taken last).

Let M[0] = 0. For each capacity x from 1 to W, consider each item i from 1 to n. If the weight of the currently considered item is less than the current capacity, let $CVL[i] = M[x - w_i]$. If the weight of the item is greater than the current capacity, then let $CVL[i] = 0$. After considering each item, let $M[x] = max(CVL)$. At the end of the outer loop, return $M[W]$ = the max value at capacity W.

The algorithm loops from 1 to W, and within each iteration loops over all the items (1 to n) and loops over the array CLV, which has length n. The algorithm takes time $W \times 2n = O(Wn)$

16

Create a binary tree of height n where each node keeps track of the set's sum of cubed elements and the product of all the elements. For a given node at level i, let its left child be the set including v_{i+1} and the right child be the set excluding v_{i+1}. Any leaf whose sum of the elements cubed equals the product of the elements contains the solution subset S. Since there are 2^n leaf nodes, we must remove some of them perform the following pruning strategies to reduce the problem to a polynomial size.

1. At any depth, when $\Sigma_{v \in S} v_i^3 = \Pi_{v \in S} v$, immediately prune any other nodes with this same result and all child nodes. There is no need to continue searching the tree since a solution has already been found. So if a solution exists, it can be found at any level of the tree, and so that level becomes the only leaf node and the searching stops. In the worst case, it is found at level n, if at all.

If there is only a solution at the ith level, then it must be the case that v_i was included. So the solution is the left child of a node from level $i-1$. If v_i was not included, then there would be a solution at level $i-1$.

At $i-1$, it must be the case that $S + v_i^3 = Pv_i$, where S is the accumulated sum and P is the accumulated product.

The following is a recursive algorithm, where -1 indicates no recursive solution found:

```
int A(int running_sum, int running_product, int i) {
    if(running_sum == running_product) return running_sum;

    if(i == N) return -1;

    return
        A(running_sum + (v[i]*v[i]*v[i]), running_product * v[i], i + 1) ||
        A(running_sum, running_product, i + 1);
}

Output A(0, 1, 0);
```