CS/ECE 473 Fall 2020
Homework 3 Problem 2

Ryan Prendergast (ryanp4)
Noah Watson (nwatson3)
Lawson Probasco (lawsonp2)

Solution: In order to reduce the space complexity of the subset sum problem, we will use the technique of splitting subproblems into size $n/2$ rather than the ordinary reduction in dynamic programming, which is just to reduce problem size by 1.
To be able to split the input array subset sum problem in half, we must know what the sum of the left half and right half should be so we can adjust the target sum for the recurive calls. To do this, we define a method lefthand_sum which computes this value.
It should be noted that the reason for defining this algorithm is because the subproblems where the input array A is split in half can compute the output in the following way: $A_{sum=j} \circ A_{sum=T-j}$ where $A_{sum=j}$ is computed over the left half of the array $A[1, ..., n/2]$, $A_{sum=T-j}$ is computed over the right half of the array $A[n/2, ..., 1]$, and j is the output from lefthand_sum.

```
def lefthand_sum(A[1,...,n], T):
  # where subset_sum is the generic subset_sum algorithm
  # that runs in O(nT) time and uses linear space
  # and returns the entire solution column
  lefthand_sets = subset_sum(A[1,...,n/2], T)
  righthand_sets = subset_sum(A[n/2,...,n], T)

  lh_sum = -inf
  for 1 <= i <= n:
    if (lefthand_sets[i] and righthand_sets[T - i]):
      lh_sum = i
      break

  if (lh_sum == -inf):
    # this would indicate the target sum was not possible
    return None

  return lh_sum
```

This algorithm takes the result arrays from running subset sum (optimized for space without returning result) and searches all possible subset sums of A over all possible pairs of subset sums over the left and right half of A. It then returns the sum that was in the lefthand side for a subset that sums to T (if it exists, None otherwise).

Time Complexity The time complexity of this algorithm is dominated by the calls to subset sum, which run in $O(nT)$ time.
Space Complexity This algorithm assumes calls to subset sum are the optimized version that only uses linear space, thus the resulting space complexity of this algorithm is $O(n + T)$.

Algorithm This lefthand_sum can be used to solve the subset sum problem (with output) in the following algorithm.

```
def opt_subset_sum(A[1,...,n], T):
  if (T = 0):
    return subset_sum_res(A[1,...,n],T)

  j = lefthand_sum(A[1,...,n], T)
  if (j is None):
    return None

  lefthand_result = opt_subset_sum(A[1,...,n/2], j)
  righthand_result = opt_subset_sum(A[n/2,...,n], T - j)

  # + operator here is just combining/appending two lists
  return lefthand_result + righthand_result
```

Time Complexity The time complexity of this algorithm is defined by the following recurrence (using lowercase t as target): $T(n,t) \leq O(nt) + T(n/2, j) + T(n - n/2, t - j)$. Using guess for T s.t. $T(n,t) \leq \alpha * nt$, this can be approximated to (when ignoring floor computation of $n/2$) $T(n,t) \lesssim \beta * nt + \alpha * \frac{n}{2} * (t - j) = (\beta + \frac{\alpha}{2} nt)$, which is valid so long as $\alpha \geq 2\beta$, meaning that $T(n,t) \leq O(nt)$, which is the target runtime.

Space Complexity The space complexity of this algorithm is defined by the following recurrence (using lowercase t as target for consistency): $S(n,t) \leq max O(n+T), S(n/2,t), S(n-n/2,t)$, which is simply $S(n,t) \leq O(n+t)$, the target space complexity.

■