Homework 5

1. Design and analyze asymptotically an $O(n+m+p)$ **transform-conquer** algorithm for the following problem:
    - input: three **sorted** arrays *A[1..n], B[1..m], C[1..p]*;
    - output: a sorted array *D[1..n+m+p]* containing elements of *A*, *B*, and *C*.

A1: The parent array will include elements, not repeating, of all 3 arrays, extending such that the last element is n + m + p, and the one before that is n + m + p - 1 (Wrong implementation)

Ex: [1, 2, 3..10] [1, 2, 3, 4..6] [1, 2..15] → [1, 2, 3, 4, 5..29, 30, 31]

```
Int* transformSort (int myArray1[], int myArray2[], int myArray3[])
{
        //Make a parent array of size n+m+p
        Int parentArray[myArray1.size() + myArray2.size() + myArray3.size()];
        //Create a for loop that runs for n+m+p
        For (int i = 0; i <= n + m + p - 1; ++i)
        {
                //Paste in the values of each element cubby, corresponds to i + 1
                //Element 0 should have value 1, element 30 should have value 32
                parentArray[i] = i + 1;
        }
        Return parentArray;
}
```

Analysis: I will be analyzing the number of addition operations.

$M(n + m + p) = 2 + \sum_{i=0}^{n+m+p-1} (4) + 1 \rightarrow 2 + (4)((n + m + p - 1) - (0) + 1) + 1 \rightarrow 2 + 4(n + m + p) + 1 \rightarrow 3 + 4(n + m + p) \rightarrow M(n + m + p) \in \Theta(n + m + p)$

2nd Attempt At #1:

The parent array will include elements, with repeating, of all 3 arrays, should have the same amount of elements as the three combined (n + m + p) elements

```
Int* transformSort (int myArray1[], int myArray2[], int myArray3[])
{
        //Have 3 separate counter variables for each array, starting index 0
        Int index1 = 0;
        Int index2 = 0;
        Int index3 = 0;
        //Make a parent array of size n+m+p
        Int parentArray[myArray1.size() + myArray2.size() + myArray3.size()];
        //Create a for loop that runs for n+m+p
        For (int i = 0; i <= myArray1.size() + myArray2.size() + myArray3.size() - 1; ++i)
```

```
	{
		//Paste in the values of each element cubby, the least first
		//Only myArray1 is left
		If ((index2 >= myArray2.size() && index3 >= myArray3.size()) && index1 <
myArray1.size())
		{
			parentArray[i] = myArray1[index1];
		}
		//Only myArray2 is left
		Else If ((index1 >= myArray1.size() && index3 >= myArray3.size()) &&
index2 < myArray2.size())
		{
			parentArray[i] = myArray2[index2];
		}
		//Only myArray3 is left
		Else If ((index1 >= myArray1.size() && index2 >= myArray2.size()) &&
index3 < myArray3.size())
		{
			parentArray[i] = myArray3[index3];
		}

		//myArray1 and myArray2 is left
		Else If ((index1 < myArray1.size() && index2 < myArray2.size()) && index3
>= myArray3.size())
		{
			If (myArray1[index1] <= myArray2[index2])
			{
				parentArray[i] = myArray1[index1];
				Index1 += 1;
			} else
			{
				parentArray[i] = myArray2[index2];
				Index2 += 1;
			}
		}
		//myArray1 and myArray3 is left
		Else If ((index1 < myArray1.size() && index3 < myArray3.size()) && index2
>= myArray2.size())
		{
			If (myArray1[index1] <= myArray3[index3])
```

```
				{
					parentArray[i] = myArray1[index1];
					Index1 += 1;
				} else
				{
					parentArray[i] = myArray3[index3];
					Index3 += 1;
				}
			}
			//myArray2 and myArray3 is left
			Else If ((index2 < myArray2.size() && index3 < myArray3.size()) && index1
>= myArray1.size())
			{
				If (myArray2[index2] <= myArray3[index3])
				{
					parentArray[i] = myArray2[index2];
					Index2 += 1;
				} else
				{
					parentArray[i] = myArray3[index3];
					Index3 += 1;
				}
			}

			//Check myArray1 beats both
			Else If (myArray1[index1] <= myArray2[index2] && myArray1[index1] <=
myArray3[index3])
			{
				parentArray[i] = myArray1[index1];
				//Increment to next element
				Index1 += 1;
			} else If (myArray2[index2] <= myArray1[index1] && myArray2[index2] <=
myArray3[index3])	// myArray2 beats both
			{
				parentArray[i] = myArray2[index2];
				//Increment to next element
				Index2 += 1;
			} else If (myArray3[index3] <= myArray1[index1] && myArray3[index3] <=
myArray2[index2])	// myArray3 beats both
			{
```

```
                parentArray[i] = myArray3[index3];
                //Increment to next element
                Index3 += 1;
            }
        }
        Return parentArray;
}
```

Analysis: I will be analyzing the number of comparison operations.

$M(n + m + p) = {}_{i=0}\Sigma^{n+m+p-1} (28) + 1 \rightarrow (28)((n + m + p - 1) - (0) + 1) + 1 \rightarrow 28(n + m + p) + 1 \rightarrow 1 + 28(n + m + p) \rightarrow M(n + m + p) \in \Theta(n + m + p)$

2. Design and analyze asymptotically an O(nlgn) **transform-conquer** algorithm for the following problem:
    - input: an array *A[lo..hi]* of *n* real values;
    - output: true iff the array contains two elements (at different indices) whose sum is 2020.

A2:

```
Bool isFound(int myArray[])
{
        //Base case, 1 element
        If (myArray.size() == 1)
        {
                Return false; // Can't add one element
        } else  // Recursive case
        {
                //First sort the array
                mergeSort(myArray);
                //Create 2 subarrays, half of the original array, unless it has odd size
                If (myArray.size() % 2 == 0)          // Even size case
                {
                        Int array1[myArray.size() / 2];
                        Int array2[myArray.size() / 2];
                        //Copy elements into 2 subarrays
                        For (int i = 0; i <= myArray.size() - 1; ++i)
                        {
                                //First half
                                If (i <= (myArray.size() / 2) - 1)
                                {
                                        Array1[i] = myArray[i];
```

```
			} else  // Second half
			{
				Array2[i] = myArray[i];
			}
		}
		//Check answers in lower cases
		Bool isFound1 = isFound(array1);
		Bool isFound2 = isFound(array2);
		//Last element index
		Int lastIndex = myArray.size() - 1;
		//Search starting with first and last element, converging in, until they
meet
		For (int i = 0; i <= myArray.size() / 2; ++i)
		{
			//Compares if i (inner) adds up with lastIndex (outer) to 2020
			If (myArray[i] + myArray[lastIndex] == 2020)
			{
				Return true;
			}
			//Also compare the adjacent elements of inner/outer
			If (myArray[i] + myArray[i + 1] == 2020)
			{
				Return true;
			} else if (myArray[lastIndex] + myArray[lastIndex - 1] ==
2020)
			{
				Return true;
			}
			//i will be incremented, and lastIndex will be decremented
			lastIndex -= 1;
		}
		//This will cover the bridges
		//After all of those checks fail, check the last line of defense
		Return isFound1 || isFound2;
	} else  // Odd size case
	{
		Int array1[myArray.size() / 2];
		Int array2[myArray.size() / 2];
		//Copy elements into 2 subarrays
		For (int i = 0; i <= myArray.size() - 1; ++i)
```

```
        {
            //First half
            If (i <= (myArray.size() / 2) - 1)
            {
                Array1[i] = myArray[i];
            } else  if (i != myArray.size() / 2)    // Second half, as long as i isn't middle element
            {
                Array2[i] = myArray[i];
            }
        }
        //Check answers in lower cases
        Bool isFound1 = isFound(array1);
        Bool isFound2 = isFound(array2);
        //Last element index
        Int lastIndex = myArray.size() - 1;
        //Search starting with first and last element, converging in, until they meet
        For (int i = 0; i <= myArray.size() / 2; ++i)
        {
            //Compares if i (inner) adds up with lastIndex (outer) to 2020
            If (myArray[i] + myArray[lastIndex] == 2020)
            {
                Return true;
            }
            //Also compare the adjacent elements of inner/outer
            If (myArray[i] + myArray[i + 1] == 2020)
            {
                Return true;
            } else if (myArray[lastIndex] + myArray[lastIndex - 1] == 2020)
            {
                Return true;
            }
            //i will be incremented, and lastIndex will be decremented
            lastIndex -= 1;
        }
        //This covers if there is an odd number of elements, middle check
        //Check every element with the middle element, but overlaps middle
        For (int i = 0; i <= myArray.size() - 1; ++i)
```

```
                {
                        If (myArray[(myArray.size() / 2) + 1] + myArray[i] == 2020)
                        {
                                Return true;
                        }
                }
                //This will cover the bridges
                //After all of those checks fail, check the last line of defense
                Return isFound1 || isFound2;
        }
    }
}
```

Analysis: I will be analyzing the number of addition operations

$M(1) = 0$

$$M(n) = \Theta(n (\lg n)) + \sum_{i=0}^{n-1} (1) + 2M(n / 2) + \sum_{i=0}^{n/2} (5) + \sum_{i=0}^{n-1} (3)$$

$$= \Theta(n (\lg n)) + (1)(n) + 2M(n / 2) + (5)((n / 2) + 1) + (3)(n)$$

$$= 2M(n / 2) + 9n + (5n / 2) + \Theta(n (\lg n))$$

Using the master theorem, a = 2, b = 2, c = 0, and d = 1. So, a ? $b^d \rightarrow 2 ? 2^1 \rightarrow 2 = 2^1$, which means that $M(n) \in \Theta(n^d (\lg n)) \rightarrow M(n) \in \Theta(n (\lg n))$

3. Design and analyze asymptotically a **transform-conquer** algorithm for the following problem:
    - input: an array *A[lo..hi]* of *n* **double** numbers;
    - output: an array representing the **min**-heap whose elements are elements of A.

A3:

```
double* minHeap (double myArray[])
{
    //Presort the array, least to greatest, applicable for double data type
    mergeSort(myArray);
    //Let this be a min heap, easier if element 0 is empty and first element is at i = 1
    //double newArray[myArray.size() + 1];
    //Copy the array into this new array, skip element 0
    For (int i = 0; i <= myArray.size() - 1; ++i)
    {
        newArray[i + 1] = myArray[i];
    }
    //Now that it is all in order, it basically already represents the min heap, as long as parent
node i is less than its children, such that i > 0 → arr[i] < arr[2i] and arr[(2i + 1)]
```

//Ex: 1 2 3 10 24 30 → 1 < 2 and 3

}

Analysis: I will analyze the number of addition operations. There is only 1 function call happening, which is mergeSort → $_{0}\Sigma^{n-1}$ (1) + Θ(n (lg n)) → n + Θ(n (lg n)) → Θ(n (lg n))