**Ejercicio 1**

1. Esta implementación me parece adecuada debido a que no se repite, es decir, no vuelve a hacer el mismo Fibonacci dos veces y eso impacta bastante en la rapidez del algoritmo. El Fibonacci recursivo que hicimos en clase sin memoria, con solo pasarle el número 30 demoraba aproximadamente 2 minutos, pero de esta manera, guardando los resultados en memoria cambia radicalmente el asunto.
2. Yo creo que el costo de memoria que tiene este algoritmo es el Stack y la HashTable que almacena todos los Fibonacci realizados. El Stack debido a que al hacer llamadas recursivas se irán apilando nuevos frames (o llamadas de función) a dicho Stack. Esto no se si será un problema grave o un costo, pero no creo que sea buena práctica tener un Stack muy cargado.

Luego la HashTable porque allí se irán guardando todos los Fibonacci anteriores a un cierto N que el usuario pase, por ende, mientras mas grande sea el N más elementos tendrá esa HashTable.

1. En cuanto a las limitaciones que puede tener creo que lo mencionado anteriormente sobre la memoria podría ser una limitación, mientras más grande el N más llamadas recursivas y por ende un Stack mas cargado.

**Ejercicio 2**

1. Esta implementación es otra forma de resolver el problema que también es válida. Lo resuelve de una forma iterativa en la que define el Fibonacci anterior y el actual, los cuales va redefiniendo en cada iteración para poder llegar al resultado esperado. Divide o bifurca el problema en 2 partes en las que si el n = 0 el código tomará un camino y en caso contrario iterará hasta llegar al resultado.
2. En cuanto a variables no presenta ningún costo considerable de memoria. Lo que se me ocurre es que cuando se le pase un N demasiado grande habrá una iteración que se repita esa N cantidad de veces, por lo que podría verse afectado en algún momento. Aunque dentro de todo, esta forma de resolverlo es bastante adecuada y no creo que presente un costo grande de memoria.
3. Lo mencionado anteriormente. Cuando más grande sea la entrada, más iteraciones deberá hacer. Aunque como comenté, no creo que esto sea un problema ya que con un N demasiado grande incluso lo resuelve demasiado rápido.
4. El anterior necesita guardar los resultados para ser eficiente, este los hace uno por uno hasta llegar al resultado. Además, como este no es recursivo no tendría el Stack demasiado cargado ya que no se llama a si mismo para obtener el resultado.