**DP** for solving the **3WAYPARTITION** problem **P:**

**1. Specify a problem P:**

Given an array **A[1…n]** ofpositive integers (duplicates allowed), the **3WAYPARTITION(A,b,c,d)** method returns **YES** if **A** can be partitioned into **3** disjoint subsets **B, C, D** such that **B** ∪ **C** ∪ **D** = **A** and the total value of **B, C,** and **D** are equal, i.e., the total sum of elements of **B** **(b)** = total sum of elements of **C** **(c)**= total sum of elements of **D (d)**. In other words **b=c=d=(**total sum of elements of **A)/3.**

**3WAYPARTITION(A,b,c,d)** will be called as **3WAYPARTITION(A, 0, 0, 0).**

Along with that, there is a **4-D** the **Memoization** table of dimension **(n+1) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ).**

**total\_value(A)** is the sum of all elements of **A[1…n].**

**2. Give a recurrence expression/formula or recursive algorithm for solving P :**

**3WAYPARTITION(A,b,c,d)** method returns **YES** if **A** can be partitioned into **3** disjoint subsets **B, C, D** such that **B** ∪ **C** ∪ **D** = **A** and the total value of **b,c** and **d** are equal.

**3WAYPARTITION(A[1..n], b, c, d) =** **1** **;** when **|A|** = **0** & **b=c=d**

**= 0 ;** when **|A|** = **0** & **b!=c!=d**

**= max { 3WAYPARTITION(A[2..n], b + A[1], c, d) ,**

**3WAYPARTITION(A[2..n], b, c + A[1], d),**

**3WAYPARTITION(A[2..n], b, c, d + A[1])**

**};**  otherwise **(**i.e. **|A| !**= **0 )**

**3. Prove the correctness of the recurrence relation.**

The **3WAYPARTITION(A,b,c,d)** function will return **1**, when **A** exhausts i.e. when **|A|=0 & b=c=d**. This reflects the case when **A** is traversed entirely & we have found the sum of the required subsets **B,C & D.** Thus **b=c=d**=**total\_value(A) / 3**

The **3WAYPARTITION(A,b,c,d)** function will return **0**, when **A** exhausts i.e. when **|A|=0** but **b!=c!=d**. This means that we could not divide **A** into **3** disjoint subsets with the given constraint.

The **3WAYPARTITION(A,b,c,d)** function also recursively calls itself thrice when **|A| !**= **0** , depending on which subset the first element of **A** is being added to. Accordingly, these 3 cases follows:

* When the first element of **A** i.e. **A[1]** is considered for subset **B** only, i.e., the sum would be **b + A[1].** Thus, we trim the array **A** off its first element & the function is called as **3WAYPARTITION(A[2..n], b + A[1], c, d).**
* When the first element of **A** i.e. **A[1]** is considered for subset **C** only, i.e., the sum would be **c + A[1].** Thus, we trim the array **A** off its first element & the function is called as **3WAYPARTITION(A[2..n], b , c + A[1] , d).**
* When the first element of **A** i.e. **A[1]** is considered for subset **D** only, i.e., the sum would be **d + A[1].** Thus, we trim the array **A** off its first element & the function is called as **3WAYPARTITION(A[2..n], b , c , d + A[1] ).**

// SUBPROBLEMS

Building on the above intuition, an exhaustive search will be done on whether to include the first element of **A** to subset **B, C or D**. At each step, the problem is broken into 3 subproblems with input reduced by a size of 1.

Eventually this recurrence will converge to find whether such subsets can be constructed or not.

**4. Describe a memoization data structure:**

The **Memoization** table **M** is a **4-D** arrayof dimension **(n+1) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 );** as in any worst case **b, c or d** can extend to the sum of all the elements in the array **A[1…n].**

Any index of **M**, says **(i, j, k, l)** tells when **i** elements are left in **A**, **j** is the sum of the elements in subset **B**, **k** is the sum of the elements in subset **C & l** is the sum of the elements in subset **D.**

**Initialization of array:**

**M** [0] [j] [k] [l] = 1 , when j=k=l // implies **|A|=0** & **b=c=d**

**M** [0] [j] [k] [l] = 0 , when j!=k!=l // implies **|A|=0** & **b!=c!=d**

**5. Give an algorithm/ordering for solving P for all values.**

for **i** in **1…n:**

for **j** in **1…total\_value(A):**

for **k** in **1…total\_value(A):**

for **l** in **1…total\_value(A):**

**if** **[**(j+A[i]) > **total\_value(A) OR (**k+A[i]) > **total\_value(A) OR (**l+A[i]) > **total\_value(A)]:**

**M** [i] [j] [k] [l] = 0 // array out of bound

**else:**

**M** [i] [j] [k] [l] = **max {** **M** [i-1] [j+A[i]] [k] [l] **,**

**M** [i-1] [j] [k+A[i]] [l] **,**

**M** [i-1] [j] [k] [l+A[i]] **}**

**6. How to solve the original problem from memo**

If **M** [n] [total\_value(A) / 3] [total\_value(A) / 3] [total\_value(A) / 3] contains **1,** it implies that **0** elements are left in **A (**as the entire array is traversed from 1 to n) & **j=k=l=**total\_value(A)/3 **i.e.** the sum of the elements in subsets **B**, **C & D** are each equal to one third of the sum of the elements in **A**.

Thus we have found 3 disjoint subsets to satisfy our constraints. Thus, problem P is solved.

If **M** [n] [total\_value(A) / 3] [total\_value(A) / 3] [total\_value(A) / 3] contains **0,** it implies that **0** elements are left in **A, &** the sum of the elements in subsets **B**, **C & D** are not equal to each other, hence the problem constraint is not satisfied.

**7. What is space and time complexity for solving problems?**

The above problem uses a **4-D** array for **Memoization** of dimension **(n+1) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ).**

**total\_value(A)** can be **O(n)** in the worst case.

Space complexity - **O(** **n4** **)**

With further observation, it can be noticed that **M** [i] [j] [k] [l] depends on **M** [i-1] [j] [k] [l] only. Hence, storing two **3-D arrays** of dimension **O(n3)** will still solve the question.

Hence, Space complexity can be optimized to **O(n3).**

**However, the subsets cannot be retrieved in this optimized space.**

As filling the Memoization table requires traversing over all **(n+1) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 ) \* ( total\_value(A)+1 )** elements of **M**, and **total\_value(A)** can be **O(n),** in the worst case.

So, the time complexity of filling the table would also be **O(n4)**, assuming constant work in each iteration.