1) (10 pts) ANL (Algorithm Analysis)

With proof, determine the Big-Oh run time of the function, f, below, in terms of the input parameter n. (Note: You may use results from algorithms studied previously in COP 3502 without restating the full proof of run time.)

```
int f(int array[], int n) {
    return frec(array, 0, n-1);
}

int frec(int array[], int lo, int hi) {

    if (lo == hi) return array[lo];

    int left = frec(array, lo, (lo+hi)/2);
    int right = frec(array, (lo+hi)/2+1, hi);

    int i, lCnt = 0, rCnt = 0;
    for (i=lo; i<=hi; i++) {
        if (abs(array[i]-left) < abs(array[i]-right))
            lCnt++;
        else
            rCnt++;
    }
    if (lCnt > rCnt) return lCnt;
    return rCnt;
}
```

The function f is a wrapper function that calls the recursive function frec. f takes in an array of size n while frec takes in a subsection of an array of size hi-lo+1. Let T(n) be the run time of the function frec where hi-lo+1 = n.

To determine what T(n) equals, first note that two recursive calls are made, each to arrays of size n/2. Each of these recursive calls, by definition, takes T(n/2) time. This is follows by a for loop that runs n times, inside of which there are only a few O(1) operations. Thus, we add O(n) to the runtime of the function for the second portion of the code. Thus, our total tally is:

$T(n) = T(n/2) + T(n/2) + O(n)$
$T(n) = 2T(n/2) + O(n)$

If one recognizes this as the recurrence of Merge Sort solved in COP 3502, one can state that the result of solving this recurrence relation is **T(n) = O(nlgn).** Alternatively, either the Master Theorem or Iteration Technique can be used to arrive at the final solution for the recurrence.

Grading: 2 pts for setting up any sort of recurrence relation, 2 pts for recognizing that there are two recursive calls, 2 pts for recognizing that the input size for the recursive calls is n/2, 2 pts for recognizing that there is O(n) extra work in the function and 2 pts for the final solution.