We first define two auxiliary functions `GetChicken` and `NegateWord`. `GetChicken` will tell you, for a given value and a word you says, how many chicken you could get. The `NegateWord` simply return the other word given your current word.

```
GetChicken(value, word):
    if word = "Ring" return value
    if word = "Ding" return -value

NegateWord(word):
    if word = "Ring" return "Ding"
    if word = "Ding" return "Ring"
```

Then we use a 3-d memoization array called `MaxChickenArray[1..n, "Ring" or "Ding", 0..3]`. The syntax `"Ring" or "Ding"` means the second index of the `MaxChickenArray` could either be `Ring` or `Ding`. `MaxChickenArray[i, word, count]`, means that given we have already said `word` `count` number of times before, the maximum chicken we could get starting from the `A[i]` to `A[n]`, inclusive. Few things to notice:

- Since each `MaxChickenArray[i, word, count]` depends on `MaxChickenArray[i + 1, "Ring" or "Ding", 0..3]` (this means all the elements in `MaxChickenArray` that could be access by the index triplet where first index should be `i + 1` and second index could be either `Ring` or `Ding`, and third index could be either `0, 1, 2, 3`). That means we need to calculate the `MaxChickenArray[i]` in the order where `i` goes from `n` to `1`. (The order of the `word` and `count` doesn't really matter in this case)
- We return `MaxChickenArray[1, "Ring", 0]` (The `MaxChicken[1, "Ding", 0]` also works). This element in the array means the maximum number of chicken we could get if we start in the `A[1]` and goes to `A[n]`, given that we previous have say `Ring` 0 times (or `Ding` 0 times). (This basically means we haven't say anything before we start). This is exactly what the problem asks us to find.

The pseudocode is shown below:

```
MaxChicken(A[1..n]):
    MaxChickenArray[1..n, "Ring" or "Ding", 0..3]

    EvaluateSubProblem(i, word, count):
        if i = n:
            if count = 3:
                return GetChicken(A[i], NegateWord(word))
            else:
                return max(
                    GetChicken(A[i], word),
                    GetChicken(A[i], NegateWord(word))
                )
        else:
            if count = 3:
                return getChicken(A[i], NegateWord(word)) +
                        MaxChickenArray[i + 1, NegateWord(word), 1]
            else:
```

```
                return max(
                    getChicken(A[i], word) +
                        maxChickenArray[i + 1, word, count + 1]
                    getChicken(A[i], NegateWord(word)) +
                        MaxChickenArray[i + 1, NegateWord(word), 1]
                )


    for i ← n to 1:
        foreach word in ["Ring", "Ding"]:
            for count ← 0 to 3:
                MaxChickenArray[i, word, count] ← EvaluateSubProblem(i, word,
count)

    return MaxChickenArray[1, "Ring", 0]
```

The total time complexity is just $O(n)$. We see that the function `EvaluateSubProblem` takes `O(1)` time, and three for loops will execute `EvaluateSubProblem` for $n \cdot 2 \cdot 4 = 8n$ times, which have the total time complexity of $8n \cdot O(1) = O(n)$