3.1:Create a memoize function that remembers previous inputs and stores them in cache so that it
won’t have to compute the same inputs more than once. The function will take an unspecified
number of integer inputs and a reducer method.

```
let cache = {};
const memoizedAdd = () => {

 return (n , m) => {
   if (n+m in cache) {
     console.log('Fetching from cache');
     return cache[n+m];
   }
   else {
     console.log('Calculating result');
     let result = n + m ;
     cache[n + m] = result;
     return result;
   }
 }
}
// returned function from memoizedAdd
const newAdd = memoizedAdd();
console.log(newAdd(5,6)); // calculated
console.log(newAdd(5,6)); // cached
```

Exercise 3.2:
Create 3 simple functions where call, bind and apply are used. The intention of this exercise is
to understand how they work and their differences.
**call() :**
Example:-

```
Let obj = {num:2};
Function add(a,b){
Return this.num= a+b;
}
 console.log(add.call(obj, 3,2));
```

## Apply():

Example:-

```
Let obj = {num:2};
Function add(a,b){
Return this.num= a+b;
}
console.log(apply(obj,[3,2]))
```

## Bind():

Example:-

```
Let obj = {num:2};
Function add(a,b){
Return this.num= a+b;
}

Const func = add.bind(obj, 3,5);
func();
```

```
//creating a increment function using clouser and encapsulation
function createIncrement () {
//Initilizing count
let count = 0;
//Encapsulation function count will not bhe increment outside this function
function increment () {
count++;
}
let message = `Count is ${count}`; //printing count
//Creating a clouser function log
function log() {
console.log(message);
}
return [increment,log];
}
const [increment,log] = createIncrement();
Increment ();
Increment ();
Increment ();
Log ();
// What is logged?
//logged is a cluster function .
```

Exercise 3.4
Refactor the above stack implementation, using the concept of closure, such that there is no
way to access items array outside of createStack() function scope:

Solution:-
```
function createStack(){
  let list = [];

  function push(value){
      list.push(value);
      console.log(list);
  }
  function pop(){
      list.pop();
      console.log(list);
  }

  return { push, pop
  };
}
const stack = new createStack;
stack.push(10); //pushing 10 to array list<< [10]
stack.push(5); //pushing 5 to array list<< [10,5]
stack.pop(); //poping 5 from the array list<<[10]
stack.list ; //undefined
```