Problem 3.

Define a procedure `test-strategy` that tests two strategies by playing a specified number of simulated Twenty-One games using the two strategies. `Test-strategy` should return the number of games that were won by the player (and thus lost by the house). For example,

```
(test-strategy (stop-at 16) (stop-at 15) 10)
```

should play ten games of Twenty-One, using the value returned by `(stop-at 16)` as the player's strategy and the value of `(stop-at 15)` as the house strategy. It should return a non-negative integer indicating how many games were won by the player. Turn in a listing of your procedure and some sample results.

Answer.

As is often the case, there are two ways to implement the procedure `test-strategy`, the recursive version and the iterative version.

As a way for starting, lat's first come to deal with the recursive one. Remember the principle to design a recursive procedure:

i. Wishful thinking.

ii. Decompose the problem.

iii. Identify non-decomposable problems.

By wishful thinking, suppose that we can solve the problem, but only for versions of the problem smaller than the current one. That is the smaller version of `test-strategy` just plays one less round of game compare to the original one.

Given that assumpsion, the second stage proceeds to convert that simplified solution to the desired solution. We saw this with `test-strategy`, where we combine the solution to a smaller version of `test-strategy`, plus the result of the current round, to create the full version of `test-strategy`.

```
(define (test-strategy player-strategy house-strategy n)
  (+ <current-result>
     (test-strategy player-strategy house-strategy (- n 1))))
```

Now, it time to consider the simplest case where I can solve directly, without using wishful thinking. In the case of `test-strategy`, that is just knowing that when it comes to play only a single round of the game, number of games that were won by the player is simply the result of the current round.

```
(if (< n 2)
    <current-result>
    (...))
```

However, there is still a few problems left to tackle: how to represent the result of the current round? And how can we denote the number of games that were won by the player? Well, note what the game convey to us about its rule: the procedure `twenty-one` returns 1 if the player wins the simulated game and 0 if the house wins. Hence, this solve the first problem. On the other hand, `test-strategy` should return the number of games that were won by the player. This indicates that the number of the game that won by the player is simply the result accumulated from every round of the game. Therefore, we can denote the result of the current round by setting a local variable, namely `pw`, and we the game ends, the value of `pw` is just the the number of games that were won by the player.

After assembling, the building blocks, we have craft, our `test-strategy` procedure turns out to be:

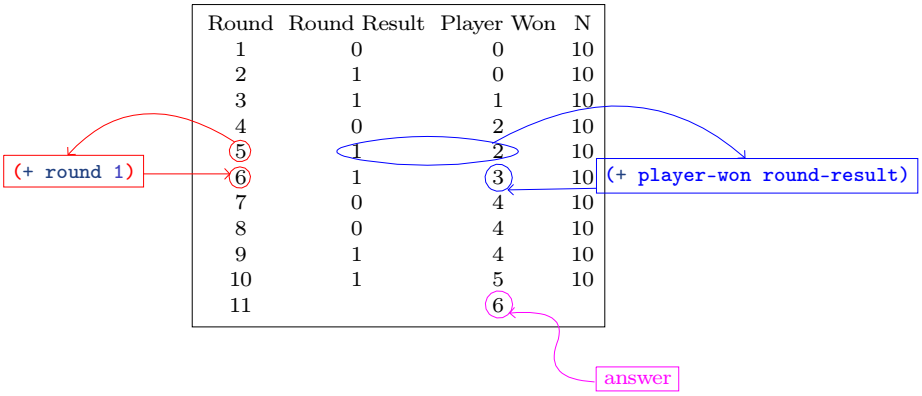
```
(define (test-strategy player-strategy house-strategy n)
  (let (pw (twenty-one player-strategy house-strategy))
    (if (< n 2)
        pw
        (+ pw
           (test-strategy player-strategy house-strategy (- n 1))))))
```

Now, let's come to design the iterative version of procedure `test-strategy`. To do this, we'd better figure out a way to accumalate partial answer, that is, how many variables are needed to depict the computation. Well, in this process, we have to keep track of 4 things, the current round, the number of games that were won by the player, the result of the current round and the total number of games. Suppose that both of the opponents have played 10 rounds of games in which the player took 6.

As is shown in Table 1, the rule for getting the next value of round is simply increasing it by 1. In order to figure out how many rounds have been won by the player, just add the current round result to the record that keeps the number of rounds formerly won by the player. Further more, as is indicated by the table, the number of rounds that won by the player reveals to be our final result when the value exceeds N, here it is 10. Finally, we need a way to start, that's easy, just set the initial value of Round, Round Result and Player Won to 1, 0 and 0 with respect, as is shown in Table 1.

Having done the analysis above, now we can write the procedure `test-strategy` in a quite clear way:

```
(define (test-strategy player-strategy house-strategy n)
  (define (play round pw)
    (if (> round n)
        pw
        (play (+ round 1)
              (+ pw
                 (twenty-one player-strategy house-strategy)))))
  (play 1 0))
```



Round	Round Result	Player Won	N
1	0	0	10
2	1	0	10
3	1	1	10
4	0	2	10
5	1	2	10
6	1	3	10
7	0	4	10
8	0	4	10
9	1	4	10
10	1	5	10
11		6	

Table 1. The Evoluation of Process in Evaluating `(test-strategy (stop-at 16) (stop-at 15) 10)`