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:

```
int f(int array[], int n) {

    int i, t = 0, a = 0, b = n-1;
    while (a < b) {

        for (i=a; i<=b; i++)
            t += array[i];

        if (array[a] < array[(a+b)/2])
            b = (a+b)/2-1;
        else
            a = (a+b)/2+1;

    }
    return t;
}
```

In your work, you may use the following result: $\sum_{i=0}^{\infty}(\frac{1}{2})^i = 2$.

The main loop in the function is a while loop. The while loop is controlled by a and b, which initially start as the low and high indexes into array (assuming its of length n). The if else statement essentially resets either a or b to be halfway between the two. In essence, the difference between a and b is being divided by 2, roughly, at each iteration of the while loop. Thus, it follows that the while loop will run roughly $\log_2 n$ times. It turns out that knowing this isn't critical to the Big-Oh analysis.

Now, at each iteration of the while loop, we see that a for loop is run and that the assignment statement inside the loop runs b - a + 1 times, which essentially equals the difference between a and b.

Clearly, the first time, this difference is n (with the +1 factor). The following time, this difference will be roughly $\frac{n}{2}$, since the difference between a and b gets divided by 2 with each while loop iteration. On the third while loop iteration, the inner for loop will run roughly $\frac{n}{4}$ times. In general, on the k^{th} iteration of the while loop, the inner for loop runs no more than $\frac{n}{2^{k-1}}$ times. We can claim it's an upper bound because a and b are reset to be slightly less than half of the range with the -1 and +1 respectively. It follows that the run time of the code is less than or equal to

$$\sum_{k=1}^{\infty} {}^{n}/_{2^{k-1}} = \sum_{i=0}^{\infty} {}^{n}/_{2^{i}} = n\sum_{i=0}^{\infty}(\tfrac{1}{2})^i = 2n = O(n)$$

Grading: 3 pts for realizing that the difference between a and b is being divided by 2 through each loop iteration, 3 pts for determining that the relevant sum has the pattern n + n/2 + n/4 + … 4 pts for setting up the sum, completing it and getting a Big-Oh bound. Max 7 pts for a valid O(nlgn) justification.