1) (10 pts) ANL (Algorithm Analysis)

Consider the following problem: You are given a set of weights, $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$ and a target weight T. The target weight is placed on one side of a balance scale. The problem is to determine if there exists a way to use some subset of the weights to add on either side of the balance so that the scale will perfectly balance or not. For example, if T = 12 and the set of weights was {6, 2, 19, 1}, then one possible solution would be to place the weights 6 and 1 on the same side of the balance as 12 and place the weight 19 on the other side. Below is a function that solves this problem recursively, with a wrapper function to make the initial recursive call. In terms of n, the size of the input array of weights, **with proof,** determine the **worst case** run time of the wrapper function. **(Note: Since only the run time must be analyzed, it's not necessary to fully understand WHY the solution works. Rather, the analysis can be done just by looking at the structure of the code.)**

```
int makeBalance(int weights[], int n, int target) {
    return makeBalanceRec(weights, n, 0, target);
}
int makeBalanceRec(int weights[], int n, int k, int target) {
    if (k == n) return target == 0;
    int left = makeBalanceRec(weights, n, k+1, target-weights[k]);
    if (left) return 1;
    int right = makeBalanceRec(weights, n, k+1, target+weights[k]);
    if (right) return 1;
    return makeBalanceRec(weights, n, k+1, target);
}
```

In the worst case, a recursive call with input k makes 3 recursive calls with input k+1. Upon examining the code, what we really see is that as k increases, the subarray of weights we are considering decreases in length. Thus, a better way of interpreting the code is that given an array of size n, at worst, 3 recursive calls are made on an array of size n-1. The extra work beyond the recursive calls is all constant work (some additions and if statements). Thus, if we let T(n) represent the run time of the wrapper function in terms of the size of the input array, the worst case run time of the function is represented by the recurrence relation: T(n) = 3T(n-1) + 1, (note for a more accurate analysis one may say +c, a constant.)

Iterate twice:

T(n) = 3(3T(n-2) + 1) + 1 = 9T(n-2) + 4
T(n) = 9(3T(n-3) + 1) + 4 = 27T(n-3) + 13

In general after k steps we get: $T(n) = 3^k T(n - k) + \sum_{i=0}^{k-1} 3^i$. Since T(0) = 1 (base case), plug in n = k into the recurrence to get:

$T(n) = 3^n T(0) + \sum_{i=0}^{n-1} 3^i = 3^n + \sum_{i=0}^{n-1} 3^i = \sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{3-1}$ = **$O(3^n)$.** (Note: Since $O(3^{n-1})$ and $O(3^{n+1})$ are equivalent to $O(3^n)$, these get full credit also.) Also, okay just to look at the bottom level…

Grading: 4 pts for recurrence relation, 4 pts for solving it, 2 pts for the final answer. May use some leeway to award partial credit, but please give 0/10 for $O(n^2)$ or other polynomial answers.