Problemes NodeJS

1. Crea una funció **f1** que rebi un paràmetre **a** i que l'escrigui a la consola fent servir la funció **log** de l'objecte **console**.

   Exemple d'us:
   ```
   f1(3)
   ```
   Resultat:
   ```
   3
   ```

2. Crea una funció **f2** que rebi un paràmetre **a** i que retorni `2 * a` si `a >= 0` i `-1` en cas contrari.

   Exemple d'us:
   ```
   f2(3)
   ```
   Resultat:
   ```
   6
   ```

   Exemple d'us:
   ```
   f2(-2)
   ```
   Resultat:
   ```
   -1
   ```

3. Crea una funció **f3** que rebi una llista com a primer paràmetre i retorni una llista.

   ```
   llista2 = f3(llista)
   ```

   Cada element **y** de **llista2** ha de ser el resultat d'aplicar

   ```
   f2(x) + 23
   ```

   a cada element **x** de **llista**, on **f2** és la funció de l'exercici anterior.

   Exemple d'us:
   ```
   f3([1,2,3])
   ```
   Resultat:
   ```
   [25,27,29]
   ```

4. Afegeix una nova funció **printaki** a l'objecte **console** que imprimeixi "aqui" per consola.

   Exemple d'us:
   ```
   console.printaki()
   ```
   Resultat:
   ```
   aqui
   ```

5. Primer fes una funció **f4** que sumi dos números (i.e., **f4(a,b)** $\mapsto$ **a + b**), i fes una llista

   ```
   llistaA = [1,2,3,4]
   ```

   Fes servir **llistaB = llistaA.map(...)** per sumar 23 a cada element de **llistaA**, i fes servir **f4** per fer la suma. Indicació: et caldrà fer una funció addicional, ja que no es pot fer servir **f4** directament.

   Exemple d'us:
   ```
   llistaB
   ```
   Resultat:
   ```
   [24,25,26,27]
   ```

6. Crea una funció **f5** que agafi un objecte i dues funcions per paràmetres, anomenats respectivament **a**, **b**, i **c**:

   ```
   f5 = function(a, b, c) {
           // a és un objecte, b és una funció, i c és una funció.
   }
   ```

   La funció **f5** ha d'aplicar la funció **b** a l'objecte **a**. El resultat se li ha de passar a **c**. La funció **c** ha de ser una funció *callback* amb un paràmetre (i.e., aquesta funció s'ha de cridar quan la feina que faci **f5** s'hagi acabat, i el resultat de la feina se li ha de passar com a paràmetre).

   Exemple d'us:
   ```
   f5(1, f2, function(r) { console.log(r) })
   ```
   Resultat:
   ```
   2
   ```

7. Afegeix una nova funció **printaki2** a l'objecte **console** que imprimeixi per consola "aqui 1", "aqui 2", "aqui 3", etc. És a dir "aqui {número anterior + 1}". No facis servir cap variable global (ni cap objecte global) per guardar el comptador; fes servir una clausura.

   Exemple d'us:
   ```
   console.printaki2()
   ```
   Resultat:
   ```
   aqui 1
   ```

   Exemple d'us:
   ```
   console.printaki2()
   ```
   Resultat:
   ```
   aqui 2
   ```

8. Crea una funció **f6** que tingui dos paràmetres: una llista de noms d'arxiu i una funció callback.

   ```
   f6 = function(llista, callback_final) { ... }
   ```

   La funció **f6** ha de llegir els arxius anomenats a **llista** i ha de crear una nova llista **resultat** amb el contingut d'aquests arxius. I.e., cada element de **resultat** ha de ser el contingut d'un dels arxius.

   Fes servir **.forEach(...)**, **fs.readFile(filename, callback)**. Per afegir un element al final d'una llista pots usar **Array.push(..)**.

   Quan la funció hagi acabat de llegir *tots* els arxius, s'ha de cridar la funció callback que **f6** rep com a paràmetre (i.e., **callback_final**) amb la llista resultant. Fixa't en que quan s'ha cridat l'última funció callback passada a **fs.readFile(...)** és quan realment s'han acabat de llegir tots els arxius. Fixa't també en que l'última callback del **fs.readFile** que s'executa no té perquè ser la que se li ha donat al **fs.readFile** a l'última iteració del **.forEach**.

   Nota: el resultat no té perquè seguir l'ordre correcte (depèn de quan vagin acabant els **fs.readFile**).

   Exemple d'us:
   ```
   f6(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Resultat:
   ```
   ['contingut a2.txt', 'contingut a1.txt']
   ```

9. Modifica la funció **f6** de l'exercici anterior perquè l'ordre de la llista **resultat** coincideixi amb l'ordre original de **llista**. És a dir que a cada posició **resultat[i]** hi ha d'haver el contingut de l'arxiu anomenat a **llista[i]**. Anomena aquesta funció modificada com **f7**.

   Fes servir **llista.forEach(function (element, index) {})**.

   Exemple d'us:
   ```
   f7(['a1.txt','a2.txt'], function (result) { console.log(result) })
   ```
   Resultat:
   ```
   ['contingut a1.txt', 'contingut a2.txt']
   ```

10. Explica perquè, a l'exercici anterior, hi podria haver problemes si en comptes de fer això

    ```
    llista.forEach(function (element, index) { /* ... */ } )
    ```

    féssim això altre

    ```
    let index = 0
    llista.forEach(function (element) { /* ... */ index += 1} )
    ```

    Indicació: ...suposant que fem servir **index** al callback de **fs.readFile**.

11. Implementa la funció **asyncMap**. Aquesta funció té la següent convenció d'us:

    ```
    function asyncMap(list, f, callback2) {...}

    function callback2(err, resultList) {...}
    function f(a, callback1) {...}
    function callback1(err, result) {...}
    ```

Fixa't en que **`f(...)`** té la mateixa forma que **`fs.readFile(...)`**.

La funció **`asyncMap`** aplica **`f`** a cada element de **`list`** i crida **`callback2`** quan acaba d'aplicar **`f`** a tots els elements de la llista. La funció **`callback2`** s'ha de cridar, o bé amb el primer **`err != null`** que se li hagi passat a **`callback1`**, o bé amb **`resultList`** contenint el resultat de la feina feta per **`asyncMap`** (en l'ordre correcte).

Indicació: fixa't en els paral·lelismes entre aquest exercici i els anteriors. Intenta entendre que vol dir fer un map asíncron.

Exemple d'us:
```
asyncMap(['a1.txt'], fs.readFile, (a,b)=>console.log(`error: ${a} data:${b}`))
```
Resultat:
```
['content 1']
```

Exemple d'us:
```
asyncMap(['a1.txt', 'a2.txt', 'notExists1.txt', 'notExists2.txt'], fs.readFile,
  (a,b)=>console.log(`error: ${a} data:${b}`))
```

12. Fes un objecte **`o1`** amb tres propietats: un comptador **`count`**, una funció **`inc`** que incrementi el comptador, i una variable **`notify`** que contindrà **`null`** o bé una funció d'un paràmetre. Feu que el comptador cridi la funció guardada a la propietat **`notify`** per "notificar" cada cop que el comptador s'incrementi.

    Indicació: en un patró observador com el que tenim aquí, l'acció de "notificar" es fa cridant la funció.

    Exemple d'us:
    ```
    o1.notify = null; o1.inc()
    ```
    Resultat:

    Exemple d'us:
    ```
    o1.count = 1; o1.notify = function() { console.log("notified") }; o1.inc()
    ```
    Resultat:
    ```
    notified
    ```

12+ Millora l'exercici anterior. Fes que al fer la notificació, s'indiqui el nou valor del comptador a la funció guardada a la propietat **`notify`**.

Exemple d'us:
```
o1.count = 1; o1.notify = function(a) { console.log(a) }; o1.inc()
```
Resultat:
```
2
```

13. Fes el mateix que a l'exercici anterior però fes servir el *module pattern* per amagar el valor del comptador i la funció especificada a **`notify`**. Fes un setter per la funció triada per l'usuari. Anomena l'objecte com **`o2`**.

    El següent codi és un exemple de module pattern que pots reaprofitar:

```
let testModule = (function() {
        let count = 1
        return {
                inc: function() { count++ },
                count: function() { return count }
        }
})();
```

Exemple d'us:
```
o2.setNotify(function (a) { console.log(a) }); o2.inc()
```
Resultat:
```
2
```

14. Converteix l'exemple anterior en una classe i assigna'l a un objecte **`o3`**. En què es diferencien els dos exemples?

    El següent codi és un exemple d'una classe que pots reaprofitar:

```
Counter = function() {
        this.a = 1
        this.inc = function () { this.a++ }
        this.count = function() { return this.a }
```

```
}

new Counter();
```

Exemple d'us:

```
o3.setNotify(function (a) { console.log(a) }); o3.inc()
```

Resultat:

```
2
```

14. [+] Modifica la classe **Counter** anterior per a que només "reveli" els mètodes **inc** i **setNotify**.

15. Fes una nova classe, **DecreasingCounter**, que estengui l'anterior per herència i que faci que el mètode **inc** en realitat decrementi el comptador.

16. Diem que tenim un objecte de "tipus future" si aquest és un objecte de dos camps tal i com es mostra a continuació:

    ```
    let future = { isDone: false, result: null }
    ```

    Aquest objecte representa el resultat d'una operació que pot haver acabat o estar-se executant encara. El camp **isDone** ens indica si l'operació ja ha acabat; inicialment és **false**, i passa a ser **true** quan l'operació ha acabat. El camp **result** és **null** mentre **isDone == false** i conté el resultat de l'operació quan aquesta ja ha acabat.

    Es demana que implementis la funció **readIntoFuture(filename)**. Aquesta funció ha de llegir l'arxiu **filename** fent servir **fs.readFile**, però ha de retornar un objecte de tipus future (amb un **return**) encara que l'operació de lectura no hagi acabat. L'objecte retornat, s'actualitzarà quan l'arxiu s'hagi llegit.

    Exemple d'us:

    ```
    future = readIntoFuture('a1.txt'); console.log(future)
    ```

    Resultat:

    ```
    { isDone: false, result: null }
    ```

    Exemple d'us:

    ```
    future = readIntoFuture('a1.txt');
    setTimeout(function() { console.log(future) }, 1000)
    ```

    Resultat:

    ```
    { isDone: true, result: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 31 0a> }
    ```

17. Suposem que tenim una funció **f** amb la mateixa convenció de crida que **fs.readFile** (això vol dir que **f** podria ser **fs.readFile**).

    Generalitza l'exercici anterior de la següent manera. Fes una funció **asyncToFuture(f)** que "converteixi" la funció **f** en una nova funció equivalent però que només rebi un paràmetre i que retorni un future tal que l'exercici anterior. La funció **asyncToFuture** tornarà aquesta funció (**g**) que encapsula la funció **f**.

    Noteu que **asyncToFuture(fs.readFile)** ha de ser equivalent a **readIntoFuture**.

    Exemple d'us:

    ```
    readIntoFuture2 = asyncToFuture(fs.readFile);
    future2a = readIntoFuture2('a1.txt');
    setTimeout(function() { console.log(future2a) }, 1000)
    ```

    Resultat:

    ```
    { isDone: true, result: <Buffer 63 6f 6e 74 69 6e 67 75 74 20 31 0a> }
    ```

    Exemple d'us:

    ```
    statIntoFuture = asyncToFuture(fs.stat);
    future2b = statIntoFuture('a1.txt');
    setTimeout(function() { console.log(future2b) }, 1000)
    ```

    Resultat:

    ```
    { isDone: true, res : Stats { dev: 64769, mode: 33188, /*...*/ } }
    ```

18. Fes la funció **asyncToEnhancedFuture** que faci el mateix que la funció anterior, però que retorni un objecte de "tipus enhanced future". Els objectes d'aquest tipus tenen els tres camps que es mostren a continuació:

    ```
    enhancedFuture = { isDone: false, result: null,
                       registerCallback: [Function] }
    ```

Els dos primers camps funcionen igual que amb un tipus future dels anteriors. Addicionalment, els objectes de tipus enhanced future tenen un tercer camp **registerCallback**. Aquest camp és una funció que rep un callback per paràmetre (i.e., **enhancedFuture.registerCallback(cb)**) i té un funcionament similar a la funció **setNotify()** vista a exercicis anteriors, és a dir, pot ser us cal una nova propietat de la future (**notify**) que inicialitzareu mitjançant la funció **registerCallback**.

Quan es registra una funció **cb** cridant a **registerCallback(cb)**, l'objecte **enhancedFuture** fa servir la callback registrada (cb) per notificar quan s'ha produit un canvi a **isDone**. Si **isDone** ja es **true** quan es registra el callback amb **registerCallback(cb)**, s'ha de cridar **cb** directament.

La funció **cb** rep un paràmetre per poder accedir als camps d'**enhancedFuture**. De fet:

```
function cb(enhancedFuture) { ... }
```

rep com a paràmetre l'objecte enhancedFuture que ha cridat **cb** mitjançant la funció **notify**.

Exemple d'us:

```
const utfReadFile = (f, c) => fs.readFile(f, 'utf-8', c);
const readIntoEnhancedFuture = asyncToEnhancedFuture(utfReadFile);
enhancedFuture = readIntoEnhancedFuture('a1.txt');
enhancedFuture.registerCallback( function(ef) {console.log(ef) } )
```

Resultat:

```
{ isDone: true, result: 'contingut 1', registerCallback: [Function] }
```

19. Tenim que **f1(callback)** és una funció que rep un paràmetre de callback, i **f2(error, result)** és la funció de callback que faríem servir a **f1**, aleshores volem fer la funció **when** que ha de funcionar de la següent manera.

    La funció **when** separa una funció de callback de la funció que la crida. Fa servir la següent sintaxi:

    ```
    when(f1).do(f2)
    ```

    És a dir que ha de fer el mateix que:

    ```
    f1(f2)
    ```

    Exemple d'us:

    ```
    f1 = function(callback) { fs.readFile('a1.txt', 'utf-8', callback) }
    f2 = function(error, result) { console.log(result) }
    when(f1).do(f2)
    ```

    Resultat:

    ```
    'contingut 1'
    ```

    Fixa't en que **when** retorna un objecte amb un sol camp de nom `do`.

    Nota: si no s'està a un terminal interactiu, cal fer servir **console.log** per veure el resultat.

20. Modifica la solució de l'exercici anterior perquè funcioni així:

    ```
    when(f1).and(f2).do(f3)
    ```

    En aquest cas, **f1** i **f2** són funcions amb un sol callback per paràmetre, i que segueixen la mateixa convenció que **fs.readFile** (i.e., el callback té dos paràmetres, **error** i **result**). La funció **f3** rep quatre paràmetres: **error1**, **error2**, **result1** i **result2**.

    Exemple d'us:

    ```
    f1 = function(callback) { fs.readFile('a1.txt', 'utf-8', callback) }
    f2 = function(callback) { fs.readFile('a2.txt', 'utf-8', callback) }
    f3 = function(err1, err2, res1, res2) { console.log(res1, res2) }
    when(f1).and(f2).do(f3)
    ```

    Resultat:

    ```
    'contingut 1 contingut 2'
    ```

20+. Millora l'exercici anterior. Fes que les funcions **f1** i **f2** es cridin just quan sigui possible. És a dir que la funció **f1** es cridi abans que la funció **and**, i que la funció **f2** es cridi abans que la funció `do`.

    Exemple d'us:

```
f1 = callback => { console.log('f1'); callback(null, 'ok1'); }
f2 = callback => { console.log('f2'); callback(null, 'ok2'); }
f3 = (err1, err2, res1, res2) => console.log('callback', res1, res2);
when(f1)
```

Resultat:

```
f1
```

Exemple d'us:

```
when(f1).and(f2)
```

Resultat:

```
f1
f2
```

Exemple d'us:

```
when(f1).and(f2).do(f3)
```

Resultat:

```
f1
f2
callback ok1 ok2
```

21. Fes la funció **composer** que rebi dues funcions d'un sol paràmetre.

```
composer = function(f1, f2) { ... }
```

El resultat d'executar **composer** ha de ser una tercera funció **f3**, que rebi un paràmetre i que sigui la composició de **f1** i **f2**. És a dir que **f3(x)** fa el mateix que **f1(f2(x))**.

Exemple d'us:

```
f1 = function(a) { return a + 1 }
f3 = composer(f1, f1)
f3(3)

f4 = function(a) { return a * 3 }
f5 = composer(f3, f4)
f5(3)
```

Resultat:

```
5
11
```

Nota: si no s'està a un terminal interactiu, cal fer servir **console.log** per veure el resultat.

22. Converteix l'exercici anterior en asíncron tal com s'explica a continuació. Fes la funció **asyncComposer** que rebi dues funcions: **f1** i **f2**.

```
asyncComposer = function(f1, f2) { ... }
```

En aquest cas, **f1** i **f2** són funcions de dos paràmetres que segueixen la mateixa convenció que **fs.readFile**: el primer paràmetre és un valor qualsevol, i el segón és un callback (que té dos paràmetres: **error** i **result**).

El resultat d'executar **asyncComposer** ha de ser una tercera funció **f3**, que tingui la mateixa convenció de crida que **f1** i que **f2**, i que sigui la composició de **f1** i **f2**. És a dir el callback de **f2** ha de cridar a **f1**, i el callback de **f1** ha de cridar al de **f3**.

Exemple d'us:

```
f1 = function(a, callback1) { callback1(null, a + 1) }
f2 = function(a, callback2) { callback2(null, a + 2) }
f3 = asyncComposer(f1, f2)
f3(3, function(error, result) { console.log(result) } )
```

Resultat:

```
6
```

Exemple d'us:

```
f1 = function(a, callback) { callback(null, a + 1) }
f3 = asyncComposer(f1, f1)
f3(3, function(error, result) { console.log(result) } )
```

Resultat:

```
5
```

Si **error** és diferent de **null** al resultat de **f2**, aleshores el callback de **f3** ha de tornar directament aquest error.

Exemple d'us:

```
f1 = function(a, callback) { callback(null, a + 1) }
f2 = function(a, callback) { callback("error", "") }
f3 = asyncComposer(f1, f2)
f3(3, function(error, result) { console.log(error, result) } )
```

Resultat:

```
'error' 1
```

23. Fes un **p.then(x => console.log(x))** per cadascuna de les promisses **p** que es mostren a continuació. Digues què s'imprimeix per pantalla i el perquè.

    (a) `p = Promise.resolve(0).then(x => x + 1).then(x => x + 2).then(x => x + 4);`

    (b) `p = Promise.reject(0).then(x => x + 1).catch(x => x + 2).then(x => x + 4);`

    (c) `p = Promise.resolve(0).then(x => x + 1).then(x => x + 2).catch(x => x + 4).then(x => x + 8);`

    (d) `p = Promise.reject(0).then(x => x + 1).then(x => x + 2).catch(x => x + 4).then(x => x + 8);`

    (e) `p = Promise.reject(0).then(x => x + 1, null).catch(x => x + 2).catch(x => x + 4);`

24. Fes una funció **antipromise** que rebi una promise per paràmetre i retorni una promise diferent. La promise retornada s'ha de resoldre (resolve) quan la promise original es rebutjada (reject) i viceversa.

Exemple d'us:

```
antipromise(Promise.reject(0)).then(console.log);
```

Resultat:

```
0
```

Exemple d'us:

```
antipromise(Promise.resolve(1)).catch(console.log);
```

Resultat:

```
1
```

Indicació per si us apareix un **UnhandledPromiseRejectionWarning**:

> Vol dir que la promesa s'ha resolt abans que el **.catch** s'hagi executat. És a dir que
>
> ```
> let p = Promise.reject(0);
> setTimeout(() => p.catch(console.log));
> ```
>
> no és el mateix que
>
> ```
> Promise.reject(0).catch(console.log);
> ```
>
> Aleshores, si feu
>
> ```
> let p = Promise.reject(0)
> p.then(x => 'this callback will not be called')
> p.catch(x => 'this callback will be called')
> ```
>
> el resultat del **then** serà una rejected promise que a la que no li esteu fent **catch** (veure exercici 23).

25. Fes la funció **promiseToCallback** que converteixi la funció **f** en la funció **g**, on **f** retorna el resultat en forma de promesa, mentre que **g** retorna el resultat amb un callback.

```
let g = promiseToCallback(f);
```

La funció **f** és una funció que pren un sol paràmetre i retorna una promesa. La funció **g** és una funció que pren dos paràmetres, el primer és el mateix paràmetre que rep la funció la **f** i el segon és una funció callback que cridarà amb el resultat. La funció de callback ha de fer servir la convenció d'ús **callback(err, res)**.

Exemple d'us:

```
const isEven = x => new Promise(
    (resolve, reject) => x % 2 ? reject(x) : resolve(x)
);
const isEvenCallback = promiseToCallback(isEven);
isEven(2).then(() => console.log("OK"), () => console.log("KO"));
isEvenCallback(2, (err, res) => console.log(err, res));
isEven(3).then(() => console.log("OK"), () => console.log("KO"));
isEvenCallback(3, (err, res) => console.log(err, res));
```

Resultat:

```
OK
null 2
KO
3 null
```

26. Fes la funció **readToPromise(file)** que llegeixi l'arxiu **file** amb **fs.readFile** i retorni el resultat en forma de promise.

Exemple d'us:

```
readToPromise("a1.txt").then(x => console.log("Contents: ", x))
  .catch(x => console.log("Error: ", x));
```

Resultat:

```
Contents:  <Buffer 74 65 73 74 0a>
```

Exemple d'us:

```
readToPromise("notfound.txt").then(x => console.log("Contents: ", x))
  .catch(x => console.log("Error: ", x));
```

Resultat:

```
Error:  { [Error: ENOENT: no such file or directory, open 'notfound.txt']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'notfound.txt' }
```

27. Generalitza l'exercici anterior de la mateixa manera com **asyncToFuture** generalitza la funció **readIntoFuture**.

    És a dir, crea una funció **callbackToPromise** amb la següent signatura:

    ```
    callbackToPromise = (f) => h
    ```

    a on **f** és una funció de dos paràmetres amb la mateixa convenció d'ús que **fs.readFile** i **h** és una funció que té la següent signatura:

    ```
    h(x) => Promise
    ```

    La funció **h** ha de fer el mateix que **f** però retornant el resultat amb una promise que es resoldrà amb el resultat d'executar **f**. Si **f** dona error la promesa serà rebutjada (rejected) apropiadament.

Exemple d'us:

```
const readToPromise2 = callbackToPromise(fs.readFile);
readToPromise2("a1.txt").then(x => console.log("Contents: ", x))
  .catch(x => console.log("Error: ", x));
```

Resultat:

```
Contents:  <Buffer 74 65 73 74 0a>
```

28. Fes la funció **enhancedFutureToPromise** que donat un objecte **enhancedFuture** el converteixi en una promise:

    ```
    enhancedFutureToPromise = (enhancedFuture) => Promise
    ```

    És a dir, que quan es cridi a la funció **notify** de l'**enhancedFuture**, és resolgui la promesa amb el valor **result** de l'**enhancedFuture**.

    Pista1: Noteu que la funció **notify** s'inicialitza amb una callback. Que quan cridem a la funció **notify** estem cridant aquesta callback. Ara, a més cridar de cridar la callback ens cal resoldre una promesa.

    Pista2: Pot ser us cal sobreescriure el mètode **registerCallback** per poder canviar el functionament de la funció **notify**. Recordeu que **registerCallback** rep com a paràmetre una callback que rep com a paràmetre un objecte **enhancedFuture**:

```
registerCallback(<callback(<enhancedFuture>)>)
```

Nota: Podeu importar el codi de l'exercici 18 amb el codi:

```
const readIntoEnhancedFuture = require('./18').readIntoEnhancedFuture;
```

Per a que aquesta importació funcioni us caldrà exportar la funció **readIntoEnhancedFuture** en el fitxer **18.js** seguint el patró de diseny **revealing module pattern**:

```
module.exports = {readIntoEnhancedFuture}
```

Exemple d'us:

```
const enhancedFuture = readIntoEnhancedFuture('a1.txt');
const promise = enhancedFutureToPromise(enhancedFuture);
promise.then(console.log)
enhancedFuture.registerCallback(console.log)
```

Resultat:

```
{
  isDone: true,
  result: 'contingut 1',
  notify: [Function (anonymous)],
  registerCallback: [Function (anonymous)],
  cb: [Function: log]
}
'contingut 1'
```

29. Fes la funció **mergedPromise** que, donada una promesa, retorni una altra promesa. Aquesta segona promesa sempre s'ha de resoldre i mai refusar (resolve and never reject) al valor que es resolgui o es refusi la promesa original.

Exemple d'us:

```
mergedPromise(Promise.resolve(0)).then(console.log);
mergedPromise(Promise.reject(1)).then(console.log);
```

Resultat:

```
0
1
```

30. Donades dues funcions, **f1** i **f2**, on totes dues funcions prenen un sol paràmetre i retornen una promesa, fes la funció **promiseComposer** que prengui aquestes dues funcions per paràmetre i que retorni la funció **f3**.

```
const f3 = promiseComposer(f1, f2);
```

La funció **f3** ha de retornar una promesa que ha de resoldre amb el resultat de la composició de les funcions **f1** i **f2**. És a dir **f3(x)** donaria el mateix resultat que **f1(f2(x))** si cap de les tres funcions retornessin promises. Indicació: el funcionament és el mateix que el de la funció **composer** vista anteriorment, però en aquest cas les funcions **f1**, **f2**, i **f3** retornen promises.

Exemple d'us:

```
const f1 = x => new Promise((resolve, reject) => resolve(x + 1));
promiseComposer(f1, f1)(3).then(console.log);

const f2 = x => new Promise((resolve, reject) => reject('always fails'));
promiseComposer(f1, f2)(3).catch(console.log);

let f3 = x => new Promise((resolve, reject) =>
  setTimeout(() => resolve(x * 2), 500));
promiseComposer(f1, f3)(3).then(console.log);
```

Resultat:

```
5
always fails
7
```

Nota: l'ordre dels resultat a l'exemple d'us pot canviar en funció de la implementació.

31. Fes la funció **`parallelPromise`** que rep dues promises per paràmetre i retorna una tercera promise per resultat. Aquesta tercera promise serà el resultat d'executar les dues promises per paràmetre en paral·lel i ha de resoldre a un array de dues posicions a on hi han els valors als que han resolt les promeses passades per paràmetre, en ordre (en la primera posició hi ha el valor al que resol la promesa del primer paràmetre). És a dir, l'exercici del `when(f1).and(f2).do(f3)` però amb promises.

Exemple d'us:

```
let p1 = parallelPromise(Promise.resolve(0), Promise.resolve(1));
p1.then(console.log);
```

Resultat:

```
[0, 1]
```

Exemple d'us:

```
let plast = new Promise((resolve, reject) =>
setTimeout(() => resolve('left'), 200));
let pfirst = new Promise((resolve, reject) =>
setTimeout(() => resolve('right'), 100));

let p2 = parallelPromise(plast, pfirst);
p2.then(console.log);
```

Resultat:

```
[ 'left', 'right' ]
```

Exemple d'us:

```
let plast = new Promise((resolve, reject) =>
setTimeout(() => reject('left rejected'), 200));
let pfirst = new Promise((resolve, reject) =>
setTimeout(() => reject('right rejected'), 100));

let p2 = parallelPromise(plast, pfirst);
p2.then(console.log);
```

Resultat:

```
[ 'left rejected', 'right rejected' ]
```

32. Fes la funció **`promiseBarrier`**. Aquesta funció rep per paràmetre un enter estrictament més gran que zero, i retorna una llista amb tantes funcions com indiqui el paràmetre:

```
let list = promiseBarrier(3);
```

on **`list`** és **`[f1, f2, f3]`**.

Cadascuna de les funcions de la llista resultant rebrà un sol paràmetre i retornarà una promesa que es resoldrà al valor del paràmetre. És a dir que a

```
f1(x1).then(x2 => ...)
```

les variables **x1** i **x2** sempre prendran el mateix valor.

El detall important serà que, independentment de quan és cridin cadascuna de les funcions de la llista que retorna **promiseBarrier**, aquestes només resoldran les seves promeses un cop totes les funcions s'hagin cridat.

És a dir, seguint amb l'exemple anterior, només es cridarà a la funció de callback que se li passi a **f1(x1).then(cb)** un cop **f1**, **f2** i **f3** hagin estat cridades.

La idea és que podem usar les funcions que retorna **promiseBarrier** per sincronitzar diferents cadenes de promeses.

Indicacions:

- La funció executora que se li passa a **new Promise** s'executa just quan se li passa.
- Segurament us caldrà guardar els paràmetres de les funcions executores *fora* de les funcions executores.

Nota: l'ordre dels resultats pot variar als següents dos exemples, però les linies acabades en **a** han de precedir les acabades en **b**.

Exemple d'us:

```
let [f1, f2] = promiseBarrier(2);

Promise.resolve(0)
  .then(f1)
  .then(x => { console.log("c1 s1 b"); return x; })
  .then(x => { console.log("c1 s2 b"); return x; })

Promise.resolve(0)
  .then(x => { console.log("c2 s1 a"); return x; })
  .then(x => { console.log("c2 s2 a"); return x; })
  .then(x => { console.log("c2 s3 a"); return x; })
  .then(x => { console.log("c2 s4 a"); return x; })
  .then(f2)
```

Resultat:

```
c2 s1 a
c2 s2 a
c2 s3 a
c2 s4 a
c1 s1 b
c1 s2 b
```

Exemple d'us:

```
let [f1, f2, f3] = promiseBarrier(3);

Promise.resolve(0)
  .then(x => { console.log("c1 s1 a"); return x; })
  .then(x => { console.log("c1 s2 a"); return x; })
  .then(x => { console.log("c1 s3 a"); return x; })
  .then(f1)
  .then(x => { console.log("c1 s4 b"); return x; })

Promise.resolve(0)
  .then(x => { console.log("c2 s1 a"); return x; })
  .then(f2)
  .then(x => { console.log("c2 s2 b"); return x; })

Promise.resolve(0)
  .then(f3)
  .then(x => { console.log("c3 s1 b"); return x; })
  .then(x => { console.log("c3 s2 b"); return x; })
  .then(x => { console.log("c3 s3 b"); return x; })
```

Resultat:

```
c1 s1 a
c2 s1 a
c1 s2 a
c1 s3 a
c2 s2 a
c3 s1 b
c3 s2 b
c1 s4 b
c3 s3 b
```

Exemple d'us:

```
let [f1, f2] = promiseBarrier(2);

Promise.resolve(1).then(f1).then(console.log)
Promise.resolve(2).then(f2).then(console.log)
```

Resultat:

```
1
2
```

32. Millora l'exercici anterior. Fes la funció **timedPromiseBarrier** que permeti limitar el temps màxim que estarà la barrera funcionant.

```
let list = timedPromiseBarrier(n, t);
```

Aquesta funció ha de fer el mateix que **promiseBarrier**, però si les **n** funcions no s'han cridat un cop hagin transcorregut **t** mil·lisegons, es farà el següent:

- es resoldran les promeses associades a les funcions que ja s'hagin cridat, i
- s'hauran de resoldre les promeses associades a les funcions que no s'hagin cridat tan bon punt es cridin.

Cal començar a comptar el temps el primer cop que es crida una de les funcions retornades per **timedPromiseBarrier** (i.e., no des de que s'executa **timedPromiseBarrier**).

33. Fes la funció constructora **PromisedPriorityQueue**, que garanteixi la resolució de promises d'una forma ordenada per prioritat.

```
let ppq = new PromisedPriorityQueue();
```

A una **PromisedPriorityQueue** s'hi pot afegir una promise **p** amb el metode **decorate**, i aquest mètode retornarà una segona promise **q**.

```
let p = new Promise(/*...*/)
let q = ppq.decorate(p, priority)
```

A l'afegir **p** a la cua, caldrà indicar una prioritat numèrica.

Si **p** és la promise que té la prioritat més alta de totes les que hi ha a la cua, s'ha de resoldre **q** prenent el mateix valor que **p** quan es resolgui **p**, o immediatament si **p** ja s'ha resolt. Un cop resolta **q**, s'ha d'eliminar **p** de la cua, i s'ha de procedir amb la següent promesa de més prioritat.

La prioritat d'una promise ve donada pel valor **priority**. Quan més gran, més prioritat.

Atenció: pot donar-se el cas que **p** sigui la promesa de més prioritat, però que deixi de ser-ho mentre s'espera a la seva resolució. En aquest cas, cal esperar a que la promise que ha esdevingut de més prioritat es resolgui primer.

Pista1: Com que no sabeu quan es podrà resoldre una promise **q**, pot ser us caldrà guardar la seva funció de ressolució.

Pista2: Pot ser us cal fer servir recursivitat pel procès de resoldre una promise i buscar la següent més prioritària a resoldre.

Nota: Per eliminar una posició d'un array podeu utilitzar la funció: **Array.splice**

Exemple d'us:

```
let ppq = new PromisedPriorityQueue()

p1 = new Promise((resolve, reject)=>{
  setTimeout(()=>{resolve(1)}, 1000)
})

p2 = new Promise((resolve, reject)=>{
  setTimeout(()=>{resolve(2)},2000)
})

p3 = new Promise((resolve, reject)=>{
  setTimeout(()=>{resolve(3)},3000)
})

(q1 = ppq.decorate(p1,1)).then(console.log)
(q2 = ppq.decorate(p2,2)).then(console.log)
(q3 = ppq.decorate(p3,3)).then(console.log)
```

Resultat:

```
3
2
1
```

33† Millora l'exercici anterior. Fes que si **p** es refusa, **q** sigui refusada de forma immediata.

Exemple d'us:

```
let ppq = new PromisedPriorityQueue()

//use case
p1 = new Promise((resolve, reject)=>{
  setTimeout(()=>{reject('rejected 1')}, 1000)
})

p2 = new Promise((resolve, reject)=>{
  setTimeout(()=>{resolve(2)},2000)
})

p3 = new Promise((resolve, reject)=>{
  setTimeout(()=>{resolve(3)},3000)
})

(q1 = ppq.decorate(p1,1)).then(console.log).catch(console.log)
(q2 = ppq.decorate(p2,2)).then(console.log).catch(console.log)
(q3 = ppq.decorate(p3,3)).then(console.log).catch(console.log)
```

Resultat:

```
'rejected 1'
2
1
```