Homework 4

1. Design and analyze asymptotically a **divide-conquer** algorithm for the following problem:
   - input: an array *A[lo..hi]* of real numbers;
   - output: the sum of elements of *A[lo..hi].*

A1:

```
Int sum (int[] myArray)

{

    //Base case, 1 element in array

    If (myArray.size() == 1)

    {

        //Return the only element as the total net sum

        Return myArray[0];

    } else  // Recursive case

    {

        //We will split the array in half

        If (myArray.size() % 2 == 0) // Array size is even, can be evenly divided in half

        {

            //Int sum = 0;

            //We are going to have 2 partitions of the array, each half of the original

            Int array1[myArray.size() / 2];

            Int array2[myArray.size() / 2];

            //Copy the corresponding elements to each array

            For (int i = 0; i <= myArray.size() - 1; ++i)

            {

                If (i =< (myArray.size() / 2) - 1)        // First half of array
```

```
                {
                        Array1[i] = myArray[i];
                } else   // Second half of array
                {
                        Array2[i] = myArray[i];
                }
        }
        Int sum1 = sum(array1);
        Int sum2 = sum(array2);
        Return sum1 + sum2;
} else If (myArray.size() % 2 != 0) // Array size is odd
{
        //Int sum = 0;
        //We are going to have 2 partitions of the array, each half of the original
        Int array1[myArray.size() / 2];
        Int array2[myArray.size() / 2];
        //Copy the corresponding elements to each array
        For (int i = 0; i <= myArray.size() - 1; ++i)
        {
                If (i =< (myArray.size() / 2) - 1)          // First half of array
                {
                        Array1[i] = myArray[i];
                } else   // Second half of array
                {
```

```
                    If (i != (myArray.size() / 2))     // Skip the middle pivot
                    {
                        Array2[i] = myArray[i];
                    }
                }
            }
            Int sum1 = sum(array1);
            Int sum3 = myArray[(myArray.size() / 2)];     // Middle pivot
            Int sum2 = sum(array2);
            Return sum1 + sum2 + sum3;
        }
    }
}
```

Analysis: I will be analyzing the amount of integer comparisons

M(1) = 1

M(n) $= 2 + 1 + ({}_{i=0}\Sigma^{n-1})(3) + 2M(n / 2)$, for $n > 1$

$= 2 + 1 + (n - 1 - 0 + 1)(3) + 2M(n / 2)$

$= 2 + 1 + 3n + 2M(n / 2) = 2M(n / 2) + (3n + 3)$

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

Side note: M(n) = 2 + [(i = 0En - 1)(2) + 1] + 1 + (i = 0En - 1)(3), for n > 1 [] means the part in even case, didn't account for recursive calls (mistake), not final answer

2. Design and analyze asymptotically an $O(\lg N)$ **divide-conquer** algorithm for the following problem:
   - input: a nonnegative integer N;
   - output: 3^N.

A2:

```
Int multiply (unsigned int N)
{
	//Base case
	If (N == 1)
	{
		Return 3;
	} else	// Recursive case
	{
		//We will split the product segment in half
		If (N % 2 == 0) // N is even, can be evenly divided in half
		{
			//Int product = 1;
			//We are going to have 2 partitions of the product, each half of the original
			Int product1 = multiply(N / 2);
			Int product2 = multiply(N / 2);
			Return product1 * product2;
		} else If (N % 2 != 0) // N is odd
		{
			//Int product = 1;
			//We are going to have 2 partitions of the product, each half of the original
			Int product1 = multiply(N / 2);
			Int product3 = 3;		// Middle term in the product
			Int product2 = multiply(N / 2);
```

```
                    Return product1 * product2 * product3;

          }

     }

}
```

Analysis: I will be analyzing the amount of multiplications, disregarding the mod.

$M(1) = 0$

$M(n) \quad = 2M(n / 2) + 2$, for $n > 1$

Using the master theorem, $a = 2$, $b = 2$, $c = 0$, and $d = 0$. So, a ? $b^d \rightarrow 2$? $2^0 \rightarrow 2 > 1 \rightarrow a > b^d$, which means that $M(n) \in \Theta(n^{\log_b a}) \rightarrow M(n) \in \Theta(n^{\log_2 2})$

Fix #2: Reduce to only 1 recursive call / Possible! This should be analysis

Second Attempt At #2

```
Int multiply (unsigned int N)

{

     //Base case

     If (N == 1)

     {

          Return 3;

     } else   // Recursive case

     {

          //We will split the product segment in half

          If (N % 2 == 0) // N is even, can be evenly divided in half

          {

               //Int product = 1;

               //We are going to have 2 partitions of the product, each half of the original
```

```
            Int product1 = multiply(N / 2);

            //Instead of doing another recursive call, we know product2 = product1

            Int product2 = product1;

            Return product1 * product2;

        } else If (N % 2 != 0) // N is odd

        {

            //Int product = 1;

            //We are going to have 2 partitions of the product, each half of the original

            Int product1 = multiply(N / 2);

            Int product3 = 3;       // Middle term in the product

            //Instead of doing another recursive call, we know product2 = product1

            Int product2 = product1;

            Return product1 * product2 * product3;

        }

    }

}
```

M(1) = 0

M(n) = M(n / 2) + 3, for n > 1

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

3. Design and analyze asymptotically an $O(n\lg n)$ **divide-conquer** algorithm for the following problem (do not call a sorting algorithm):
    - input: an array $A[lo..hi]$ of n real numbers;
    - output: the smallest difference (in absolute value) between two elements in $A[lo..hi]$.

A3:

```
Int difference (float[] myArray)
{
	//Base case, 1 element in array

	If (myArray.size() == 1)

	{

		Return myArray[0];

	} else	// Recursive case

	{

		//We will split the array in half

		If (myArray.size() % 2 == 0) // Array size is even, can be evenly divided in half

		{

			//The smallest difference in the bridges cases

			float bridgeDifference = 0;

			//We are going to have 2 partitions of the array, each half of the original

			float array1[myArray.size() / 2];

			float array2[myArray.size() / 2];

			//Copy the corresponding elements to each array

			For (int i = 0; i <= myArray.size() - 1; ++i)

			{

				If (i =< (myArray.size() / 2) - 1)		// First half of array

				{

					Array1[i] = myArray[i];

				} else	// Second half of array

				{

					Array2[i] = myArray[i];
```

```
		}
	}
	float difference1 = difference(array1);
	float difference2 = difference(array2);
	//Initialize the bridgesDifference to start doing comparisons
	bridgesDifference = array1[0] - array2[0];
	//This simulates an absolute value, flips sign
	If (difference1 < 0)
	{
		difference1 *= -1;
	}
	If (difference2 < 0)
	{
		difference2 *= -1;
	}
	If (bridgesDifference < 0)
	{
		bridgesDifference *= -1;
	}
	//This compares all of the values between subarrays, the bridges
	For (int i = 0; i <= myArray.size() - 1; ++i)
	{
		//You only need to compare 7 values in the bridge
		For (int j = 0; j <= 7; ++j)
```

```
        {
            //Current difference beats bridgeDifference, update
            If (bridgeDifference > array1[i] - myArray[i + j])
            {
                bridgeDifference = myArray[i] - myArray[i + j];
            }
            If (bridgesDifference < 0)
            {
                bridgesDifference *= -1;
            }
        }
    }
    If (difference1 <= difference2)
    {
        If (difference1 <= bridgesDifference)
        {
            Return difference1;
        } else
        {
            Return bridgesDifference;
        }
    } else
    {
        If (difference2 <= bridgesDifference)
```

```
                {
                        Return difference2;
                } else
                {
                        Return bridgesDifference;
                }
        }
} else If (myArray.size() % 2 != 0) // Array size is odd
{
        //The smallest difference in the bridges cases
        float bridgeDifference = 0;
        //We are going to have 2 partitions of the array, each half of the original
        float array1[myArray.size() / 2];
        float array2[myArray.size() / 2];
        //Copy the corresponding elements to each array
        For (int i = 0; i <= myArray.size() - 1; ++i)
        {
                If (i =< (myArray.size() / 2) - 1)        // First half of array
                {
                        Array1[i] = myArray[i];
                } else   // Second half of array
                {
                        If (i != (myArray.size() / 2))     // Skip the middle pivot
                        {
```

```
                    Array2[i] = myArray[i];
                }
            }
        }
        float difference1 = difference(array1);
        float difference2 = difference(array2);
        //Initialize the bridgesDifference to start doing comparisons
        bridgesDifference = array1[0] - array2[0];
        //Int sum1 = sum(array1);
        float pivDifference = myArray[(myArray.size() / 2)];   // Middle pivot
        //Int sum2 = sum(array2);
        //This simulates an absolute value, flips sign
        If (difference1 < 0)
        {
            difference1 *= -1;
        }
        If (difference2 < 0)
        {
            difference2 *= -1;
        }
        If (pivDifference < 0)
        {
            pivDifference *= -1;
        }
```

```
If (bridgesDifference < 0)
{
	bridgesDifference *= -1;
}
//This compares all of the values between subarrays, the bridges
For (int i = 0; i <= myArray.size() - 1; ++i)
{
	//You only need to compare 7 values in the bridge
	For (int j = 0; j <= 7; ++j)
	{
		//Current difference beats bridgeDifference, update
		If (bridgeDifference > array1[i] - myArray[i + j])
		{
			bridgeDifference = myArray[i] - myArray[i + j];
		}
		If (bridgesDifference < 0)
		{
			bridgesDifference *= -1;
		}
	}
}
If (difference1 <= difference2)
{
	If (difference1 <= bridgesDifference)
```

```
        {
            Return difference1;
        } else if (bridgesDifference < difference1)
        {
            If (bridgesDifference < pivDifference)
            {
                Return bridgesDifference;
            } else
            {
                Return pivDifference;
            }
        }
    } else
    {
        If (difference2 <= bridgesDifference)
        {
            Return difference2;
        } else if (bridgesDifference < difference2)
        {
            If (bridgesDifference < pivDifference)
            {
                Return bridgesDifference;
            } else
            {
```

```
                              Return pivDifference;
                         }
                    }
               }
          }
     }
}
```

Analysis: I will be analyzing the amount of comparisons

M(1) = 1

$$M(n) = 1 + 1 + [1 + ((\sum_{i=0}^{n-1})(2) + 1) + 2M(n) + 4 + (\Theta(n)) + 4]$$

$$= 1 + 1 + [1 + ((n - 1 - 0 + 1)(2) + 1) + 2M(n) + 4 + (\Theta(n)) + 4]$$

$$= 3 + (2n + 1) + 2M(n) + 4 + (\Theta(n)) + 4]$$

$$= 2M(n / 2) + 2n + 12 + \Theta(n) = 2M(n / 2) + \Theta(n)$$

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