3 Problem 4.3

1. *uniqueSums*$(X = \{x : x \in \mathbb{Z}\}, Y = \{y : y \in \mathbb{Z}\})$: given two sets sets of integers X and Y, each with n items, returns the unique sums between X and Y

Algorithm 3 *uniqueSums*

```
1:  procedure UNIQUESUMS(X, Y)
2:      let map be an array of size max(X) + max(Y) initialized to 0
3:      let sumlist be a empty list with O(1) insert and O(1) remove
4:      for x ∈ X do
5:          for y ∈ Y do
6:              map[x + y] ← map[x + y] + 1
7:              sumlist.insert(x + y)
8:          end for
9:      end for
10:     for all elements sum ∈ sumlist do
11:         if map[sum] > 1 then
12:             sumlist.remove(sum)
13:         end if
14:     end for
15:     return sumlist
16: end procedure
```

Time Complexity:

The function has a nested for loop that iterates through every element in Y per every element in X. The size of both of these sets is n. All actions within the nested loops take constant time. Thus, lines 4-12 take $O(n^2)$ time. The for loop on lines 14-18 runs for all elements in *sumlist*, and we have just put n^2 elements in *sumlist* (from lines 4-12). The loop does constant time actions every iteration, meaning lines 14-18 are also $\in O(n^2)$

$T(n) = 2n^2 + c \in O(n^2) \in O(n^2 \log n)$

Therefore, our algorithm runs in $O(n^2 \log n)$.

Does it actually work doe?:

Every possible pair of elements from two sets can be found by matching every element of set X to every element of set Y such that $\forall x \in X, y \in Y((x, y)$ exists). Every one of these pairs is represented by our nested for loops since for all x's their sum with every y (every (x, y) pair) is computed exactly once. While being computed, the frequency of every sum is also stored in *map*. By definition, a sum can only be unique when

the choice of $(x, y) \in X \times Y$ is unique, meaning a sum z is unique when $\forall x_1, x_2 \in X, \forall y_1, y_2 \in Y((x_1 + y_1 = z = x_2 + y_2) \rightarrow ((x_1 = x_2) \wedge (y_1 = y_2)))$. Since every pair is only calculated once, if an item z in the map has a frequency greater than 1, it cannot be unique (because that would indicate for z that $\exists x_1, x_2 \in X, \exists y_1, y_2 \in Y((x_1 + y_1 = z = x_2 + y_2) \rightarrow ((x_1 \neq x_2) \wedge (y_1 \neq y_2))))$. With this in mind, our algorithm starts with every possible sum (lines 4-13), and removes any items with frequency greater than 1 in the *map* since those sums cannot be unique (lines 14-18). Therefore, our algorithm correctly return all unique sums of X and Y

2. $uniqueSum(X = \{x : x \in [0, ...M]\}, Y = \{y : y \in [0, ...M]\}, M \in \mathbb{N})$: given two sets sets of integers X and Y that contain values between 0 and M, returns the unique sums between X and Y

Algorithm 4 *uniqueSums*

```
1: procedure UNIQUESUMS(X, Y, M)
2:     let k ← ⌈log_2 (2M + 1)⌉
3:     M ← 2^k
4:     let X' be an array of size M filled with zeroes
5:     let Y' be an array of size M filled with zeroes
6:     for i from 0 → M do
7:         if i exists in X then
8:             X'[i] ← 1
9:         end if
10:        if i exists in Y then
11:            Y'[i] ← 1
12:        end if
13:    end for
14:    let R ← the product of the pairwise multiplication between F_M X' and F_M Y'
15:    let R ← (F*_M R)/M
16:    let Z be a set of integers
17:    for i from 0 → M do
18:        if R[i] = 1 then
19:            Z.insert(i)
20:        end if
21:    end for
22:    return Z
23: end procedure
```

Time Complexity:

The function has a for loop from 0 to M on lines 6-13 and another for from 0 to M on lines 17-21. The Fourier transforms of $F_M X'$, $F_M Y'$, and $F^*_M R$ will all be $\in O(M \log M)$ when M is a power of 2 (which it is forced to be by lines 2-3). The pairwise multiplication on line 14 will take M time

$$T(n) = O(M) + O(M \log M) \in O(M \log M)$$

But wait... doesn't M change sizes in the function? This is true, but note that the M on line 3 can never be greater than 8 times the input M. This can easily be seen through a proof by contradiction Consider $\lceil \log_2 (2M + 1) \rceil \leq \lceil \log_2 (2M + 2M) \rceil = \lceil \log_2 (4M) \rceil = 2 + \lceil log_2 M \rceil$. If the M on line 3 is greater than 8 times the input M, then $\lceil log_2 M \rceil$ must have rounded up $\log_2 M$ by more than 1, which is impossible.

Does it actually work doe?:

Before proving our algorithm, let it be stated that the product of two polynomials $r(x) = p(x)q(x)$ involves the multiplication of every monomial in p with every monomial in q. Therefore, the process of multiplying two polynomials through the use of Fourier transforms as covered in section **4.3.2** necessarily computes the multiplication of every monomial (term) in p to every monomial (term) in q. Note also that the multiplication of two exponents $x^a \cdot x^b = x^{a+b}$, where the exponents are added together.

With this in mind, what our algorithm does is translate the sets of X and Y into polynomial functions where if $m \in X$, then $a_m x^m$ with $a_m > 0$ will be in the polynomial function representation of X (if an element m exists in X, then there will exist a term with degree m in the polynomial representation of X). Since it is guaranteed[3] that there are no duplicates in X, all coefficients a_i will either be 1 or 0, where the coefficient $a_i = 1$ iff $i \in X$. The process of doing this is what occurs from lines 4-13. In our implementation, X' and Y' become the coefficient representations of the polynomial transformation of X and Y. With these representations, our algorithm computes the product of the polynomials X' and Y' by taking $(F_M^*(F_M X' F_M Y'))/M$ (lines 14-15), where the results of the discrete transforms $F_M X'$ and $F_M Y'$ are multiplied pairwise before taking the conjugate Fourier transform of that pairwise result. This process was proved in section **4.3.2** to correctly calculate in product between two polynomials. From the earlier paragraph, we know that this product of two polynomials necessarily contains the product of every monomial in X' with every monomial in Y' (if $x^a \in X'$ and $x^b \in Y'$, then there necessarily exists $x^a \cdot x^b = x^{a+b} \in R$, represented, of course, still by coefficients), and since the degree of these monomials comes from the original sets of X and Y, the product Z then contains every possible sum of the form $a + b = c$ (encoded as $x^a \cdot x^b = x^c$), where $a \in X$, $b \in Y$, $x^a \in X'$, $x^b \in Y'$, $x^c \in R$ and $c \in Z$.

We now have all possible sums, but we need all unique sums. Recall from **4.3.1** that the coefficient a_k that corresponds to x^k in the product $R = X'Y'$ is given by the convolution

$$a_k = \sum_{i=0}^{k} x'_i y'_{k-i}$$

[3]If distinctness within a set was not guaranteed (this is technically impossible by the definition of set, but this idea is why we don't clear non-distinct terms per recursive call in question 4.3.3), we could just count the number of times any element was in the set and make that the coefficient for that term of the polynomial. This would ensure any non-unique sums made by the non-distinct elements (which may also be present distinctly in the other set) get properly cleared away in lines 17-20, but we don't need to worry about that here

where x'_i and y'_{k-i} represent the coefficients of x^i and x^{k-i} in X' and Y' respectively. Recall also that x'_i and y'_{k-i} must be either 0 or 1. Let us consider 3 cases: when $a_k < 1$, $a_k > 1$, and $a_k = 1$.

When $a_k < 1$, a_k can only be 0, this would indicate that

$$a_k = \sum_{i=0}^{k} x'_i y'_{k-i} = x'_0 y'_k + \dots x'_k y'_0 = 0$$

Since all x_i and y_{k-i} are coefficients representing the degrees of terms in X' and Y', this means that no two terms from these sets can create x^k, $\forall x^a \in X', \forall x^b \in Y'(\neg(x^a \cdot x^b = x^k))$, which necessarily means $\forall a \in X, \forall b \in Y(\neg(a+b=k))$, meaning no sum of an element in X and an element in Y can produce k. Therefore, our algorithm does not include the indices of any coefficients of 0 in the final set to return of unique sums

When $a_k > 1$,

$$a_k = \sum_{i=0}^{k} x'_i y'_{k-i} = x'_0 y'_k + \dots x'_k y'_0 > 1$$

Since all x_i and y_{k-i} are either 0 or 1, this can only happen when more than one of $x'_0 y'_k, \dots x'_k y'_0$ produces 1, meaning that there exists at least two sets of $x^a \in X'$ and $x^b \in Y'$ such that $x^a \cdot x^b = x^k$, which necessarily means that there exists at least two (a, b) pairs $a \in X, b \in Y$ such that $a + b = k$, which means that k is not a unique element (there is more than one way an element from X and an element from Y can sum to k). Therefore, our algorithm does not include the indices of any coefficients of value greater than one in the final set to return of unique sums

When $a_k = 1$,

$$a_k = \sum_{i=0}^{k} x'_i y'_{k-i} = x'_0 y'_k + \dots x'_k y'_0 = 1$$

This can only happen when exactly one of $x'_0 y'_k, \dots x'_k y'_0$ produces 1, meaning that there exists exactly one of $x^a \in X'$ and $x^b \in Y'$ such that $x^a \cdot x^b = x^k$, which necessarily means that there exists exactly one (a, b) pair $a \in X, b \in Y$ such that $a + b = k$, which means that k is a unique element (there is only one way an element from X and an element from Y can sum to k). Therefore, our algorithm includes the indices of any coefficients of value 1 in the final set to return of unique sums

After finding all possible sums between X and Y, out algorithm correctly only returns the unique sums. Thus, our algorithm is correct.

3. *multiUniqueSum*$(A[1 \dots k][1 \dots M_i], M)$, given an array of A of $k = 2^l$ sets of internally distinct integers between 1 to M, and given M, finds the number of unique sums; that is, the number of s such that $s = \sum_{i=1}^{k} A[i][j_i]$ for exactly one set of j_i.

   multiPolyProd$(A[1 \dots k](a_{k0} \dots a_{kn-1}))$, given an array A of $k = 2^l$ vectors representing polynomials of degree at most $n-1$ (that is, $P_k(x) = \sum_{i=0}^{n-1} A[k][i]x^i$), returns a vector of size $2^{\lceil \log kn \rceil}$ representing the coefficients of the product of the polynomials.

Algorithm 5 *multiUniqueSum*

```
1:  procedure MULTIUNIQUESUM(A[1...k][1...M_i], M)
2:      let B ← []
3:      for i from 1 to k do
4:          append to B, (a_{i0} ... a_{iM-1}) : a_{ij} = 1 if j + 1 ∈ A[i], else a_{ij} = 0
5:      end for
6:      let P ← multiPolyProd(B)
7:      return count of 1 in P
8:  end procedure
9:  procedure MULTIPOLYPROD(A[1...k](a_{k0} ... a_{kn-1}))
10:     if k = 1 then
11:         return A[1]
12:     end if
13:     let A_L ← A[1...k/2]
14:     let A_R ← A[k/2 + 1...k]
15:     let P_L ← multiPolyProd(A_L)
16:     let P_R ← multiPolyProd(A_R)
17:     let m ← degree of P_L + degree of P_R + 1
18:     round m up to power of 2
19:     let T ← pointwise product F_m(P_L) × F_m(P_R)
20:     return F*_m(T)/m
21: end procedure
```

We see that the base case $k = 1$ for *multiPolyProd* is correct since if there is only one polynomial to multiply, then it is already the product. It is impossible that $k = 0$ since we are given k is a perfect power of 2.

Now suppose it is correct for all A meeting the assumptions, of size less than k, then consider an A of length k. A_L and A_R are then also of size a power of 2 by the given constraint on k, so by our inductive hypothesis, P_L and P_R are the correctly computed products of $A[1 \dots k/2]$ and $A[k/2 + 1, k]$, respectively. Then by associativity, the desired $\prod_{i=1}^{k} A[i] = (\prod_{i=1}^{k/2} A[i])(\prod_{i=k/2+1}^{k} A[i]) = P_L P_R$. Thus, all that remains is to compute this product of two polynomials.

By the same reasoning as is present in the textbook, section 4.3.1, we know that $P_L P_R(\omega_m^i) = P_L(\omega_m^i) P_R(\omega_m^i)$ due to the definition of function multiplication. This is, by definition of fourier transform, $F_m(P_L P_R) = F_m(P_L) \times F_m(P_R)$ pointwise, which is correctly computed on line 19. Then, by theorem 4.1, $P_L P_R = F_m^*(F_m(P_L P_R))/m$, which is correctly returned, so we the procedure is correct for a set of k polynomials, and thus all $k = 2^l, l \in \mathbb{Z}^+$ by induction. We also note that the condition on m for the fast fourier transform to work is satisfied, that is, m is a power of 2, and $\deg P_L P_R \leq \deg P_L + \deg P_R \leq m - 1$.

We now prove the correctness of *multiUniqueSum*, which hinges on the claim that the number of ways that a given sum s can be generated from A is precisely the coefficient of $s - k$ in the polynomial product of one-hot polynomial embeddings as described in line 4. That is, for an input set of integers, the coefficient in the corresponding polynomial for the x^a term is 1 if $a - 1$ is in the set, and 0 otherwise.

(I apologize for my descent into notation hell) Consider the cx^a term in the final product P. For j_i, being the index of the factor from the ith polynomial in a term of the expansion, and c_{ij}, the coefficient of the jth index term in the ith polynomial, we have

$$cx^a = \sum_{j_1 \dots j_k \,:\, j_1 + \dots j_k = a} \prod_{i=1}^{k} c_{ij_i} x^{j_i} = \sum_{j_1 \dots j_k \,:\, j_1 + \dots j_k = a} x^a \prod_{i=1}^{k} c_{ij_i}$$

Since $c_{ij} = 1$ or 0, the product $\prod_{i=1}^{k} c_{ij_i}$ is nonzero (equal to 1) only when $c_{ij_i} = 1$ for all j_i. Thus this can be rewritten, then, as

$$c = \sum_{\substack{j_1 \dots j_k \,:\, j_1 + \dots j_k = a, \\ c_{1j_1} \dots c_{kj_k} = 1}} 1$$

Using our embedding from integer set to polynomial, we notice that this precisely counts the number of sums $n_1 \dots n_k$ from our integer sets (with n_i coming from the ith integer set) such that $\sum_{i=1}^{k} (n_i - 1) = a$. This is, $s - k = a$ for s, the sum of the n_i. Then, by our definition of unique sum, s is a unique sum if and only if the coefficient on x^{s-k} in our polynomial is 1. Therefore, the number of unique sums is the count of 1 in the vector returned by *multiPolyProd*(B), as done in lines 6-7. Thus, our algorithm is correct.

Let $T(k, n)$ denote the runtime for *multiPolyProd* in terms of the number of polynomials k, and the size of each polynomial, $n = \deg P - 1$. The base case $T(1, n)$ is constant time. Lines 13-14 are linear by array copying. The recursive calls can be represented by $2T(k/2, n)$, and determining the degrees can be done in at most $\mathcal{O}(n)$. Finally, the degree

of P_L and P_R are at most $\frac{k}{2}(n-1)$ (since they are the product of $k/2$ polynomials of degree at most $n-1$), so $m \approx kn$, within an order of magnitude due to rounding up to a power of 2. The fourier transforms are then $\mathcal{O}(kn\log(kn))$, and the pointwise multiplication is linear in kn. Thus, $T(k,n) = 2T(k/2,n) + \mathcal{O}(kn\log(kn))$.

Consider the recursion tree; for the ith level, we have 2^i nodes, each of which contributes $\frac{k}{2^i}n\log(\frac{k}{2^i}n)$ work. Since there are $\log k$ levels, the total work is the summation $\sum_{i=0}^{\log k} kn\log(\frac{k}{2^i}n) = \sum_{i=0}^{\log k} kn(\log(kn)-i) = kn(\log(kn)\log(k) + \log kn - \frac{\log^2(k)+\log k}{2})$. The $kn\log(kn)\log(k)$ term clearly dominates, so $T(k,n) \in \mathcal{O}(kn\log(kn)\log(k))$.

$multiUniqueSum$ does kM work initializing B (since each polynomial has M entries to initialize), so the loop from lines 3-5 is $\mathcal{O}(kM)$, and the degree of P is also at most kM, so this is also linear with the same complexity. Therefore, $multiUniqueSum$ is also $\mathcal{O}(kM\log(kM)\log(k))$.